Getting Started with Erlang

version 5.7

Typeset in LaTeX from SGML source using the DocBuilder-0.9.8.5 Document System.

Contents

Chapter 1

Getting Started With Erlang

1.1 Introduction

1.1.1 Introduction

This is a “kick start” tutorial to get you started with Erlang. Everything here is true, but only part of the truth. For example, I’ll only tell you the simplest form of the syntax, not all esoteric forms. Where I’ve greatly oversimplified things I’ll write *manual* which means there is lots more information to be found in the Erlang book or in the *Erlang Reference Manual*.

I also assume that this isn’t the first time you have touched a computer and you have a basic idea about how they are programmed. Don’t worry, I won’t assume you’re a wizard programmer.

1.1.2 Things Left Out

In particular the following has been omitted:

- References
- Local error handling (catch/throw)
- Single direction links (monitor)
- Handling of binary data (binaries / bit syntax)
- List comprehensions
- How to communicate with the outside world and/or software written in other languages (ports). There is however a separate tutorial for this, *Interoperability Tutorial*
- Very few of the Erlang libraries have been touched on (for example file handling)
- OTP has been totally skipped and in consequence the Mnesia database has been skipped.
- Hash tables for Erlang terms (ETS)
- Changing code in running systems

1.2 Sequential Programming

1.2.1 The Erlang Shell

Most operating systems have a command interpreter or shell, Unix and Linux have many, Windows has the Command Prompt. Erlang has its own shell where you can directly write bits of Erlang code and evaluate (run) them to see what happens (see [shell(3)]). Start the Erlang shell (in Linux or UNIX) by starting a shell or command interpreter in your operating system and typing `erl`, you will see something like this.

```
% erl
Erlang (BEAM) emulator version 5.2 [source] [hipe]

Eshell V5.2  (abort with ^G)
1>
```

Now type in “2 + 5.” as shown below.

```
1> 2 + 5.
7
2>
```

In Windows, the shell is started by double-clicking on the Erlang shell icon.

You’ll notice that the Erlang shell has numbered the lines that can be entered, (as 1> 2>) and that it has correctly told you that 2 + 5 is 7! Also notice that you have to tell it you are done entering code by finishing with a full stop “.” and a carriage return. If you make mistakes writing things in the shell, you can delete things by using the backspace key as in most shells. There are many more editing commands in the shell (See the chapter [”tty - A command line interface”] in ERTS User’s Guide).

(Note: you will find a lot of line numbers given by the shell out of sequence in this tutorial as it was written and the code tested in several sessions).

Now let’s try a more complex calculation.

```
2> (42 + 77) * 66 / 3.
2618.0
```

Here you can see the use of brackets and the multiplication operator “*” and division operator “/”, just as in normal arithmetic (see the chapter [”Arithmetic Expressions”] in the Erlang Reference Manual).

To shutdown the Erlang system and the Erlang shell type Control-C. You will see the following output:

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

Type “a” to leave the Erlang system.

Another way to shutdown the Erlang system is by entering `halt()`:

```
3> halt().
%
```

1.2.2 Modules and Functions

A programming language isn't much use if you can just run code from the shell. So here is a small Erlang program. Enter it into a file called `tut.erl` (the file name `tut.erl` is important, also make sure that it is in the same directory as the one where you started `erl`) using a suitable text editor. If you are lucky your editor will have an Erlang mode which will make it easier for you to enter and format your code nicely (see the chapter ["The Erlang mode for Emacs"] in Tools User's Guide), but you can manage perfectly well without. Here's the code to enter:

```
-module(tut).
-export([double/1]).

double(X) ->
    2 * X.
```

It's not hard to guess that this "program" doubles the value of numbers. I'll get back to the first two lines later. Let's compile the program. This can be done in your Erlang shell as shown below:

```
3> c(tut).
{ok,tut}
```

The {ok,tut} tells you that the compilation was OK. If it said "error" instead, you have made some mistake in the text you entered and there will also be error messages to give you some idea as to what has gone wrong so you can change what you have written and try again.

Now lets run the program.

```
4> tut:double(10).
20
```

As expected double of 10 is 20.

Now let's get back to the first two lines. Erlang programs are written in files. Each file contains what we call an Erlang *module*. The first line of code in the module tells us the name of the module (see the chapter ["Modules"] in the Erlang Reference Manual).

```
-module(tut).
```

This tells us that the module is called *tut*. Note the "." at the end of the line. The files which are used to store the module must have the same name as the module but with the extension ".erl". In our case the file name is `tut.erl`. When we use a function in another module, we use the syntax, `module_name:function_name(arguments)`. So

```
4> tut:double(10).
```

means call function `double` in module `tut` with argument "10".

The second line:

```
-export([double/1]).
```

says that the module `tut` contains a function called `double` which takes one argument (`X` in our example) and that this function can be called from outside the module `tut`. More about this later. Again note the "." at the end of the line.

Now for a more complicated example, the factorial of a number (e.g. factorial of 4 is 4 * 3 * 2 * 1). Enter the following code in a file called `tut1.erl`.

```
-module(tut1).
-export([fac/1]).

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

Compile the file

```
5> c(tut1).
{ok,tut1}
```

And now calculate the factorial of 4.

```
6> tut1:fac(4).
24
```

The first part:

```
fac(1) ->
    1;
```

says that the factorial of 1 is 1. Note that we end this part with a ";" which indicates that there is more of this function to come. The second part:

```
fac(N) ->
    N * fac(N - 1).
```

says that the factorial of N is N multiplied by the factorial of N - 1. Note that this part ends with a "." saying that there are no more parts of this function.

A function can have many arguments. Let's expand the module `tut1` with the rather stupid function to multiply two numbers:

```
-module(tut1).
-export([fac/1, mult/2]).

fac(1) ->
    1;
fac(N) ->
    N * fac(N - 1).

mult(X, Y) ->
    X * Y.
```

Note that we have also had to expand the `-export` line with the information that there is another function `mult` with two arguments.

Compile:

```
7> c(tut1).
{ok,tut1}
```

and try it out:

```
8> tut1:mult(3,4).
12
```

In the example above the numbers are integers and the arguments in the functions in the code, `N`, `X`, `Y` are called variables. Variables must start with a capital letter (see the chapter ["Variables"] in the Erlang Reference Manual). Examples of variable could be `Number`, `ShoeSize`, `Age` etc.

1.2.3 Atoms

Atoms are another data type in Erlang. Atoms start with a small letter ((see the chapter ["Atom"] in the Erlang Reference Manual)), for example: `charles`, `centimeter`, `inch`. Atoms are simply names, nothing else. They are not like variables which can have a value.

Enter the next program (file: `tut2.erl`) which could be useful for converting from inches to centimeters and vice versa:

```
-module(tut2).
-export([convert/2]).

convert(M, inch) ->
    M / 2.54;

convert(N, centimeter) ->
    N * 2.54.
```

Compile and test:

```
9> c(tut2).
{ok,tut2}
10> tut2:convert(3, inch).
1.1811023622047243
11> tut2:convert(7, centimeter).
17.78
```

Notice that I have introduced decimals (floating point numbers) without any explanation, but I guess you can cope with that.

See what happens if I enter something other than centimeter or inch in the convert function:

```
12> tut2:convert(3, miles).
** exception error: no function clause matching tut2:convert(3,miles)
```

The two parts of the convert function are called its clauses. Here we see that "miles" is not part of either of the clauses. The Erlang system can't *match* either of the clauses so we get an error message function_clause. The shell formats the error message nicely, but the error tuple is saved in the shell's history list and can be output by the shell command v/1:

```
13> v(12).
{'EXIT',{function_clause,[{tut2,convert,[3,miles]},
                          {erl_eval,do_apply,5},
                          {shell,exprs,6},
                          {shell,eval_exprs,6},
                          {shell,eval_loop,3}]}}
```

1.2.4 Tuples

Now the tut2 program is hardly good programming style. Consider:

```
tut2:convert(3, inch).
```

Does this mean that 3 is in inches? or that 3 is in centimeters and we want to convert it to inches? So Erlang has a way to group things together to make things more understandable. We call these *tuples*. Tuples are surrounded by "{" and "}".

So we can write {inch,3} to denote 3 inches and {centimeter,5} to denote 5 centimeters. Now let's write a new program which converts centimeters to inches and vice versa. (file tut3.erl).

```
-module(tut3).
-export([convert_length/1]).

convert_length({centimeter, X}) ->
    {inch, X / 2.54};
convert_length({inch, Y}) ->
    {centimeter, Y * 2.54}.
```

Compile and test:

```
14> c(tut3).
{ok,tut3}
15> tut3:convert_length({inch, 5}).
{centimeter,12.7}
16> tut3:convert_length(tut3:convert_length({inch, 5})).
{inch,5.0}
```

Note on line 16 we convert 5 inches to centimeters and back again and reassuringly get back to the original value. I.e the argument to a function can be the result of another function. Pause for a moment and consider how line 16 (above) works. The argument we have given the function {inch,5} is first matched against the first head clause of convert_length i.e. convert_length({centimeter,X}) where it can be seen that {centimeter,X} does not match {inch,5} (the head is the bit before the "->"). This having failed, we try the head of the next clause i.e. convert_length({inch,Y}), this matches and Y get the value 5.

We have shown tuples with two parts above, but tuples can have as many parts as we want and contain any valid Erlang *term*. For example, to represent the temperature of various cities of the world we could write

```
{moscow, {c, -10}}
{cape_town, {f, 70}}
{paris, {f, 28}}
```

Tuples have a fixed number of things in them. We call each thing in a tuple an element. So in the tuple {moscow,{c,-10}}, element 1 is moscow and element 2 is {c,-10}. I have chosen c meaning Centigrade (or Celsius) and f meaning Fahrenheit.

1.2.5 Lists

Whereas tuples group things together, we also want to be able to represent lists of things. Lists in Erlang are surrounded by "[" and "]". For example a list of the temperatures of various cities in the world could be:

```
[{moscow, {c, -10}}, {cape_town, {f, 70}}, {stockholm, {c, -4}},
 {paris, {f, 28}}, {london, {f, 36}}]
```

Note that this list was so long that it didn't fit on one line. This doesn't matter, Erlang allows line breaks at all "sensible places" but not, for example, in the middle of atoms, integers etc.

A very useful way of looking at parts of lists, is by using "|". This is best explained by an example using the shell.

```
17> [First |TheRest] = [1,2,3,4,5].
[1,2,3,4,5]
18> First.
1
19> TheRest.
[2,3,4,5]
```

We use | to separate the first elements of the list from the rest of the list. (First has got value 1 and TheRest value [2,3,4,5]).

Another example:

```
20> [E1, E2 | R] = [1,2,3,4,5,6,7].
[1,2,3,4,5,6,7]
21> E1.
1
22> E2.
2
23> R.
[3,4,5,6,7]
```

Here we see the use of | to get the first two elements from the list. Of course if we try to get more elements from the list than there are elements in the list we will get an error. Note also the special case of the list with no elements [].

```
24> [A, B | C] = [1, 2].
[1,2]
25> A.
1
26> B.
2
27> C.
[]
```

In all the examples above, I have been using new variable names, not reusing the old ones: `First`, `TheRest`, `E1`, `E2`, `R`, `A`, `B`, `C`. The reason for this is that a variable can only be given a value once in its context (scope). I'll get back to this later, it isn't so peculiar as it sounds!

The following example shows how we find the length of a list:

```
-module(tut4).

-export([list_length/1]).

list_length([]) ->
    0;
list_length([First | Rest]) ->
    1 + list_length(Rest).
```

Compile (file `tut4.erl`) and test:

```
28> c(tut4).
{ok,tut4}
29> tut4:list_length([1,2,3,4,5,6,7]).
7
```

Explanation:

```
list_length([]) ->
    0;
```

The length of an empty list is obviously 0.

```
list_length([First | Rest]) ->
    1 + list_length(Rest).
```

The length of a list with the first element `First` and the remaining elements `Rest` is 1 + the length of `Rest`.

(Advanced readers only: This is not tail recursive, there is a better way to write this function).

In general we can say we use tuples where we would use "records" or "structs" in other languages and we use lists when we want to represent things which have varying sizes, (i.e. where we would use linked lists in other languages).

Erlang does not have a string date type, instead strings can be represented by lists of ASCII characters. So the list `[97,98,99]` is equivalent to "abc". The Erlang shell is "clever" and guesses the what sort of list we mean and outputs it in what it thinks is the most appropriate form, for example:

```
30> [97,98,99].
"abc"
```

1.2.6 Standard Modules and Manual Pages

Erlang has a lot of standard modules to help you do things. For example, the module `io` contains a lot of functions to help you do formatted input/output. To look up information about standard modules, the command `erl -man` can be used at the operating shell or command prompt (i.e. at the same place as that where you started `erl`). Try the operating system shell command:

```
% erl -man io
ERLANG MODULE DEFINITION                                    io(3)

MODULE
     io - Standard I/O Server Interface Functions

DESCRIPTION
     This module provides an  interface  to  standard  Erlang  IO
     servers. The output functions all return ok if they are suc-
     ...
```

If this doesn't work on your system, the documentation is included as HTML in the Erlang/OTP release, or you can read the documentation as HTML or download it as PDF from either of the sites www.erlang.se (commercial Erlang) or www.erlang.org (open source), for example for release R9B:

```
http://www.erlang.org/doc/r9b/doc/index.html
```

1.2.7 Writing Output to a Terminal

It's nice to be able to do formatted output in these example, so the next example shows a simple way to use to use the `io:format` function. Of course, just like all other exported functions, you can test the `io:format` function in the shell:

```
31> io:format("hello world~n", []).
hello world
ok
32> io:format("this outputs one Erlang term: ~w~n", [hello]).
this outputs one Erlang term: hello
ok
33> io:format("this outputs two Erlang terms: ~w~w~n", [hello, world]).
this outputs two Erlang terms: helloworld
ok
34> io:format("this outputs two Erlang terms: ~w ~w~n", [hello, world]).
this outputs two Erlang terms: hello world
ok
```

The function `format/2` (i.e. `format` with two arguments) takes two lists. The first one is nearly always a list written between " ". This list is printed out as it stands, except that each ~w is replaced by a term taken in order from the second list. Each ~n is replaced by a new line. The `io:format/2` function itself returns the atom `ok` if everything goes as planned. Like other functions in Erlang, it crashes if an error occurs. This is not a fault in Erlang, it is a deliberate policy. Erlang has sophisticated mechanisms to handle errors which we will show later. As an exercise, try to make `io:format` crash, it shouldn't be difficult. But notice that although `io:format` crashes, the Erlang shell itself does not crash.

1.2.8 A Larger Example

Now for a larger example to consolidate what we have learnt so far. Assume we have a list of temperature readings from a number of cities in the world. Some of them are in Celsius (Centigrade) and some in Fahrenheit (as in the previous list). First let's convert them all to Celsius, then let's print out the data neatly.

```
%% This module is in file tut5.erl

-module(tut5).
-export([format_temps/1]).

%% Only this function is exported
format_temps([])->                      % No output for an empty list
    ok;
format_temps([City | Rest]) ->
    print_temp(convert_to_celsius(City)),
    format_temps(Rest).

convert_to_celsius({Name, {c, Temp}}) ->  % No conversion needed
    {Name, {c, Temp}};
convert_to_celsius({Name, {f, Temp}}) ->  % Do the conversion
    {Name, {c, (Temp - 32) * 5 / 9}}.

print_temp({Name, {c, Temp}}) ->
    io:format("~-15w ~w c~n", [Name, Temp]).

35> c(tut5).
{ok,tut5}
36> tut5:format_temps([{moscow, {c, -10}}, {cape_town, {f, 70}},
{stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
moscow          -10 c
cape_town       21.11111111111111 c
stockholm       -4 c
paris           -2.2222222222222223 c
london          2.2222222222222223 c
ok
```

Before we look at how this program works, notice that we have added a few comments to the code. A comment starts with a % character and goes on to the end of the line. Note as well that the `-export([format_temps/1]).` line only includes the function `format_temps/1`, the other functions are *local* functions, i.e. they are not visible from outside the module `tut5`.

Note as well that when testing the program from the shell, I had to spread the input over two lines as the line was too long.

When we call `format_temps` the first time, `City` gets the value `{moscow,{c,-10}}` and `Rest` is the rest of the list. So we call the function `print_temp(convert_to_celsius({moscow,{c,-10}}))`.

Here we see a function call as `convert_to_celsius({moscow,{c,-10}})` as the argument to the function `print_temp`. When we *nest* function calls like this we execute (evaluate) them from the inside out. I.e. we first evaluate `convert_to_celsius({moscow,{c,-10}})` which gives the value `{moscow,{c,-10}}` as the temperature is already in Celsius and then we evaluate `print_temp({moscow,{c,-10}})`. The function `convert_to_celsius` works in a similar way to the `convert_length` function in the previous example.

print_temp simply calls io:format in a similar way to what has been described above. Note that ~-15w says to print the "term" with a field length (width) of 15 and left justify it. ([io(3)]).

Now we call format_temps(Rest) with the rest of the list as an argument. This way of doing things is similar to the loop constructs in other languages. (Yes, this is recursion, but don't let that worry you). So the same format_temps function is called again, this time City gets the value {cape_town,{f,70}} and we repeat the same procedure as before. We go on doing this until the list becomes empty, i.e. [], which causes the first clause format_temps([]) to match. This simply returns (results in) the atom ok, so the program ends.

1.2.9 Matching, Guards and Scope of Variables

It could be useful to find the maximum and minimum temperature in lists like this. Before extending the program to do this, let's look at functions for finding the maximum value of the elements in a list:

```
-module(tut6).
-export([list_max/1]).

list_max([Head|Rest]) ->
   list_max(Rest, Head).

list_max([], Res) ->
    Res;
list_max([Head|Rest], Result_so_far) when Head > Result_so_far ->
    list_max(Rest, Head);
list_max([Head|Rest], Result_so_far)  ->
    list_max(Rest, Result_so_far).

37> c(tut6).
{ok,tut6}
38> tut6:list_max([1,2,3,4,5,7,4,3,2,1]).
7
```

First note that we have two functions here with the same name list_max. However each of these takes a different number of arguments (parameters). In Erlang these are regarded as completely different functions. Where we need to distinguish between these functions we write name/arity, where name is the name of the function and arity is the number of arguments, in this case list_max/1 and list_max/2.

This is an example where we walk through a list "carrying" a value with us, in this case Result_so_far. list_max/1 simply assumes that the max value of the list is the head of the list and calls list_max/2 with the rest of the list and the value of the head of the list, in the above this would be list_max([2,3,4,5,7,4,3,2,1],1). If we tried to use list_max/1 with an empty list or tried to use it with something which isn't a list at all, we would cause an error. Note that the Erlang philosophy is not to handle errors of this type in the function they occur, but to do so elsewhere. More about this later.

In list_max/2 we walk down the list and use Head instead of Result_so_far when Head > Result_so_far. when is a special word we use before the -> in the function to say that we should only use this part of the function if the test which follows is true. We call tests of this type a *guard*. If the guard isn't true (we say the guard fails), we try the next part of the function. In this case if Head isn't greater than Result_so_far then it must be smaller or equal to is, so we don't need a guard on the next part of the function.

Some useful operators in guards are, < less than, > greater than, == equal, >= greater or equal, =< less or equal, /= not equal. (see the chapter ["Guard Sequences"] in the Erlang Reference Manual).

To change the above program to one which works out the minimum value of the element in a list, all we would need to do is to write < instead of >. (But it would be wise to change the name of the function to `list_min` :-).

Remember that I mentioned earlier that a variable could only be given a value once in its scope? In the above we see, for example, that `Result_so_far` has been given several values. This is OK since every time we call `list_max/2` we create a new scope and one can regard the `Result_so_far` as a completely different variable in each scope.

Another way of creating and giving a variable a value is by using the match operator = . So if I write `M = 5`, a variable called `M` will be created and given the value 5. If, in the same scope I then write `M = 6`, I'll get an error. Try this out in the shell:

```
39> M = 5.
5
40> M = 6.
** exception error: no match of right hand side value 6
41> M = M + 1.
** exception error: no match of right hand side value 6
42> N = M + 1.
6
```

The use of the match operator is particularly useful for pulling apart Erlang terms and creating new ones.

```
43> {X, Y} = {paris, {f, 28}}.
{paris,{f,28}}
44> X.
paris
45> Y.
{f,28}
```

Here we see that `X` gets the value `paris` and `Y`{`f`,`28`}.

Of course if we try to do the same again with another city, we get an error:

```
46> {X, Y} = {london, {f, 36}}.
** exception error: no match of right hand side value {london,{f,36}}
```

Variables can also be used to improve the readability of programs, for example, in the `list_max/2` function above, we could write:

```
list_max([Head|Rest], Result_so_far) when Head > Result_so_far ->
    New_result_far = Head,
    list_max(Rest, New_result_far);
```

which is possibly a little clearer.

1.2.10 More About Lists

Remember that the | operator can be used to get the head of a list:

```
47> [M1|T1] = [paris, london, rome].
[paris,london,rome]
48> M1.
paris
49> T1.
[london,rome]
```

The | operator can also be used to add a head to a list:

```
50> L1 = [madrid | T1].
[madrid,london,rome]
51> L1.
[madrid,london,rome]
```

Now an example of this when working with lists - reversing the order of a list:

```
-module(tut8).

-export([reverse/1]).

reverse(List) ->
    reverse(List, []).

reverse([Head | Rest], Reversed_List) ->
    reverse(Rest, [Head | Reversed_List]);
reverse([], Reversed_List) ->
    Reversed_List.
```

```
52> c(tut8).
{ok,tut8}
53> tut8:reverse([1,2,3]).
[3,2,1]
```

Consider how `Reversed_List` is built. It starts as [], we then successively take off the heads of the list to be reversed and add them to the the `Reversed_List`, as shown in the following:

```
reverse([1|2,3], []) =>
    reverse([2,3], [1|[]])

reverse([2|3], [1]) =>
    reverse([3], [2|[1])

reverse([3|[]], [2,1]) =>
    reverse([], [3|[2,1]])

reverse([], [3,2,1]) =>
    [3,2,1]
```

The module `lists` contains a lot of functions for manipulating lists, for example for reversing them, so before you write a list manipulating function it is a good idea to check that one isn't already written for you. (see [lists(3)]).

Now lets get back to the cities and temperatures, but take a more structured approach this time. First let's convert the whole list to Celsius as follows and test the function:

```
-module(tut7).
-export([format_temps/1]).

format_temps(List_of_cities) ->
    convert_list_to_c(List_of_cities).

convert_list_to_c([{Name, {f, F}} | Rest]) ->
    Converted_City = {Name, {c, (F -32)* 5 / 9}},
    [Converted_City | convert_list_to_c(Rest)];

convert_list_to_c([City | Rest]) ->
    [City | convert_list_to_c(Rest)];

convert_list_to_c([]) ->
    [].
```

```
54> c(tut7).
{ok, tut7}.
55> tut7:format_temps([{moscow, {c, -10}}, {cape_town, {f, 70}},
{stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
[{moscow,{c,-10}},
 {cape_town,{c,21.11111111111111}},
 {stockholm,{c,-4}},
 {paris,{c,-2.2222222222222223}},
 {london,{c,2.2222222222222223}}]
```

Looking at this bit by bit:

```
format_temps(List_of_cities) ->
    convert_list_to_c(List_of_cities).
```

Here we see that `format_temps/1` calls `convert_list_to_c/1`. `convert_list_to_c/1` takes off the head of the `List_of_cities`, converts it to Celsius if needed. The `|` operator is used to add the (maybe) converted to the converted rest of the list:

```
[Converted_City | convert_list_to_c(Rest)];
```

or

```
[City | convert_list_to_c(Rest)];
```

We go on doing this until we get to the end of the list (i.e. the list is empty:

```
convert_list_to_c([]) ->
    [].
```

Now we have converted the list, we add a function to print it:

```
-module(tut7).
-export([format_temps/1]).

format_temps(List_of_cities) ->
    Converted_List = convert_list_to_c(List_of_cities),
    print_temp(Converted_List).

convert_list_to_c([{Name, {f, F}} | Rest]) ->
    Converted_City = {Name, {c, (F -32)* 5 / 9}},
    [Converted_City | convert_list_to_c(Rest)];

convert_list_to_c([City | Rest]) ->
    [City | convert_list_to_c(Rest)];

convert_list_to_c([]) ->
    [].

print_temp([{Name, {c, Temp}} | Rest]) ->
    io:format("~-15w ~w c~n", [Name, Temp]),
    print_temp(Rest);
print_temp([]) ->
    ok.

56> c(tut7).
{ok,tut7}
57> tut7:format_temps([{moscow, {c, -10}}, {cape_town, {f, 70}},
{stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
moscow          -10 c
cape_town       21.11111111111111 c
stockholm       -4 c
paris           -2.2222222222222223 c
london          2.2222222222222223 c
ok
```

We now have to add a function to find the cities with the maximum and minimum temperatures. The program below isn't the most efficient way of doing this as we walk through the list of cities four times. But it is better to first strive for clarity and correctness and to make programs efficient only if really needed.

```
-module(tut7).
-export([format_temps/1]).

format_temps(List_of_cities) ->
    Converted_List = convert_list_to_c(List_of_cities),
    print_temp(Converted_List),
    {Max_city, Min_city} = find_max_and_min(Converted_List),
    print_max_and_min(Max_city, Min_city).

convert_list_to_c([{Name, {f, Temp}} | Rest]) ->
    Converted_City = {Name, {c, (Temp -32)* 5 / 9}},
```

```
    [Converted_City | convert_list_to_c(Rest)];

convert_list_to_c([City | Rest]) ->
    [City | convert_list_to_c(Rest)];

convert_list_to_c([]) ->
    [].

print_temp([{Name, {c, Temp}} | Rest]) ->
    io:format("~-15w ~w c~n", [Name, Temp]),
    print_temp(Rest);
print_temp([]) ->
    ok.

find_max_and_min([City | Rest]) ->
    find_max_and_min(Rest, City, City).

find_max_and_min([{Name, {c, Temp}} | Rest],
         {Max_Name, {c, Max_Temp}},
         {Min_Name, {c, Min_Temp}}) ->
    if
        Temp > Max_Temp ->
            Max_City = {Name, {c, Temp}};           % Change
        true ->
            Max_City = {Max_Name, {c, Max_Temp}} % Unchanged
    end,
    if
         Temp < Min_Temp ->
            Min_City = {Name, {c, Temp}};           % Change
        true ->
            Min_City = {Min_Name, {c, Min_Temp}} % Unchanged
    end,
    find_max_and_min(Rest, Max_City, Min_City);

find_max_and_min([], Max_City, Min_City) ->
    {Max_City, Min_City}.

print_max_and_min({Max_name, {c, Max_temp}}, {Min_name, {c, Min_temp}}) ->
    io:format("Max temperature was ~w c in ~w~n", [Max_temp, Max_name]),
    io:format("Min temperature was ~w c in ~w~n", [Min_temp, Min_name]).

58> c(tut7).
{ok, tut7}
59> tut7:format_temps([{moscow, {c, -10}}, {cape_town, {f, 70}},
{stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
moscow          -10 c
cape_town       21.1111111111111 c
stockholm       -4 c
paris           -2.2222222222222223 c
london          2.2222222222222223 c
Max temperature was 21.1111111111111 c in cape_town
Min temperature was -10 c in moscow
ok
```

1.2.11 If and Case

The function `find_max_and_min` works out the maximum and minimum temperature. We have introduced a new construct here `if`. If works as follows:

```
if
    Condition 1 ->
        Action 1;
    Condition 2 ->
        Action 2;
    Condition 3 ->
        Action 3;
    Condition 4 ->
        Action 4
end
```

Note there is no ";" before `end`! Conditions are the same as guards, tests which succeed or fail. Erlang starts at the top until it finds a condition which succeeds and then it evaluates (performs) the action following the condition and ignores all other conditions and action before the `end`. If no condition matches, there will be a run-time failure. A condition which always is succeeds is the atom, `true` and this is often used last in an `if` meaning do the action following the `true` if all other conditions have failed.

The following is a short program to show the workings of `if`.

```
-module(tut9).
-export([test_if/2]).

test_if(A, B) ->
    if
        A == 5 ->
            io:format("A == 5~n", []),
            a_equals_5;
        B == 6 ->
            io:format("B == 6~n", []),
            b_equals_6;
        A == 2, B == 3 ->                       %i.e. A equals 2 and B equals 3
            io:format("A == 2, B == 3~n", []),
            a_equals_2_b_equals_3;
        A == 1 ; B == 7 ->                      %i.e. A equals 1 or B equals 7
            io:format("A == 1 ; B == 7~n", []),
            a_equals_1_or_b_equals_7
    end.
```

Testing this program gives:

```
60> c(tut9).
{ok,tut9}
61> tut9:test_if(5,33).
A == 5
a_equals_5
62> tut9:test_if(33,6).
B == 6
```

```
b_equals_6
63> tut9:test_if(2, 3).
A == 2, B == 3
a_equals_2_b_equals_3
64> tut9:test_if(1, 33).
A == 1 ; B == 7
a_equals_1_or_b_equals_7
65> tut9:test_if(33, 7).
A == 1 ; B == 7
a_equals_1_or_b_equals_7
66> tut9:test_if(33, 33).
** exception error: no true branch found when evaluating an if expression
     in function  tut9:test_if/2
```

Notice that `tut9:test_if(33,33)` did not cause any condition to succeed so we got the run time error `if_clause`, here nicely formatted by the shell. See the chapter ["Guard Sequences"] in the Erlang Reference Manual for details of the many guard tests available. `case` is another construct in Erlang. Recall that we wrote the `convert_length` function as:

```
convert_length({centimeter, X}) ->
    {inch, X / 2.54};
convert_length({inch, Y}) ->
    {centimeter, Y * 2.54}.
```

We could also write the same program as:

```
-module(tut10).
-export([convert_length/1]).

convert_length(Length) ->
    case Length of
        {centimeter, X} ->
            {inch, X / 2.54};
        {inch, Y} ->
            {centimeter, Y * 2.54}
    end.
```

```
67> c(tut10).
{ok,tut10}
68> tut10:convert_length({inch, 6}).
{centimeter,15.24}
69> tut10:convert_length({centimeter, 2.5}).
{inch,0.984251968503937}
```

Notice that both `case` and `if` have *return values*, i.e. in the above example `case` returned either `{inch,X/2.54}` or `{centimeter,Y*2.54}`. The behaviour of `case` can also be modified by using guards. An example should hopefully clarify this. The following example tells us the length of a month, given the year. We need to know the year of course, since February has 29 days in a leap year.

```
-module(tut11).
-export([month_length/2]).

month_length(Year, Month) ->
    %% All years divisible by 400 are leap
    %% Years divisible by 100 are not leap (except the 400 rule above)
    %% Years divisible by 4 are leap (except the 100 rule above)
    Leap = if
        trunc(Year / 400) * 400 == Year ->
            leap;
        trunc(Year / 100) * 100 == Year ->
            not_leap;
        trunc(Year / 4) * 4 == Year ->
            leap;
        true ->
            not_leap
    end,
    case Month of
        sep -> 30;
        apr -> 30;
        jun -> 30;
        nov -> 30;
        feb when Leap == leap -> 29;
        feb -> 28;
        jan -> 31;
        mar -> 31;
        may -> 31;
        jul -> 31;
        aug -> 31;
        oct -> 31;
        dec -> 31
    end.
```

```
70> c(tut11).
{ok,tut11}
71> tut11:month_length(2004, feb).
29
72> tut11:month_length(2003, feb).
28
73> tut11:month_length(1947, aug).
31
```

1.2.12 Built In Functions (BIFs)

Built in functions BIFs are functions which for some reason is built in to the Erlang virtual machine. BIFs often implement functionality that is impossible to implement in Erlang or is to inefficient to implement in Erlang. Some BIFs can be called by use of the function name only but they are by default belonging to the erlang module so for example the call to the BIF `trunc` below is equivalent with a call to `erlang:trunc`.

As you can see, we first find out if a year is leap or not. If a year is divisible by 400, it is a leap year. To find this out we first divide the year by 400 and use the built in function `trunc` (more later) to cut off

any decimals. We then multiply by 400 again and see if we get back the same value. For example, year 2004:

```
2004 / 400 = 5.01
trunc(5.01) = 5
5 * 400 = 2000
```

and we can see that we got back 2000 which is not the same as 2004, so 2004 isn't divisible by 400. Year 2000:

```
2000 / 400 = 5.0
trunc(5.0) = 5
5 * 400 = 2000
```

so we have a leap year. The next two tests if the year is divisible by 100 or 4 are done in the same way. The first `if` returns `leap` or `not_leap` which lands up in the variable `Leap`. We use this variable in the guard for `feb` in the following `case` which tells us how long the month is.

This example showed the use of `trunc`, an easier way would be to use the Erlang operator `rem` which gives the remainder after division. For example:

```
74> 2004 rem 400.
4
```

so instead of writing

```
trunc(Year / 400) * 400 == Year ->
    leap;
```

we could write

```
Year rem 400 == 0 ->
    leap;
```

There are many other built in functions (BIF) such as `trunc`. Only a few built in functions can be used in guards, and you cannot use functions you have defined yourself in guards. (see the chapter ["Guard Sequences"] in the Erlang Reference Manual) (Aside for advanced readers: This is to ensure that guards don't have side effects). Let's play with a few of these functions in the shell:

```
75> trunc(5.6).
5
76> round(5.6).
6
77> length([a,b,c,d]).
4
78> float(5).
5.0
79> is_atom(hello).
true
80> is_atom("hello").
false
81> is_tuple({paris, {c, 30}}).
true
82> is_tuple([paris, {c, 30}]).
false
```

All the above can be used in guards. Now for some which can't be used in guards:

```
83> atom_to_list(hello).
"hello"
84> list_to_atom("goodbye").
goodbye
85> integer_to_list(22).
"22"
```

The 3 BIFs above do conversions which would be difficult (or impossible) to do in Erlang.

1.2.13 Higher Order Functions (Funs)

Erlang, like most modern functional programming languages, has higher order functions. We start with an example using the shell:

```
86> Xf = fun(X) -> X * 2 end.
#Fun<erl_eval.5.123085357>
87> Xf(5).
10
```

What we have done here is to define a function which doubles the value of number and assign this function to a variable. Thus `Xf(5)` returned the value 10. Two useful functions when working with lists are `foreach` and `map`, which are defined as follows:

```
foreach(Fun, [First|Rest]) ->
    Fun(First),
    foreach(Fun, Rest);
foreach(Fun, []) ->
    ok.

map(Fun, [First|Rest]) ->
    [Fun(First)|map(Fun,Rest)];
map(Fun, []) ->
    [].
```

These two functions are provided in the standard module `lists`. `foreach` takes a list and applies a fun to every element in the list, `map` creates a new list by applying a fun to every element in a list. Going back to the shell, we start by using `map` and a fun to add 3 to every element of a list:

```
88> Add_3 = fun(X) -> X + 3 end.
#Fun<erl_eval.5.123085357>
89> lists:map(Add_3, [1,2,3]).
[4,5,6]
```

Now lets print out the temperatures in a list of cities (yet again):

```
90> Print_City = fun({City, {X, Temp}}) -> io:format("~-15w ~w ~w~n",
[City, X, Temp]) end.
#Fun<erl_eval.5.123085357>
91> lists:foreach(Print_City, [{moscow, {c, -10}}, {cape_town, {f, 70}},
{stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
moscow          c -10
cape_town       f 70
stockholm       c -4
paris           f 28
london          f 36
ok
```

We will now define a fun which can be used to go through a list of cities and temperatures and transform them all to Celsius.

```
-module(tut13).

-export([convert_list_to_c/1]).

convert_to_c({Name, {f, Temp}}) ->
    {Name, {c, trunc((Temp - 32) * 5 / 9)}};
convert_to_c({Name, {c, Temp}}) ->
    {Name, {c, Temp}}.

convert_list_to_c(List) ->
    lists:map(fun convert_to_c/1, List).
```

```
92> tut13:convert_list_to_c([{moscow, {c, -10}}, {cape_town, {f, 70}},
{stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
[{moscow,{c,-10}},
 {cape_town,{c,21}},
 {stockholm,{c,-4}},
 {paris,{c,-2}},
 {london,{c,2}}]
```

The `convert_to_c` function is the same as before, but we use it as a fun:

```
lists:map(fun convert_to_c/1, List)
```

When we use a function defined elsewhere as a fun we can refer to it as `Function/Arity` (remember that `Arity` = number of arguments). So in the `map` call we write `lists:map(fun convert_to_c/1, List)`. As you can see `convert_list_to_c` becomes much shorter and easier to understand.

The standard module `lists` also contains a function `sort(Fun, List)` where `Fun` is a fun with two arguments. This fun should return `true` if the the first argument is less than the second argument, or else `false`. We add sorting to the `convert_list_to_c`:

```
-module(tut13).

-export([convert_list_to_c/1]).

convert_to_c({Name, {f, Temp}}) ->
    {Name, {c, trunc((Temp - 32) * 5 / 9)}};
```

```
convert_to_c({Name, {c, Temp}}) ->
    {Name, {c, Temp}}.

convert_list_to_c(List) ->
    New_list = lists:map(fun convert_to_c/1, List),
    lists:sort(fun({_, {c, Temp1}}, {_, {c, Temp2}}) ->
                       Temp1 < Temp2 end, New_list).
```

```
93> c(tut13).
{ok,tut13}
94> tut13:convert_list_to_c([{moscow, {c, -10}}, {cape_town, {f, 70}},
{stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
[{moscow,{c,-10}},
 {stockholm,{c,-4}},
 {paris,{c,-2}},
 {london,{c,2}},
 {cape_town,{c,21}}]
```

In `sort` we use the fun:

```
fun({_, {c, Temp1}}, {_, {c, Temp2}}) -> Temp1 < Temp2 end,
```

Here we introduce the concept of an *anonymous variable* "_". This is simply shorthand for a variable which is going to get a value, but we will ignore the value. This can be used anywhere suitable, not just in fun's. `Temp1 < Temp2` returns `true` if `Temp1` is less than `Temp2`.

1.3 Concurrent Programming

1.3.1 Processes

One of the main reasons for using Erlang instead of other functional languages is Erlang's ability to handle concurrency and distributed programming. By concurrency we mean programs which can handle several threads of execution at the same time. For example, modern operating systems would allow you to use a word processor, a spreadsheet, a mail client and a print job all running at the same time. Of course each processor (CPU) in the system is probably only handling one thread (or job) at a time, but it swaps between the jobs a such a rate that it gives the illusion of running them all at the same time. It is easy to create parallel threads of execution in an Erlang program and it is easy to allow these threads to communicate with each other. In Erlang we call each thread of execution a *process*.

(Aside: the term "process" is usually used when the threads of execution share no data with each other and the term "thread" when they share data in some way. Threads of execution in Erlang share no data, that's why we call them processes).

The Erlang BIF `spawn` is used to create a new process: `spawn(Module, Exported_Function, List of Arguments)`. Consider the following module:

```
-module(tut14).

-export([start/0, say_something/2]).

say_something(What, 0) ->
    done;
```

```
say_something(What, Times) ->
    io:format("~p~n", [What]),
    say_something(What, Times - 1).

start() ->
    spawn(tut14, say_something, [hello, 3]),
    spawn(tut14, say_something, [goodbye, 3]).
```

```
5> c(tut14).
{ok,tut14}
6> tut14:say_something(hello, 3).
hello
hello
hello
done
```

We can see that function `say_something` writes its first argument the number of times specified by second argument. Now look at the function `start`. It starts two Erlang processes, one which writes "hello" three times and one which writes "goodbye" three times. Both of these processes use the function `say_something`. Note that a function used in this way by `spawn` to start a process must be exported from the module (i.e. in the `-export` at the start of the module).

```
9> tut14:start().
hello
goodbye
<0.63.0>
hello
goodbye
hello
goodbye
```

Notice that it didn't write "hello" three times and then "goodbye" three times, but the first process wrote a "hello", the second a "goodbye", the first another "hello" and so forth. But where did the <0.63.0> come from? The return value of a function is of course the return value of the last "thing" in the function. The last thing in the function `start` is

```
spawn(tut14, say_something, [goodbye, 3]).
```

`spawn` returns a *process identifier*, or *pid*, which uniquely identifies the process. So <0.63.0> is the pid of the `spawn` function call above. We will see how to use pids in the next example.

Note as well that we have used `~p` instead of `~w` in `io:format`. To quote the manual: "`~p` Writes the data with standard syntax in the same way as `~w`, but breaks terms whose printed representation is longer than one line into many lines and indents each line sensibly. It also tries to detect lists of printable characters and to output these as strings".

1.3.2 Message Passing

In the following example we create two processes which send messages to each other a number of times.

```
-module(tut15).

-export([start/0, ping/2, pong/0]).

ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished~n", []);

ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_PID).

pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    Pong_PID = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_PID]).
```

```
1> c(tut15).
{ok,tut15}
2> tut15: start().
<0.36.0>
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
ping finished
Pong finished
```

The function `start` first creates a process, let's call it "pong":

```
Pong_PID = spawn(tut15, pong, [])
```

This process executes `tut15:pong()`. `Pong_PID` is the process identity of the "pong" process. The function `start` now creates another process "ping".

```
spawn(tut15, ping, [3, Pong_PID]),
```

this process executes

```
tut15:ping(3, Pong_PID)
```

<0.36.0> is the return value from the `start` function.

The process “pong” now does:

```
receive
    finished ->
        io:format("Pong finished~n", []);
    {ping, Ping_PID} ->
        io:format("Pong received ping~n", []),
        Ping_PID ! pong,
        pong()
end.
```

The `receive` construct is used to allow processes to wait for messages from other processes. It has the format:

```
receive
   pattern1 ->
       actions1;
   pattern2 ->
       actions2;
   ....
   patternN
       actionsN
end.
```

Note: no “;” before the `end`.

Messages between Erlang processes are simply valid Erlang terms. I.e. they can be lists, tuples, integers, atoms, pids etc.

Each process has its own input queue for messages it receives. New messages received are put at the end of the queue. When a process executes a `receive`, the first message in the queue is matched against the first pattern in the `receive`, if this matches, the message is removed from the queue and the actions corresponding to the the pattern are executed.

However, if the first pattern does not match, the second pattern is tested, if this matches the message is removed from the queue and the actions corresponding to the second pattern are executed. If the second pattern does not match the third is tried and so on until there are no more pattern to test. If there are no more patterns to test, the first message is kept in the queue and we try the second message instead. If this matches any pattern, the appropriate actions are executed and the second message is removed from the queue (keeping the first message and any other messages in the queue). If the second message does not match we try the third message and so on until we reach the end of the queue. If we reach the end of the queue, the process blocks (stops execution) and waits until a new message is received and this procedure is repeated.

Of course the Erlang implementation is “clever” and minimizes the number of times each message is tested against the patterns in each `receive`.

Now back to the ping pong example.

“Pong” is waiting for messages. If the atom `finished` is received, “pong” writes “Pong finished” to the output and as it has nothing more to do, terminates. If it receives a message with the format:

```
{ping, Ping_PID}
```

it writes “Pong received ping” to the output and sends the atom pong to the process “ping”:

```
Ping_PID ! pong
```

Note how the operator “!” is used to send messages. The syntax of “!” is:

```
Pid ! Message
```

I.e. Message (any Erlang term) is sent to the process with identity Pid.

After sending the message pong, to the process “ping”, “pong” calls the pong function again, which causes it to get back to the receive again and wait for another message. Now let’s look at the process “ping”. Recall that it was started by executing:

```
tut15:ping(3, Pong_PID)
```

Looking at the function ping/2 we see that the second clause of ping/2 is executed since the value of the first argument is 3 (not 0) (first clause head is ping(0,Pong_PID), second clause head is ping(N,Pong_PID), so N becomes 3).

The second clause sends a message to “pong”:

```
Pong_PID ! {ping, self()},
```

self() returns the pid of the process which executes self(), in this case the pid of “ping”. (Recall the code for “pong”, this will land up in the variable Ping_PID in the receive previously explained).

“Ping” now waits for a reply from “pong”:

```
receive
    pong ->
        io:format("Ping received pong~n", [])
end,
```

and writes “Ping received pong” when this reply arrives, after which “ping” calls the ping function again.

```
ping(N - 1, Pong_PID)
```

N-1 causes the first argument to be decremented until it becomes 0. When this occurs, the first clause of ping/2 will be executed:

```
ping(0, Pong_PID) ->
    Pong_PID !  finished,
    io:format("ping finished~n", []);
```

The atom finished is sent to “pong” (causing it to terminate as described above) and “ping finished” is written to the output. “Ping” then itself terminates as it has nothing left to do.

1.3.3 Registered Process Names

In the above example, we first created “pong” so as to be able to give the identity of “pong” when we started “ping”. I.e. in some way “ping” must be able to know the identity of “pong” in order to be able to send a message to it. Sometimes processes which need to know each others identities are started completely independently of each other. Erlang thus provides a mechanism for processes to be given names so that these names can be used as identities instead of pids. This is done by using the `register` BIF:

```
register(some_atom, Pid)
```

We will now re-write the ping pong example using this and giving the name `pong` to the “pong” process:

```
-module(tut16).

-export([start/0, ping/1, pong/0]).

ping(0) ->
    pong ! finished,
    io:format("ping finished~n", []);

ping(N) ->
    pong ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1).

pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    register(pong, spawn(tut16, pong, [])),
    spawn(tut16, ping, [3]).
```

```
2> c(tut16).
{ok, tut16}
3> tut16:start().
<0.38.0>
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
```

```
ping finished
Pong finished
```

In the `start/0` function,

```
register(pong, spawn(tut16, pong, [])),
```

both spawns the “pong” process and gives it the name `pong`. In the “ping” process we can now send messages to `pong` by:

```
pong ! {ping, self()},
```

so that `ping/2` now becomes `ping/1` as we don’t have to use the argument `Pong_PID`.

1.3.4 Distributed Programming

Now let’s re-write the ping pong program with “ping” and “pong” on different computers. Before we do this, there are a few things we need to set up to get this to work. The distributed Erlang implementation provides a basic security mechanism to prevent unauthorized access to an Erlang system on another computer (*manual*). Erlang systems which talk to each other must have the same *magic cookie*. The easiest way to achieve this is by having a file called `.erlang.cookie` in your home directory on all machines which on which you are going to run Erlang systems communicating with each other (on Windows systems the home directory is the directory where pointed to by the $HOME environment variable - you may need to set this. On Linux or Unix you can safely ignore this and simply create a file called `.erlang.cookie` in the directory you get to after executing the command `cd` without any argument). The `.erlang.cookie` file should contain on line with the same atom. For example on Linux or Unix in the OS shell:

```
$ cd
$ cat > .erlang.cookie
this_is_very_secret
$ chmod 400 .erlang.cookie
```

The `chmod` above make the `.erlang.cookie` file accessible only by the owner of the file. This is a requirement.

When you start an Erlang system which is going to talk to other Erlang systems, you must give it a name, eg:

```
$ erl -sname my_name
```

We will see more details of this later (*manual*). If you want to experiment with distributed Erlang, but you only have one computer to work on, you can start two separate Erlang systems on the same computer but give them different names. Each Erlang system running on a computer is called an Erlang node.

(Note: `erl -sname` assumes that all nodes are in the same IP domain and we can use only the first component of the IP address, if we want to use nodes in different domains we use `-name` instead, but then all IP address must be given in full (*manual*).

Here is the ping pong example modified to run on two separate nodes:

```
-module(tut17).

-export([start_ping/1, start_pong/0,  ping/2, pong/0]).

ping(0, Pong_Node) ->
    {pong, Pong_Node} ! finished,
    io:format("ping finished~n", []);

ping(N, Pong_Node) ->
    {pong, Pong_Node} ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_Node).

pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start_pong() ->
    register(pong, spawn(tut17, pong, [])).

start_ping(Pong_Node) ->
    spawn(tut17, ping, [3, Pong_Node]).
```

Let us assume we have two computers called gollum and kosken. We will start a node on kosken called ping and then a node on gollum called pong.

On kosken (on a Linux/Unix system):

```
kosken> erl -sname ping
Erlang (BEAM) emulator version 5.2.3.7 [hipe] [threads:0]

Eshell V5.2.3.7  (abort with ^G)
(ping@kosken)1>
```

On gollum:

```
gollum> erl -sname pong
Erlang (BEAM) emulator version 5.2.3.7 [hipe] [threads:0]

Eshell V5.2.3.7  (abort with ^G)
(pong@gollum)1>
```

Now we start the “pong” process on gollum:

```
(pong@gollum)1> tut17:start_pong().
true
```

and start the "ping" process on kosken (from the code above you will see that a parameter of the `start_ping` function is the node name of the Erlang system where "pong" is running):

```
(ping@kosken)1> tut17:start_ping(pong@gollum).
<0.37.0>
Ping received pong
Ping received pong
Ping received pong
ping finished
```

Here we see that the ping pong program has run, on the "pong" side we see:

```
(pong@gollum)2>
Pong received ping
Pong received ping
Pong received ping
Pong finished
(pong@gollum)2>
```

Looking at the `tut17` code we see that the `pong` function itself is unchanged, the lines:

```
{ping, Ping_PID} ->
    io:format("Pong received ping~n", []),
    Ping_PID ! pong,
```

work in the same way irrespective of on which node the "ping" process is executing. Thus Erlang pids contain information about where the process executes so if you know the pid of a process, the "!" operator can be used to send it a message if the process is on the same node or on a different node.

A difference is how we send messages to a registered process on another node:

```
{pong, Pong_Node} ! {ping, self()},
```

We use a tuple {`registered_name`,`node_name`} instead of just the `registered_name`.

In the previous example, we started "ping" and "pong" from the shells of two separate Erlang nodes. `spawn` can also be used to start processes in other nodes. The next example is the ping pong program, yet again, but this time we will start "ping" in another node:

```
-module(tut18).

-export([start/1,  ping/2, pong/0]).

ping(0, Pong_Node) ->
    {pong, Pong_Node} ! finished,
    io:format("ping finished~n", []);

ping(N, Pong_Node) ->
    {pong, Pong_Node} ! {ping, self()},
    receive
```

```
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_Node).

pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start(Ping_Node) ->
    register(pong, spawn(tut18, pong, [])),
    spawn(Ping_Node, tut18, ping, [3, node()]).
```

Assuming an Erlang system called ping (but not the “ping” process) has already been started on kosken, then on gollum we do:

```
(pong@gollum)1> tut18:start(ping@kosken).
<3934.39.0>
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong finished
ping finished
```

Notice we get all the output on gollum. This is because the io system finds out where the process is spawned from and sends all output there.

1.3.5 A Larger Example

Now for a larger example. We will make an extremely simple “messenger”. The messenger is a program which allows users to log in on different nodes and send simple messages to each other.

Before we start, let’s note the following:

- This example will just show the message passing logic no attempt at all has been made to provide a nice graphical user interface - this can of course also be done in Erlang - but that’s another tutorial.
- This sort of problem can be solved more easily if you use the facilities in OTP, which will also provide methods for updating code on the fly etc. But again, that’s another tutorial.
- The first program we write will contain some inadequacies as regards handling of nodes which disappear, we will correct these in a later version of the program.

We will set up the messenger by allowing “clients” to connect to a central server and say who and where they are. I.e. a user won’t need to know the name of the Erlang node where another user is located to send a message.

File `messenger.erl`:

```
%%% Message passing utility.
%%% User interface:
%%% logon(Name)
%%%     One user at a time can log in from each Erlang node in the
%%%     system messenger: and choose a suitable Name. If the Name
%%%     is already logged in at another node or if someone else is
%%%     already logged in at the same node, login will be rejected
%%%     with a suitable error message.
%%% logoff()
%%%     Logs off anybody at at node
%%% message(ToName, Message)
%%%     sends Message to ToName. Error messages if the user of this
%%%     function is not logged on or if ToName is not logged on at
%%%     any node.
%%%
%%% One node in the network of Erlang nodes runs a server which maintains
%%% data about the logged on users. The server is registered as "messenger"
%%% Each node where there is a user logged on runs a client process registered
%%% as "mess_client"
%%%
%%% Protocol between the client processes and the server
%%% ----------------------------------------------------
%%%
%%% To server: {ClientPid, logon, UserName}
%%% Reply {messenger, stop, user_exists_at_other_node} stops the client
%%% Reply {messenger, logged_on} logon was successful
%%%
%%% To server: {ClientPid, logoff}
%%% Reply: {messenger, logged_off}
%%%
%%% To server: {ClientPid, logoff}
%%% Reply: no reply
%%%
%%% To server: {ClientPid, message_to, ToName, Message} send a message
%%% Reply: {messenger, stop, you_are_not_logged_on} stops the client
%%% Reply: {messenger, receiver_not_found} no user with this name logged on
%%% Reply: {messenger, sent} Message has been sent (but no guarantee)
%%%
%%% To client: {message_from, Name, Message},
%%%
%%% Protocol between the "commands" and the client
%%% ----------------------------------------------
%%%
%%% Started: messenger:client(Server_Node, Name)
%%% To client: logoff
%%% To client: {message_to, ToName, Message}
%%%
```

```
%%% Configuration: change the server_node() function to return the
%%% name of the node where the messenger server runs

-module(messenger).
-export([start_server/0, server/1, logon/1, logoff/0, message/2, client/2]).

%%% Change the function below to return the name of the node where the
%%% messenger server runs
server_node() ->
    messenger@bill.

%%% This is the server process for the "messenger"
%%% the user list has the format [{ClientPid1, Name1},{ClientPid22, Name2},...]
server(User_List) ->
    receive
        {From, logon, Name} ->
            New_User_List = server_logon(From, Name, User_List),
            server(New_User_List);
        {From, logoff} ->
            New_User_List = server_logoff(From, User_List),
            server(New_User_List);
        {From, message_to, To, Message} ->
            server_transfer(From, To, Message, User_List),
            io:format("list is now: ~p~n", [User_List]),
            server(User_List)
    end.

%%% Start the server
start_server() ->
    register(messenger, spawn(messenger, server, [[]])).


%%% Server adds a new user to the user list
server_logon(From, Name, User_List) ->
    %% check if logged on anywhere else
    case lists:keymember(Name, 2, User_List) of
        true ->
            From ! {messenger, stop, user_exists_at_other_node},  %reject logon
            User_List;
        false ->
            From ! {messenger, logged_on},
            [{From, Name} | User_List]        %add user to the list
    end.

%%% Server deletes a user from the user list
server_logoff(From, User_List) ->
    lists:keydelete(From, 1, User_List).


%%% Server transfers a message between user
server_transfer(From, To, Message, User_List) ->
    %% check that the user is logged on and who he is
    case lists:keysearch(From, 1, User_List) of
```

```
        false ->
            From ! {messenger, stop, you_are_not_logged_on};
        {value, {From, Name}} ->
            server_transfer(From, Name, To, Message, User_List)
    end.
%%% If the user exists, send the message
server_transfer(From, Name, To, Message, User_List) ->
    %% Find the receiver and send the message
    case lists:keysearch(To, 2, User_List) of
        false ->
            From ! {messenger, receiver_not_found};
        {value, {ToPid, To}} ->
            ToPid ! {message_from, Name, Message},
            From ! {messenger, sent}
    end.


%%% User Commands
logon(Name) ->
    case whereis(mess_client) of
        undefined ->
            register(mess_client,
                     spawn(messenger, client, [server_node(), Name]));
        _ -> already_logged_on
    end.

logoff() ->
    mess_client ! logoff.

message(ToName, Message) ->
    case whereis(mess_client) of % Test if the client is running
        undefined ->
            not_logged_on;
        _ -> mess_client ! {message_to, ToName, Message},
             ok
end.


%%% The client process which runs on each server node
client(Server_Node, Name) ->
    {messenger, Server_Node} ! {self(), logon, Name},
    await_result(),
    client(Server_Node).

client(Server_Node) ->
    receive
        logoff ->
            {messenger, Server_Node} ! {self(), logoff},
            exit(normal);
        {message_to, ToName, Message} ->
            {messenger, Server_Node} ! {self(), message_to, ToName, Message},
            await_result();
        {message_from, FromName, Message} ->
```

```
            io:format("Message from ~p: ~p~n", [FromName, Message])
    end,
    client(Server_Node).

%%% wait for a response from the server
await_result() ->
    receive
        {messenger, stop, Why} -> % Stop the client
            io:format("~p~n", [Why]),
            exit(normal);
        {messenger, What} ->  % Normal response
            io:format("~p~n", [What])
    end.
```

To use this program you need to:

- configure the `server_node()` function
- copy the compiled code (`messenger.beam`) to the directory on each computer where you start Erlang.

In the following example of use of this program, I have started nodes on four different computers, but if you don't have that many machines available on your network, you could start up several nodes on the same machine.

We start up four Erlang nodes, messenger@super, c1@bilbo, c2@kosken, c3@gollum.

First we start up a the server at messenger@super:

```
(messenger@super)1> messenger:start_server().
true
```

Now Peter logs on at c1@bilbo:

```
(c1@bilbo)1> messenger:logon(peter).
true
logged_on
```

James logs on at c2@kosken:

```
(c2@kosken)1> messenger:logon(james).
true
logged_on
```

and Fred logs on at c3@gollum:

```
(c3@gollum)1> messenger:logon(fred).
true
logged_on
```

Now Peter sends Fred a message:

```
(c1@bilbo)2> messenger:message(fred, "hello").
ok
sent
```

And Fred receives the message and sends a message to Peter and logs off:

```
Message from peter: "hello"
(c3@gollum)2> messenger:message(peter, "go away, I'm busy").
ok
sent
(c3@gollum)3> messenger:logoff().
logoff
```

James now tries to send a message to Fred:

```
(c2@kosken)2> messenger:message(fred, "peter doesn't like you").
ok
receiver_not_found
```

But this fails as Fred has already logged off.

First let's look at some of the new concepts we have introduced.

There are two versions of the `server_transfer` function, one with four arguments (`server_transfer/4`) and one with five (`server_transfer/5`). These are regarded by Erlang as two separate functions.

Note how we write the `server` function so that it calls itself, `server(User_List)` and thus creates a loop. The Erlang compiler is "clever" and optimizes the code so that this really is a sort of loop and not a proper function call. But this only works if there is no code after the call, otherwise the compiler will expect the call to return and make a proper function call. This would result in the process getting bigger and bigger for every loop.

We use functions in the `lists` module. This is a very useful module and a study of the manual page is recommended (`erl -man lists`). `lists:keymember(Key,Position,Lists)` looks through a list of tuples and looks at `Position` in each tuple to see if it is the same as `Key`. The first element is position 1. If it finds a tuple where the element at `Position` is the same as Key, it returns `true`, otherwise `false`.

```
3> lists:keymember(a, 2, [{x,y,z},{b,b,b},{b,a,c},{q,r,s}]).
true
4> lists:keymember(p, 2, [{x,y,z},{b,b,b},{b,a,c},{q,r,s}]).
false
```

`lists:keydelete` works in the same way but deletes the first tuple found (if any) and returns the remaining list:

```
5> lists:keydelete(a, 2, [{x,y,z},{b,b,b},{b,a,c},{q,r,s}]).
[{x,y,z},{b,b,b},{q,r,s}]
```

`lists:keysearch` is like `lists:keymember`, but it returns `{value,Tuple_Found}` or the atom `false`.

There are a lot more very useful functions in the `lists` module.

An Erlang process will (conceptually) run until it does a `receive` and there is no message which it wants to receive in the message queue. I say "conceptually" because the Erlang system shares the CPU time between the active processes in the system.

A process terminates when there is nothing more for it to do, i.e. the last function it calls simply returns and doesn't call another function. Another way for a process to terminate is for it to call `exit/1`. The argument to `exit/1` has a special meaning which we will look at later. In this example we will do `exit(normal)` which has the same effect as a process running out of functions to call.

The BIF `whereis(RegisteredName)` checks if a registered process of name `RegisteredName` exists and return the pid of the process if it does exist or the atom `undefined` if it does not.

You should by now be able to understand most of the code above so I'll just go through one case: a message is sent from one user to another.

The first user "sends" the message in the example above by:

```
messenger:message(fred, "hello")
```

After testing that the client process exists:

```
whereis(mess_client)
```

and a message is sent to `mess_client`:

```
mess_client ! {message_to, fred, "hello"}
```

The client sends the message to the server by:

```
{messenger, messenger@super} ! {self(), message_to, fred, "hello"},
```

and waits for a reply from the server.

The server receives this message and calls:

```
server_transfer(From, fred, "hello", User_List),
```

which checks that the pid `From` is in the `User_List`:

```
lists:keysearch(From, 1, User_List)
```

If `keysearch` returns the atom `false`, some sort of error has occurred and the server sends back the message:

```
From ! {messenger, stop, you_are_not_logged_on}
```

which is received by the client which in turn does `exit(normal)` and terminates. If `keysearch` returns `{value,{From,Name}}` we know that the user is logged on and is his name (peter) is in variable `Name`. We now call:

```
server_transfer(From, peter, fred, "hello", User_List)
```

Note that as this is `server_transfer/5` it is not the same as the previous function `server_transfer/4`. We do another `keysearch` on `User_List` to find the pid of the client corresponding to fred:

```
lists:keysearch(fred, 2, User_List)
```

This time we use argument 2 which is the second element in the tuple. If this returns the atom `false` we know that fred is not logged on and we send the message:

```
From ! {messenger, receiver_not_found};
```

which is received by the client, if keysearch returns:

```
{value, {ToPid, fred}}
```

we send the message:

```
ToPid ! {message_from, peter, "hello"},
```

to fred's client and the message:

```
From ! {messenger, sent}
```

to peter's client.

Fred's client receives the message and prints it:

```
{message_from, peter, "hello"} ->
    io:format("Message from ~p: ~p~n", [peter, "hello"])
```

and peter's client receives the message in the await_result function.

1.4 Robustness

There are several things which are wrong with the messenger example [page 33] from the previous chapter. For example if a node where a user is logged on goes down without doing a log off, the user will remain in the server's User_List but the client will disappear thus making it impossible for the user to log on again as the server thinks the user already logged on.

Or what happens if the server goes down in the middle of sending a message leaving the sending client hanging for ever in the await_result function?

1.4.1 Timeouts

Before improving the messenger program we will look into some general principles, using the ping pong program as an example. Recall that when "ping" finishes, it tells "pong" that it has done so by sending the atom finished as a message to "pong" so that "pong" could also finish. Another way to let "pong" finish, is to make "pong" exit if it does not receive a message from ping within a certain time, this can be done by adding a *timeout* to pong as shown in the following example:

```
-module(tut19).

-export([start_ping/1, start_pong/0,  ping/2, pong/0]).

ping(0, Pong_Node) ->
    io:format("ping finished~n", []);

ping(N, Pong_Node) ->
    {pong, Pong_Node} ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
```

```
    end,
    ping(N - 1, Pong_Node).

pong() ->
    receive
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    after 5000 ->
            io:format("Pong timed out~n", [])
    end.

start_pong() ->
    register(pong, spawn(tut19, pong, [])).

start_ping(Pong_Node) ->
    spawn(tut19, ping, [3, Pong_Node]).
```

After we have compiled this and copied the `tut19.beam` file to the necessary directories:

On (pong@kosken):

```
(pong@kosken)1> tut19:start_pong().
true
Pong received ping
Pong received ping
Pong received ping
Pong timed out
```

On (ping@gollum):

```
(ping@gollum)1> tut19:start_ping(pong@kosken).
<0.36.0>
Ping received pong
Ping received pong
Ping received pong
ping finished
```

(The timeout is set in:

```
pong() ->
    receive
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    after 5000 ->
            io:format("Pong timed out~n", [])
    end.
```

We start the timeout (`after 5000`) when we enter `receive`. The timeout is canceled if {`ping,Ping_PID`} is received. If {`ping,Ping_PID`} is not received, the actions following the timeout will be done after 5000 milliseconds. `after` must be last in the `receive`, i.e. preceded by all other message reception specifications in the `receive`. Of course we could also call a function which returned an integer for the timeout:

```
after pong_timeout() ->
```

In general, there are better ways than using timeouts to supervise parts of a distributed Erlang system. Timeouts are usually appropriate to supervise external events, for example if you have expected a message from some external system within a specified time. For example, we could use a timeout to log a user out of the messenger system if they have not accessed it, for example, in ten minutes.

1.4.2 Error Handling

Before we go into details of the supervision and error handling in an Erlang system, we need see how Erlang processes terminate, or in Erlang terminology, *exit*.

A process which executes `exit(normal)` or simply runs out of things to do has a *normal* exit.

A process which encounters a runtime error (e.g. divide by zero, bad match, trying to call a function which doesn't exist etc) exits with an error, i.e. has an *abnormal* exit. A process which executes [exit(Reason)] where `Reason` is any Erlang term except the atom `normal`, also has an abnormal exit.

An Erlang process can set up links to other Erlang processes. If a process calls [link(Other_Pid)] it sets up a bidirectional link between itself and the process called `Other_Pid`. When a process terminates its sends something called a *signal* to all the processes it has links to.

The signal carries information about the pid it was sent from and the exit reason.

The default behaviour of a process which receives a normal exit is to ignore the signal.

The default behaviour in the two other cases (i.e. abnormal exit) above is to bypass all messages to the receiving process and to kill it and to propagate the same error signal to the killed process' links. In this way you can connect all processes in a transaction together using links and if one of the processes exits abnormally, all the processes in the transaction will be killed. As we often want to create a process and link to it at the same time, there is a special BIF, [spawn_link] which does the same as `spawn`, but also creates a link to the spawned process.

Now an example of the ping pong example using links to terminate "pong":

```
-module(tut20).

-export([start/1,  ping/2, pong/0]).

ping(N, Pong_Pid) ->
    link(Pong_Pid),
    ping1(N, Pong_Pid).

ping1(0, _) ->
    exit(ping);

ping1(N, Pong_Pid) ->
    Pong_Pid ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
```

```
    end,
    ping1(N - 1, Pong_Pid).

pong() ->
    receive
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start(Ping_Node) ->
    PongPID = spawn(tut20, pong, []),
    spawn(Ping_Node, tut20, ping, [3, PongPID]).

(s1@bill)3> tut20:start(s2@kosken).
Pong received ping
<3820.41.0>
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
```

This is a slight modification of the ping pong program where both processes are spawned from the same `start/1` function, where the "ping" process can be spawned on a separate node. Note the use of the `link` BIF. "Ping" calls `exit(ping)` when it finishes and this will cause an exit signal to be sent to "pong" which will also terminate.

It is possible to modify the default behaviour of a process so that it does not get killed when it receives abnormal exit signals, but all signals will be turned into normal messages on the format `{'EXIT',FromPID,Reason}` and added to the end of the receiving processes message queue. This behaviour is set by:

```
process_flag(trap_exit, true)
```

There are several other process flags, see [erlang(3)]. Changing the default behaviour of a process in this way is usually not done in standard user programs, but is left to the supervisory programs in OTP (but that's another tutorial). However we will modify the ping pong program to illustrate exit trapping.

```
-module(tut21).

-export([start/1,  ping/2, pong/0]).

ping(N, Pong_Pid) ->
    link(Pong_Pid),
    ping1(N, Pong_Pid).

ping1(0, _) ->
    exit(ping);

ping1(N, Pong_Pid) ->
    Pong_Pid ! {ping, self()},
```

```
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping1(N - 1, Pong_Pid).

pong() ->
    process_flag(trap_exit, true),
    pong1().

pong1() ->
    receive
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong1();
        {'EXIT', From, Reason} ->
            io:format("pong exiting, got ~p~n", [{'EXIT', From, Reason}])
    end.

start(Ping_Node) ->
    PongPID = spawn(tut21, pong, []),
    spawn(Ping_Node, tut21, ping, [3, PongPID]).
```

```
(s1@bill)1> tut21:start(s2@gollum).
<3820.39.0>
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
pong exiting, got {'EXIT',<3820.39.0>,ping}
```

1.4.3 The Larger Example with Robustness Added

Now we return to the messenger program and add changes which make it more robust:

```
%%% Message passing utility.
%%% User interface:
%%% login(Name)
%%%     One user at a time can log in from each Erlang node in the
%%%     system messenger: and choose a suitable Name. If the Name
%%%     is already logged in at another node or if someone else is
%%%     already logged in at the same node, login will be rejected
%%%     with a suitable error message.
%%% logoff()
%%%     Logs off anybody at at node
%%% message(ToName, Message)
%%%     sends Message to ToName. Error messages if the user of this
%%%     function is not logged on or if ToName is not logged on at
%%%     any node.
```

```
%%%
%%% One node in the network of Erlang nodes runs a server which maintains
%%% data about the logged on users. The server is registered as "messenger"
%%% Each node where there is a user logged on runs a client process registered
%%% as "mess_client"
%%%
%%% Protocol between the client processes and the server
%%% ----------------------------------------------------
%%%
%%% To server: {ClientPid, logon, UserName}
%%% Reply {messenger, stop, user_exists_at_other_node} stops the client
%%% Reply {messenger, logged_on} logon was successful
%%%
%%% When the client terminates for some reason
%%% To server: {'EXIT', ClientPid, Reason}
%%%
%%% To server: {ClientPid, message_to, ToName, Message} send a message
%%% Reply: {messenger, stop, you_are_not_logged_on} stops the client
%%% Reply: {messenger, receiver_not_found} no user with this name logged on
%%% Reply: {messenger, sent} Message has been sent (but no guarantee)
%%%
%%% To client: {message_from, Name, Message},
%%%
%%% Protocol between the "commands" and the client
%%% ----------------------------------------------
%%%
%%% Started: messenger:client(Server_Node, Name)
%%% To client: logoff
%%% To client: {message_to, ToName, Message}
%%%
%%% Configuration: change the server_node() function to return the
%%% name of the node where the messenger server runs

-module(messenger).
-export([start_server/0, server/0,
         logon/1, logoff/0, message/2, client/2]).

%%% Change the function below to return the name of the node where the
%%% messenger server runs
server_node() ->
    messenger@super.

%%% This is the server process for the "messenger"
%%% the user list has the format [{ClientPid1, Name1},{ClientPid22, Name2},...]
server() ->
    process_flag(trap_exit, true),
    server([]).

server(User_List) ->
    receive
        {From, logon, Name} ->
            New_User_List = server_logon(From, Name, User_List),
            server(New_User_List);
```

```
        {'EXIT', From, _} ->
            New_User_List = server_logoff(From, User_List),
            server(New_User_List);
        {From, message_to, To, Message} ->
            server_transfer(From, To, Message, User_List),
            io:format("list is now: ~p~n", [User_List]),
            server(User_List)
    end.

%%% Start the server
start_server() ->
    register(messenger, spawn(messenger, server, [])).

%%% Server adds a new user to the user list
server_logon(From, Name, User_List) ->
    %% check if logged on anywhere else
    case lists:keymember(Name, 2, User_List) of
        true ->
            From ! {messenger, stop, user_exists_at_other_node},  %reject logon
            User_List;
        false ->
            From ! {messenger, logged_on},
            link(From),
            [{From, Name} | User_List]        %add user to the list
    end.

%%% Server deletes a user from the user list
server_logoff(From, User_List) ->
    lists:keydelete(From, 1, User_List).


%%% Server transfers a message between user
server_transfer(From, To, Message, User_List) ->
    %% check that the user is logged on and who he is
    case lists:keysearch(From, 1, User_List) of
        false ->
            From ! {messenger, stop, you_are_not_logged_on};
        {value, {_, Name}} ->
            server_transfer(From, Name, To, Message, User_List)
    end.

%%% If the user exists, send the message
server_transfer(From, Name, To, Message, User_List) ->
    %% Find the receiver and send the message
    case lists:keysearch(To, 2, User_List) of
        false ->
            From ! {messenger, receiver_not_found};
        {value, {ToPid, To}} ->
            ToPid ! {message_from, Name, Message},
            From ! {messenger, sent}
    end.

%%% User Commands
```

```
logon(Name) ->
    case whereis(mess_client) of
        undefined ->
            register(mess_client,
                     spawn(messenger, client, [server_node(), Name]));
        _ -> already_logged_on
    end.

logoff() ->
    mess_client ! logoff.

message(ToName, Message) ->
    case whereis(mess_client) of % Test if the client is running
        undefined ->
            not_logged_on;
        _ -> mess_client ! {message_to, ToName, Message},
             ok
end.

%%% The client process which runs on each user node
client(Server_Node, Name) ->
    {messenger, Server_Node} ! {self(), logon, Name},
    await_result(),
    client(Server_Node).

client(Server_Node) ->
    receive
        logoff ->
            exit(normal);
        {message_to, ToName, Message} ->
            {messenger, Server_Node} ! {self(), message_to, ToName, Message},
            await_result();
        {message_from, FromName, Message} ->
            io:format("Message from ~p: ~p~n", [FromName, Message])
    end,
    client(Server_Node).

%%% wait for a response from the server
await_result() ->
    receive
        {messenger, stop, Why} -> % Stop the client
            io:format("~p~n", [Why]),
            exit(normal);
        {messenger, What} ->  % Normal response
            io:format("~p~n", [What])
    after 5000 ->
            io:format("No response from server~n", []),
            exit(timeout)
    end.
```

We have added the following changes:

The messenger server traps exits. If it receives an exit signal, {'EXIT',From,Reason} this means that a client process has terminated or is unreachable because:

- the user has logged off (we have removed the “logoff” message),
- the network connection to the client is broken,
- the node on which the client process resides has gone down, or
- the client processes has done some illegal operation.

If we receive an exit signal as above, we delete the tuple, {`From,Name`} from the servers `User_List` using the `server_logoff` function. If the node on which the server runs goes down, an exit signal (automatically generated by the system), will be sent to all of the client processes: {`'EXIT',MessengerPID,noconnection`} causing all the client processes to terminate.

We have also introduced a timeout of five seconds in the `await_result` function. I.e. if the server does not reply within five seconds (5000 ms), the client terminates. This is really only needed in the logon sequence before the client and server are linked.

An interesting case is if the client was to terminate before the server links to it. This is taken care of because linking to a non-existent process causes an exit signal, {`'EXIT',From,noproc`}, to be automatically generated as if the process terminated immediately after the link operation.

1.5 Records and Macros

Larger programs are usually written as a collection of files with a well defined interface between the various parts.

1.5.1 The Larger Example Divided into Several Files

To illustrate this, we will divide the messenger example from the previous chapter into five files.

`mess_config.hrl` header file for configuration data

`mess_interface.hrl` interface definitions between the client and the messenger

`user_interface.erl` functions for the user interface

`mess_client.erl` functions for the client side of the messenger

`mess_server.erl` functions for the server side of the messenger

While doing this we will also clean up the message passing interface between the shell, the client and the server and define it using *records*, we will also introduce *macros*.

```
%%%----FILE mess_config.hrl----

%%% Configure the location of the server node,
-define(server_node, messenger@super).

%%%----END FILE----

%%%----FILE mess_interface.hrl----

%%% Message interface between client and server and client shell for
%%% messenger program

%%%Messages from Client to server received in server/1 function.
-record(logon,{client_pid, username}).
-record(message,{client_pid, to_name, message}).
```

```
%%% {'EXIT', ClientPid, Reason}  (client terminated or unreachable.

%%% Messages from Server to Client, received in await_result/0 function
-record(abort_client,{message}).
%%% Messages are: user_exists_at_other_node,
%%%               you_are_not_logged_on
-record(server_reply,{message}).
%%% Messages are: logged_on
%%%               receiver_not_found
%%%               sent  (Message has been sent (no guarantee)
%%% Messages from Server to Client received in client/1 function
-record(message_from,{from_name, message}).

%%% Messages from shell to Client received in client/1 function
%%% spawn(mess_client, client, [server_node(), Name])
-record(message_to,{to_name, message}).
%%% logoff

%%%----END FILE----

%%%----FILE user_interface.erl----

%%% User interface to the messenger program
%%% login(Name)
%%%     One user at a time can log in from each Erlang node in the
%%%     system messenger: and choose a suitable Name. If the Name
%%%     is already logged in at another node or if someone else is
%%%     already logged in at the same node, login will be rejected
%%%     with a suitable error message.

%%% logoff()
%%%     Logs off anybody at at node

%%% message(ToName, Message)
%%%     sends Message to ToName. Error messages if the user of this
%%%     function is not logged on or if ToName is not logged on at
%%%     any node.

-module(user_interface).
-export([logon/1, logoff/0, message/2]).
-include("mess_interface.hrl").
-include("mess_config.hrl").

logon(Name) ->
    case whereis(mess_client) of
        undefined ->
            register(mess_client,
                     spawn(mess_client, client, [?server_node, Name]));
        _ -> already_logged_on
    end.

logoff() ->
    mess_client ! logoff.
```

```
message(ToName, Message) ->
    case whereis(mess_client) of % Test if the client is running
        undefined ->
            not_logged_on;
        _ -> mess_client ! #message_to{to_name=ToName, message=Message},
             ok
end.

%%%----END FILE----

%%%----FILE mess_client.erl----

%%% The client process which runs on each user node

-module(mess_client).
-export([client/2]).
-include("mess_interface.hrl").

client(Server_Node, Name) ->
    {messenger, Server_Node} ! #logon{client_pid=self(), username=Name},
    await_result(),
    client(Server_Node).

client(Server_Node) ->
    receive
        logoff ->
            exit(normal);
        #message_to{to_name=ToName, message=Message} ->
            {messenger, Server_Node} !
                #message{client_pid=self(), to_name=ToName, message=Message},
            await_result();
        {message_from, FromName, Message} ->
            io:format("Message from ~p: ~p~n", [FromName, Message])
    end,
    client(Server_Node).

%%% wait for a response from the server
await_result() ->
    receive
        #abort_client{message=Why} ->
            io:format("~p~n", [Why]),
            exit(normal);
        #server_reply{message=What} ->
            io:format("~p~n", [What])
    after 5000 ->
            io:format("No response from server~n", []),
            exit(timeout)
    end.

%%%----END FILE---

%%%----FILE mess_server.erl----
```

```
%%% This is the server process of the messenger service

-module(mess_server).
-export([start_server/0, server/0]).
-include("mess_interface.hrl").

server() ->
    process_flag(trap_exit, true),
    server([]).

%%% the user list has the format [{ClientPid1, Name1},{ClientPid22, Name2},...]
server(User_List) ->
    io:format("User list = ~p~n", [User_List]),
    receive
        #logon{client_pid=From, username=Name} ->
            New_User_List = server_logon(From, Name, User_List),
            server(New_User_List);
        {'EXIT', From, _} ->
            New_User_List = server_logoff(From, User_List),
            server(New_User_List);
        #message{client_pid=From, to_name=To, message=Message} ->
            server_transfer(From, To, Message, User_List),
            server(User_List)
    end.

%%% Start the server
start_server() ->
    register(messenger, spawn(?MODULE, server, [])).

%%% Server adds a new user to the user list
server_logon(From, Name, User_List) ->
    %% check if logged on anywhere else
    case lists:keymember(Name, 2, User_List) of
        true ->
            From ! #abort_client{message=user_exists_at_other_node},
            User_List;
        false ->
            From ! #server_reply{message=logged_on},
            link(From),
            [{From, Name} | User_List]        %add user to the list
    end.

%%% Server deletes a user from the user list
server_logoff(From, User_List) ->
    lists:keydelete(From, 1, User_List).

%%% Server transfers a message between user
server_transfer(From, To, Message, User_List) ->
    %% check that the user is logged on and who he is
    case lists:keysearch(From, 1, User_List) of
        false ->
            From ! #abort_client{message=you_are_not_logged_on};
```

```
        {value, {_, Name}} ->
            server_transfer(From, Name, To, Message, User_List)
    end.
%%% If the user exists, send the message
server_transfer(From, Name, To, Message, User_List) ->
    %% Find the receiver and send the message
    case lists:keysearch(To, 2, User_List) of
        false ->
            From ! #server_reply{message=receiver_not_found};
        {value, {ToPid, To}} ->
            ToPid ! #message_from{from_name=Name, message=Message},
            From !  #server_reply{message=sent}
    end.

%%%----END FILE---
```

1.5.2 Header Files

You will see some files above with extension `.hrl`. These are header files which are included in the `.erl` files by:

```
-include("File_Name").
```

for example:

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

In our case above the file is fetched from the same directory as all the other files in the messenger example. (*manual*).

.hrl files can contain any valid Erlang code but are most often used for record and macro definitions.

1.5.3 Records

A record is defined as:

```
-record(name_of_record,{field_name1, field_name2, field_name3, ......}).
```

For example:

```
-record(message_to,{to_name, message}).
```

This is exactly equivalent to:

```
{message_to, To_Name, Message}
```

Creating record, is best illustrated by an example:

```
#message_to{message="hello", to_name=fred)
```

This will create:

```
{message_to, fred, "hello"}
```

Note that you don't have to worry about the order you assign values to the various parts of the records when you create it. The advantage of using records is that by placing their definitions in header files you can conveniently define interfaces which are easy to change. For example, if you want to add a new field to the record, you will only have to change the code where the new field is used and not at every place the record is referred to. If you leave out a field when creating a record, it will get the value of the atom undefined. (*manual*)

Pattern matching with records is very similar to creating records. For example inside a `case` or `receive`:

```
#message_to{to_name=ToName, message=Message} ->
```

is the same as:

```
{message_to, ToName, Message}
```

1.5.4 Macros

The other thing we have added to the messenger is a macro. The file `mess_config.hrl` contains the definition:

```
%%% Configure the location of the server node,
-define(server_node, messenger@super).
```

We include this file in mess_server.erl:

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

Every occurrence of `?server_node` in `mess_server.erl` will now be replaced by `messenger@super`.

The other place a macro is used is when we spawn the server process:

```
spawn(?MODULE, server, [])
```

This is a standard macro (i.e. defined by the system, not the user). `?MODULE` is always replaced by the name of current module (i.e. the `-module` definition near the start of the file). There are more advanced ways of using macros with, for example parameters (*manual*).

The three Erlang (`.erl`) files in the messenger example are individually compiled into object code file (`.beam`). The Erlang system loads and links these files into the system when they are referred to during execution of the code. In our case we simply have put them in the same directory which is our current working directory (i.e. the place we have done "cd" to). There are ways of putting the `.beam` files in other directories.

In the messenger example, no assumptions have been made about what the message being sent is. It could be any valid Erlang term.