Erlang in Real Time

Maurice Castro

This is the second draft of this book and has been used in the course 'CS584 Real Time and Concurrent Systems' at RMIT University.

Erlang in Real Time
Maurice Castro

Department of Computer Science
RMIT
124 Latrobe St
Melbourne VIC
Australia

ISBN:

Maurice Castro
Department of Computer Science
RMIT
GPO Box 2476V
Melbourne VIC 3001
Australia

maurice@serc.rmit.edu.au

Preface

Why Erlang?

In 1984 Ericsson conducted a series of experiments using a range of programming languages and programming techniques to identify the qualities required in a software development environment for telecommunications applications. The experiments included imperative, functional and logic languages, and rule based and object oriented techniques. At the conclusion of the experiments there was no existing language with all the required features, however, the functional programming languages showed promise as they resembled existing imperative programs and allowed functions to be easily combined while preserving correctness.

The Erlang language is designed to meet the requirements of a telephony environment. On the technical side telephony software must meet soft real-time timing constraints, support concurrent programming, and provide a means for replacing running code. When making a telephone call clients expect results within a short period of time. To ensure that service is delivered within client expectations, it is necessary for the programmer to be able to specify actions that occur within a given time and to program reactions if these actions do not occur. In addition, telephone exchanges have an inherent parallelism in that many similar actions must be handled at the same time, typically the making and receiving of telephone calls. The software used in telephone exchanges runs for long periods of time, and in general an exchange cannot be shut down to allow for software changes. This characteristic makes it necessary to mechanism that allow software to be updated on the fly. On the management side, the programming language must support large teams of programmers, require minimal effort in the systems integration phase of development, and be easily and quickly taught. The development and maintenance of telecommunications products are large scale endeavours of great complexity. To meet the challenge of assembling these products in a timely fashion large teams are necessary, furthermore, reducing the cost of putting together the parts made by members of the teams eliminates a significant development cost. Finally, if a special purpose language is used, short training time allows the productivity benefits of the language to be gained at a minimal cost and allows large numbers of programmers capable in the language to be procured in a reasonable time.

Erlang is a small conceptually simple language that meets these requirements.

Languages

In addition to Erlang this book will use several programming languages including C, Ada and Java to illustrate concepts in a conventional programming environment. A subsidiary aim of this book is to encourage the reader to carry concepts developed in Erlang into other programming languages. Although these languages may not enforce the discipline of Erlang, the restrictions imposed by Erlang are often helpful in reducing coding and debugging when carried out in other languages.

Approach

The focus of this work is on the transition between programming in procedural languages and programming in a declarative functional language. This leads to an unusual approach to introducing the Erlang language. Early examples tend to have a greater number of **if** statements and use pattern matching in the heads of functions less than the best of Erlang programmers would. This style is a compromise between the imperative programming approach and the declarative approach and is used while introducing the building blocks of the language. *Hopefully* this approach should make the early chapters less intimidating to programmers making the transition from an imperative languages to Erlang.

Getting Erlang

Ericsson has released a number of implementations of Erlang under a free of charge license. Further information about these implementations can be found at:

`http://www.erlang.se/erlang/sure/main/download/`

Other Resources

Further information on the language can be found in the original Erlang book 'Concurrent Programming in Erlang' by Joe Armstrong, Robert Virding, Claes Wikström and Mike Williams and published by Prentice-Hall. The first part of the book has been made available online at:

`http://www.erlang.se/erlang/sure/main/news/erlang-book-part1.ps`
`http://www.erlang.se/erlang/sure/main/news/erlang-book-part1.pdf`

The book itself is available in print:

Joe Armstrong, Robert Virding, Claes Wikström and Mike Williams
'Concurrent Programming in Erlang'
Second Edition
Prentice Hall
Englewood Cliffs, New Jersey
ISBN: 0-13-508301-X

Acknowledgements

The author was fortunate to have the assistance of Torbjörn Törnkvist from Ericsson's CS labs to clarify aspects of the Erlang programming language and its use.

The author also wishes to thank Doug Bagley for his proof reading of the document.

Thanks also to Richard O'Keefe for his suggestions on writing truly functional solutions.

Contents

Chapter 1

Introduction to Erlang

This is a 'Quick Start' chapter. It does not cover Erlang in detail, instead it focuses on getting the user going in the language as quickly as possible so that the user can try out the material in subsequent chapters.

1.1 Programming Environment

Like Java, the most generally available implementation of Erlang is interpreted. Erlang code is compiled into a byte code which is interpreted. This allows the code to be easily transported between systems. Unlike Java, Erlang provides an additional environment which allows the programmer to directly interact with the functions in their code. This environment, known as the Erlang Shell, is described in this section. The ability to directly interact with functions is a powerful and very useful feature of Erlang.

1.1.1 Starting and Leaving the Erlang Shell

The Erlang shell is invoked from the command line using the **erl** command (see figure 1.1). The shell can be exited by typing **control-g** and then **q**.

```
% erl
Erlang (JAM) emulator version 4.3.1

Eshell V4.3.1  (abort with ^G)
1>
```

Figure 1.1: Starting the Erlang Shell

1.1.2 Functions Provided by the Shell

The shell interprets user input as fragments of Erlang code. The shell allows variables to be assigned and functions to be called just as if the actions took place inside a compiled Erlang program. The only differences between the shell and a program are that the shell does not allow the user to define their own functions and allows bindings to be removed.

The shell provides some on line help. It can be accessed by typing

```
help().
```

at the shell prompt. Shell internal commands can be accessed just by typing the function name with its arguments in brackets. Commands in modules are accessed by prefacing the function name with the module name and a colon. The compiler function *c* is in the *c* module and its use is shown in section 1.1.3.

1.1.3 Compiling and Running Programs

The first example is a *Tic-Tac-Toe* game. The program file is called *ttt.erl*. To compile the program, invoke Erlang and issue the command **c:c(ttt).** and to run it use the command **ttt:init().** Figure 1.2 shows the compilation and figure 1.3 shows a game in progress.

```
% erl
Erlang (JAM) emulator version 4.5.3

Eshell V4.5.3  (abort with ^G)
1> c:c(ttt).
{ok,ttt}
2> ttt:init().
```

Figure 1.2: Compiling and Running *Tic-Tac-Toe*

1.1.4 Starting Additional Shells

The run time environment which supports the shell can be invoked by pressing **control-g**. The environment provides a set of commands which allow the user to create, kill, and connect to many processes. Figure 1.4 illustrates accessing the environment, its help information, and creating and switching between shells.

1.2 Anatomy of An Erlang Program

An Erlang program consists of a set of functions which may be collected into modules. A short program that counts the number of lines and characters in a file will be used as an example. The code for the *filecnt* program is shown in figure 1.5. Some sample output is shown in figure 1.6.

Some interesting features of the program:

- An Erlang module is a device for collecting together functions. Modules are also the unit of compilation. Thus a file which is to be compiled must contain a **module** declaration (line 1 of figure 1.5.

- The visibility of functions are controlled through the **export** declaration (line 2). The only functions in the module that can be seen by code outside the module are those listed in the export declaration. It is important to note that functions have an arity equivalent to the number of arguments of the function. The function name and the number of arguments taken by the function uniquely identify the function.

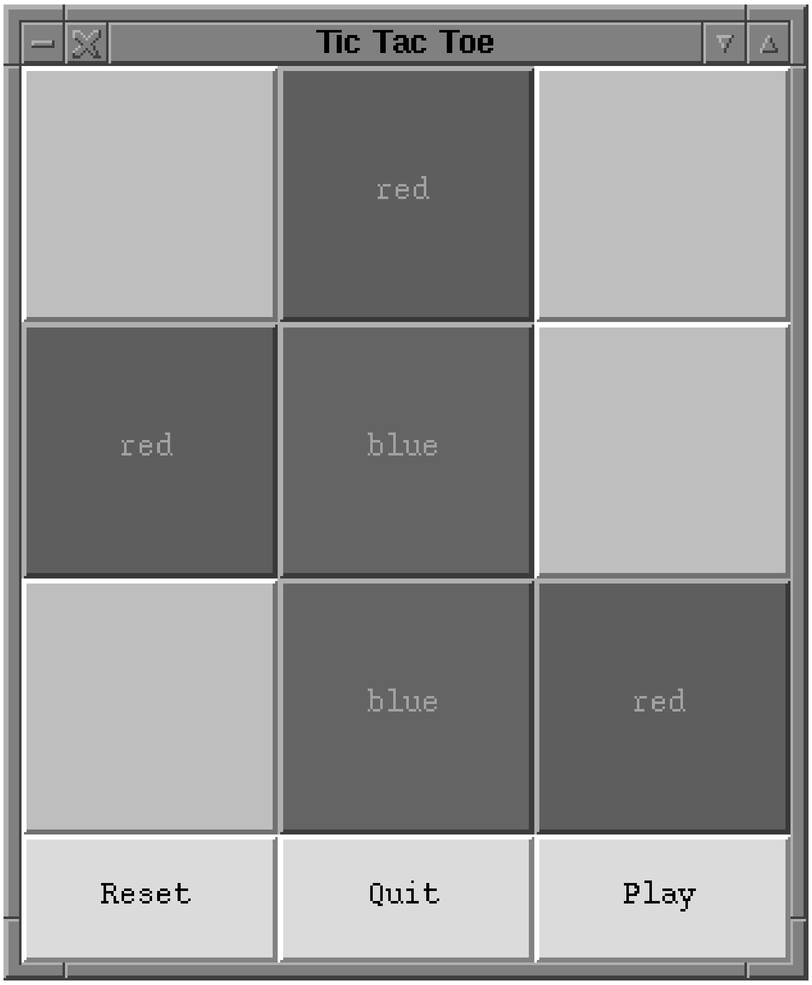


Figure 1.3: *Tic-Tac-Toe* in Action

```
% erl
Erlang (JAM) emulator version 4.5.3

Eshell V4.5.3  (abort with ^G)
1> h().
ok
2>^G
User switch command
 --> h
  c [nn]   - connect to job
  i [nn]   - interrupt job
  k [nn]   - kill job
  j        - list all jobs
  s        - start local shell
  r [node] - start remote shell
  q        - quit erlang
  ? | h    - this message
 --> s
 --> j
   1  {}
   2  {shell,start,[]}
   3* {shell,start,[]}
 --> c
Eshell V4.5.3  (abort with ^G)
1>^G
User switch command
 --> c 2

2>^G
User switch command
 --> c 3

1>
```

Figure 1.4: Accessing the Environment

```
-module(filecnt).
-export([filecnt/1]).

% Accept a filename and attempt to open the file

filecnt(FileName) ->
    {Status, Data} = file:open(FileName, [read]),
    if
        Status == error ->
            io:format("Unable to open file ~w because ~w~n",
                 [FileName, Data]);
        true ->
            fc(FileName, Data, {0,0})
    end.

% count characters and lines, return tuple

fc(FN, Fd, {Chars, Lines}) ->
    C = file:read(Fd, 1),
    if
        C == eof ->
            {Chars, Lines};
        true ->
            {Result, Data} = C,
            if
                Result == error ->
                    io:format("Unable to read file ~w because ~w~n",
                        [FN, Data]);
                true ->
                    if
                        Data == "\n" ->
                            fc(FN, Fd, {Chars+1, Lines+1});
                        true ->
                            fc(FN, Fd, {Chars+1, Lines})
                    end
            end
    end.
```

Figure 1.5: *Filecnt* Source Code

```
Erlang (JAM) emulator version 4.5.3

Eshell V4.5.3  (abort with ^G)
1> c:c(filecnt).
{ok,filecnt}
2> filecnt:filecnt('filecnt.erl').
{1006,37}
3>
```

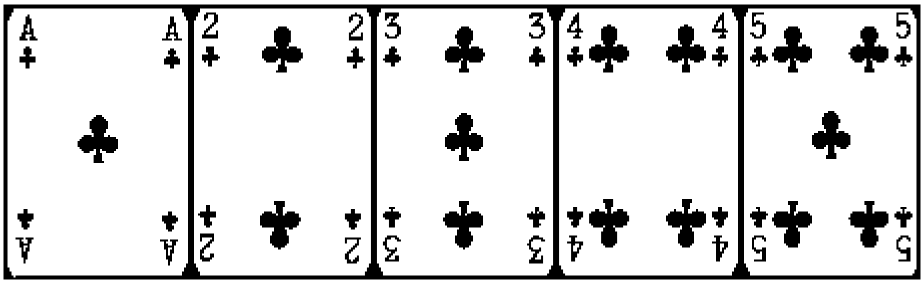
Figure 1.6: Sample output from *filecnt*

- Comments are preceded with a percent sign (lines 4 and 16).
- Two functions are defined: *filecnt* (lines 6–14) and *fc* (lines 18–37).
- Variables are assigned to exactly once and start with capital letters. (lines 7, 19, and 24)
- Recursion is used to perform any repeating actions (lines 32 and 34).
- Functions in other modules are accessed by prefixing the name of the module and a colon to the name of the function (lines 7, 10, 19, and 28).

1.3 Factorial: The Classic Recursion Example

This section provides the classical explanation of the factorial function. It is included to provide a link back to the mathematical basis of recursion. The factorial function is one of simplest the most used examples of a recursive function. Card games and games of chance were significant driving force in the development of probability. Questions similar to the following are not uncommon:

> If you were presented with five playing cards labeled 'A' to '5' in how many ways can you arrange those cards?



The classic answer is: you have 5 choices for the first card, then 4 choices for the second card and so on until you have 1 choice for the final card. Giving $5 \times 4 \times 3 \times 2 \times 1 = 120$ choices.

This answer can be generalised to any number of starting cards and the generalisation is known as the factorial function. It can be written as a recurrence relationship:

$$x! = \begin{cases} 1 & \text{if } x < 1 \\ x * (x-1)! & \text{otherwise} \end{cases}$$

The Erlang function *fact* in figure 1.7 implements the recurrence relationship.

1.4 Anatomy of a Function

The factorial function in figure 1.7 illustrates a number of interesting aspects of the Erlang programming language:

- Functions are composed of function heads (lines 4 and 6) and function bodies (lines 5 and 7)

```
1   -module(factprg).
2   -export([fact/1]).
3
4   fact(0) ->
5           1;
6   fact(N) ->
7           N * fact(N-1).
```

Figure 1.7: The factorial function: *fact*

- Functions are composed of clauses (lines 4–5 and lines 6–7). A clause is composed of a function head and a function body. The clauses of a function are separated by semicolons (line 5). The final clause of a function ends in a full-stop (line 7).
- When a function executes each of the function heads is tested in turn. The first function head which matches the argument the function was invoked with has its body executed. If no function head matches an error occurs. In the example line 6 is only executed when *fact* is called with an argument of *0*.
- The arguments in the function head are known as a pattern and the process of selecting the function head is known as pattern matching.

1.5 Resources

The code files mentioned in this chapter are:

ttt.erl
filecnt.erl
factprg.erl

These files can be retrieved from:

`http://www.serc.rmit.edu.au/~maurice/erlbk/eg/erlintro`.

1.6 Exercises

1. Extend the *filecnt* program to count full-stops in files.
2. The algorithm used in the *Tic-Tac-Toe* game for automatic play has no insight into the game. Rewrite the *play* function to play better. *Hint:* use Erlang's pattern matching facilities to match grid positions and hard-code the rules for winning play.

Chapter 2

Fundamentals

This chapter describes the fundamental parts of the Erlang programming language including: basic and compound data types, variables, functions, guards, and modules.

2.1 Data Types

Nouns, verbs, and adjectives in languages like English and Swedish are collections of words which perform particular roles in the language. These roles restrict the words to be used in a particular context and in a particular way. The types and subtypes of words and arrangements of words in language are sometimes called the 'parts of the language'. Programming languages also have parts. In particular, the data a program works on is divided into a number of different types. Normally there are constants associated with this data.

Erlang supports 5 simple data types:

- Integer - a positive or negative number with no fractional part (ie. no decimal point)
- Float - a number with a fractional part (ie. no decimal point)
- Atom - a constant name
- Pid - a process identifier
- Reference - a unique value that can be copied or passed but cannot be generated again

Two compound data types are supported in Erlang

- Tuple - a fixed length collection of elements
- List - a variable length collection of element

A term is a value made from any of the above data types

2.1.1 Integers

The Erlang programming language requires that integers have at least 24 bits precision. This means that any number between $2^{24}-1$ and $-2^{24}-1$ must be representable as an integer. There are several ways of writing integer constants some examples are illustrated below:

$$
\begin{array}{l}
16777215 \\
-16777215 \\
\$\mathrm{A} \\
2\#101 \\
16\#1A \\
0
\end{array}
$$

In order, the examples are: the largest integer guaranteed to be present in Erlang (some implementations may offer larger values); the smallest integer guaranteed to be present in Erlang (some implementations may offer smaller values); the integer corresponding to the character constant 'A' (integer value 65); the integer corresponding to '101_2' (integer value 5); the integer corresponding to '$1A_{16}$' (integer value 26); and 0.

The examples introduced 2 Erlang specific notations. The '$' and the '#'.

The '$' returns the position of the character following it in the ASCII character set:

$**char**

The '#' allows integers in the bases 2 . . . 16 to be specified using the notation:

base#value

2.1.2 Floats

Erlang uses the conventional notation for floating point numbers. Some examples are:

$$
\begin{array}{l}
16.0 \\
-16.22 \\
-1.8e2 \\
-0.36e-2 \\
1.0e3 \\
1.0e6
\end{array}
$$

In order the examples are: 16.0; -16.22; -180.0; -3.6×10^{-3}; 1000.0; and 1.0×10^6.

2.1.3 Numbers

Floats and integers can be combined into arithmetic expressions. Table 2.1 is a table of operators used for arithmetic.

Op	Description
$+X$	$+X$
$-X$	$-X$
$X * Y$	$X * Y$
X/Y	X/Y (floating point division)
X div Y	X/Y (integer division)
X rem Y	integer remainder of X / Y
X band Y	bitwise and of X and Y
$X + Y$	$X + Y$
$X - Y$	$X - Y$
X bor Y	bitwise or of X and Y
X bxor Y	bitwise xor of X and Y
X bsl Y	arithmetic shift left of X by Y bits
X bsr Y	shift right of X by Y bits

Table 2.1: Arithmetic Operations

2.1.4 Atoms

An atom is a constant name. The value of an atom is its name. Two atoms are equivalent when they are spelt identically. Atom constants either begin with a lower case letter and are delimited by white space; or an atom is quoted in single quotes (' ').

The following are atoms:

start
begin_here
'This is an atom'
'Fred'

Atoms defined using single quotes may include non-printing and special characters. Table 2.2 contains sequences that can be included in a quoted atom to represent special characters.

Some examples of quoted atoms containing special characters are:

'hello, world\n'
'first line\nsecond line\n'
'1\t2'

Long quoted atoms can be split across lines by ending the line with a backslash character ('\').

```
'this is a long atom \
continued on the next line'
```

2.1.5 Pids

The programming environment that supports Erlang is designed to run many Erlang programs in parallel. Each program operates independently to other programs: parameters and memory are not shared between Erlang programs.

Char	Meaning
\b	Backspace
\d	Delete
\e	Escape
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Tab
\v	Vertical tab
\\	Backslash
\^A . . . \^Z	Control A (0) to Control Z (26)
\’	Quote
\”	Double Quote
\OOO	Character with octal value OOO

Table 2.2: Quoted Atom Conventions

This makes each thread of execution in the Erlang programming environment a process. A **Process Identifier** (Pid) is a unique name assigned to a process. Pids are used for communicating between processes and to identify processes to the programming environment for operations which affect the operation of a process – for example creating, destroying and changing the scheduling priority of processes.

2.1.6 References

The Erlang run time environment provides a function which returns a value which is unique within all running Erlang environments. This value is called a reference. Note that although no two references can be generated which are identical among running systems, an Erlang environment which has been failed and is then restarted can produce references which were previously produced by the failed environment.

Unique values can prove useful as tokens in concurrent systems.

2.1.7 Tuples

Tuples are data structures which are used to store a fixed number of items. They are written as a a group of comma separated terms, surrounded by the curly brackets.

Examples of tuples are:

{1,2,3}
{fred,20.0,3}
{15,’fifteen’}
{3,{a,b,c}}
{3,[a,b,c]}

The items that compose a tuple are called elements. The elements are identified by their position in the tuple and may be extracted using pattern matching.

An example is shown in figure 2.1.

```
% erl
Erlang (JAM) emulator version 4.5.3

Eshell V4.5.3  (abort with ^G)
1> T = {1,2,3}.
{1,2,3}
2> {A,B,C} = T.
{1,2,3}
3> A.
1
4> B.
2
5> C.
3
6>
```

Figure 2.1: Manipulating a Tuple in the Erlang Shell

The size of a tuple is equivalent to the number of elements in the tuple.

2.1.8 Lists

The list data structure does not have a predetermined size. The Erlang programming language defines a number of operators and functions that allow new lists to be created from an existing list which either have more elements or fewer elements than the original list.

Lists are written as a a group of comma separated terms, surrounded by the square brackets.

Examples of lists are:

[1,2,3]
[fred,20.0,3]
[15,'fifteen']
[3,[a,b,c]]
[{a,b}, {a,b,c}]
[]

Erlang has a special notation for generating lists of characters easily. A string of characters enclosed in double quotation marks is converted to a list of integers representing the characters. The conventions used for quoted atoms figure 2.2 also apply. An example of this special notation is shown in figure 2.2.

The vertical separator (|) is used in the list notation to separate the specified head part of a list from the remainder of the list. Some examples of the use of this notation are shown in figure 2.3.

The majority of Erlang functions are written to manipulate and return **proper** or **well formed lists**. Proper or well formed lists have an empty list ([]) as their last element.

Some useful functions that operate on lists are found in table 2.3.

```
% erl
Erlang (JAM) emulator version 4.5.3

Eshell V4.5.3  (abort with ^G)
1> A = "hello".
"hello"
2> B = [ a | A].
[a,104,101,108,108,111]
3> [ X | Y ] = B.
[a,104,101,108,108,111]
4> X.
a
5> Y.
"hello"
6> C = [ $a | A ].
"ahello"
7> hd(C).
97
8> $a.
97
9>
```

Figure 2.2: Manipulating a String / List of Characters in the Erlang Shell

```
% erl
Erlang (JAM) emulator version 4.5.3

Eshell V4.5.3  (abort with ^G)
1> Z = [a,b,c,d,e,f].
[a,b,c,d,e,f]
2> [A, B | R] = Z.
[a,b,c,d,e,f]
3> A.
a
4> B.
b
5> R.
[c,d,e,f]
6> X = [A, B | R].
[a,b,c,d,e,f]
7> X.
[a,b,c,d,e,f]
8>
```

Figure 2.3: Manipulating a List Using | in the Erlang Shell

Func	Description
atom_to_list(X)	return the list of ASCII characters that form the atom X
float_to_list(X)	return the list of ASCII characters that represent the value of the float X
integer_to_list(X)	return the list of ASCII characters that represent the value of the integer X
list_to_atom(X)	return an atom composed of the characters formed from the list of ASCII characters X
list_to_float(X)	return a float composed of the characters formed from the list of ASCII characters X
list_to_integer(X)	return an integer composed of the characters formed from the list of ASCII characters X
hd(L)	The head – first element – of the list L
tl(L)	The tail – last element – of the list L
length(L)	The number of elements in the list L

Table 2.3: Selected List Functions

2.2 Variables

Erlang's variables behave differently to variables in procedural languages like C, Ada and Java. Erlang's variables have the following properties:

- The scope (region of the program in which a variable can be accessed) of a variable extends from its first appearance in a clause through to the end of the clause in an Erlang function.
- The contents of an Erlang variable persist from assignment until the end of the clause.
- Erlang variables may be assigned to (bound) exactly once.
- It is an error to access an unbound Erlang variable.
- Erlang variables are not typed. Any term can be bound to a variable.

The property of allowing a variable to bound exactly once is known as **single assignment**.

One way of calculating elementary functions such as exp (e^x) and sine is to use a Taylor Series. The code in figure 2.4 illustrates how these functions may be implemented. This code can also be used to show the scope of variables.

The function *sin* which takes 4 arguments is composed of two clauses (lines 30 – 37) and (lines 38 – 45). Each of these clauses has the variables Z, N, *Epsilon*, and R. Although the variables have the same names in the two functions, the variables are in fact different. Furthermore, the contents of these variables can only be accessed within the clause where they have been defined.

```
-module(tayser).
-export([exp/1, sin/1]).

epsilon() ->
        1.0e-8.

fact(0) ->
        1;
fact(N) ->
        N * fact(N-1).

taylorterm(Z, N) ->
        math:pow(Z, N) / fact(N).

exp(Z) ->
        exp(Z, 0, epsilon()).

exp(Z, N, Epsilon) ->
        R = taylorterm(Z, N),
        if
                R < Epsilon ->
                        0;
                true ->
                        R + exp(Z, N+1, Epsilon)
        end.

sin(Z) ->
        sin(Z, 0, 1, epsilon()).

sin(Z, N, 1, Epsilon) ->
        R = taylorterm(Z, (2*N)+1),
        if
                R < Epsilon ->
                        0;
                true ->
                        R + sin(Z, N+1, -1, Epsilon)
        end;
sin(Z, N, -1, Epsilon) ->
        R = taylorterm(Z, (2*N)+1),
        if
                R < Epsilon ->
                        0;
                true ->
                        -R + sin(Z, N+1, 1, Epsilon)
        end.
```

Figure 2.4: *Tayser* Source Code

2.3 Memory Management

Languages like C and C++ support functions that explicitly allocate and deallocate memory (C provides many functions including **alloca**, **malloc**, **calloc** and **free**; C++ provides **new** and **delete**). Erlang does not have explicit memory management, freeing the programmer from having to allocate and deallocate memory. Instead the language creates space to hold the contents of a variable when necessary and automatically frees the allocated space when it can no longer be referenced. The process of freeing the allocated memory is sometimes called garbage collection.

2.4 Functions

In mathematics a function describes a relationship between its inputs and its outputs. The key characteristic of this relationship is that if the same combination of inputs is supplied then the same output is produced each and every time the function is used. This *functional relationship* is often illustrated using a diagram similar to figure 2.5. Functions are sometimes said to have a many to 1 relationship.

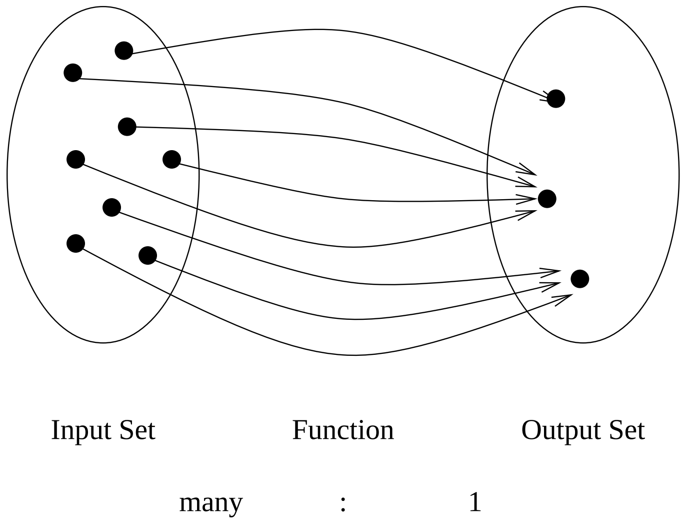


Figure 2.5: A function

The property of always getting the same result regardless of when a function is used is highly desirable. This means that a function that has been tested in one environment can be used in any other environment without worrying about the environment affecting the function. Of course the new environment may provide inputs to the function that were not present in the test environment, and hence discover a flaw in the function. However, after fixing the problem, the fixed function should be operable in both environments. In general, Erlang functions do not interact with their environment, except through their input parameters, and hence exhibit this desirable property.

```
-module(factprg2).
-export([fact/1]).

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

Figure 2.6: The factorial function: *fact*

A function which affects or interacts with its environment is said to have side effects. Very few functions in Erlang have side effects. The parts of Erlang which exhibit side effects include: Input / Output operations, process dictionaries, and message passing. These classes of operation will be discussed later.

As noted earlier (see section 1.4), Erlang functions are composed of clauses which are selected for execution by pattern matching the head of clause. Once a clause is selected it is executed until it returns a value. This value is returned to the calling function. The clauses of a function are separated by semicolons and the last clause of a function ends in a full-stop.

2.5 Guards

In addition to matching patterns, the head of a clause can be augmented with a guard clause. Figure 2.6 shows the factorial function (figure 1.7) rewritten to use a guard clause. Guard clauses are also used in conditional expressions and message reception.

Some functions are allowed to be used in guards (see table 2.4). Table 2.5 shows a table of operations permitted in guards. Guard clauses can be combined using a logical-and operator by separating the clauses with commas.

Guard	Test
atom(X)	X is an atom
constant(X)	X is not a list or tuple
float(X)	X is a float
integer(X)	X is an integer
list(X)	X is a list
number(X)	X is an integer or a float
pid(X)	X is a process identifier
port(X)	X is a port
reference (X)	X is a reference
tuple (X)	X is a tuple

The following functions are also permitted: element/2, float/1, hd/1, length/1, round/1, self/1, size/1, trunc/1, tl/1

Table 2.4: Guard Tests

The *ot* program demonstrates the aspects of guards discussed. The source code is shown in figure 2.7 and a sample run is shown in figure 2.8. The *ot*

OP	Description
$X > Y$	X greater than Y
$X < Y$	X less than Y
$X =< Y$	X less than or equal to Y
$X >= Y$	X greater than or equal to Y
$X == Y$	X equal to Y
$X/ = Y$	X not equal to Y
$X =:= Y$	X exactly equal to Y (no type conv)
$X = / = Y$	X not exactly equal to Y (no type conv)

Table 2.5: Guard Operations

program determines the type of its argument and reports it.

```
-module(ot).
-export([ot/1]).

ot(X) when integer(X), X > 0 ->
        'positive natural';
ot(X) when integer(X), X < 0 ->
        'negative integer';
ot(X) when integer(X) ->
        integer;
ot(X) when float(X) ->
        float;
ot(X) when list(X) ->
        list;
ot(X) when tuple(X) ->
        tuple;
ot(X) when atom(X) ->
        atom.
```

Figure 2.7: *ot.erl*

2.6 Modules

Modules are a mechanism for collecting functions together in Erlang. No storage is associated with a module, nor are any processes associated with a module. The module is the unit of code loading. Erlang offers a facility for dynamically replacing code while a system is running. When this occurs a whole module is imported into the system to replace a previously loaded module.

A module begins with a number of declarations which are followed by the functions of the module. The *Wobbly Invaders* program (figure 2.9 and figure 2.10) will be used to illustrate the declarations (these declarations are sometimes known as attributes).

The **module** declaration on line 1 identifies the module name. Note: the

```
% erl
Erlang (JAM) emulator version 4.6

Eshell V4.6  (abort with ^G)
1> ot:ot(1).
'positive natural'
2> ot:ot(0).
integer
3> ot:ot(-1).
'negative integer'
4> ot:ot({1}).
tuple
5> ot:ot([1,2,3]).
list
6>
```

Figure 2.8: Output from using *ot*

file containing the module must be named with the module name suffixed with a *.erl*. In this case the module *wi* is stored in the file *wi.erl*. The **import** declaration on line 2 allows the *create* and *config* functions contained in the *gs* module to be accessed without prefixing them with their module names. The use of these functions can be compared with their use in the *Tic-Tac-Toe* program in chapter 1. The **export** declaration makes functions contained in this module available to other modules. If a function is not exported it is inaccessible to functions outside its own module.

Functions in other modules can be accessed in 2 ways:

- Importing the function allows the function to be called without mentioning the module name. Eg:

  start()

- A fully qualified function name includes the module name a colon and the function name. Eg:

  gs:start()

If a module contains a function with the same name as an imported function, fully qualified names should be used to access the function in the other module. A fully qualified function name is required to access two functions with the same name and number of arguments that are exported from two different modules.

2.7 Built In Functions

Erlang defines a special class of functions known as Built In Funtions or BIFs. In an interpreted environment these functions are built in to the interpreter. BIFs are members of the module *erlang*.

```
-module(wi).
-import(gs, [create/3, config/2]).
-export([init/0]).

init() ->
    S = gs:start(),
    Win = create(window, S, [{width, 200}, {height, 200},
    {title, 'Attack of the Wobbly Invaders'}]),
    Canvas = create(canvas, Win, [{width, 200}, {height, 200},
    {bg, white}]),
    config(Win, {map, true}),
    loop(Canvas, 0, 0, 1).


loop(C, X, Y, P) ->
    drawwi(C, X, Y, P),
    receive
        {gs, Id, destroy, Data, Arg} ->
            bye
    after 500 ->
        erasewi(C, X, Y),
        if
            Y == 200 ->
                bye;
            X == 200 ->
                loop(C, 0, Y+20, -P);
            true ->
                loop(C, X+20, Y, -P)
        end
    end.

drawwi(C, X, Y, 1) ->
    create(image,C,[{load_gif,"thing1.gif"}, {coords, [{X,Y}]}]);
drawwi(C, X, Y, -1) ->
    create(image,C,[{load_gif,"thing2.gif"}, {coords, [{X,Y}]}]).

erasewi(C, X, Y) ->
    create(rectangle,C,[{coords,[{X,Y},{X+20,Y+20}]},
    {fg, white}, {fill,white}]).
```

Figure 2.9: *Wobbly Invaders* Source Code (*wi.erl*)

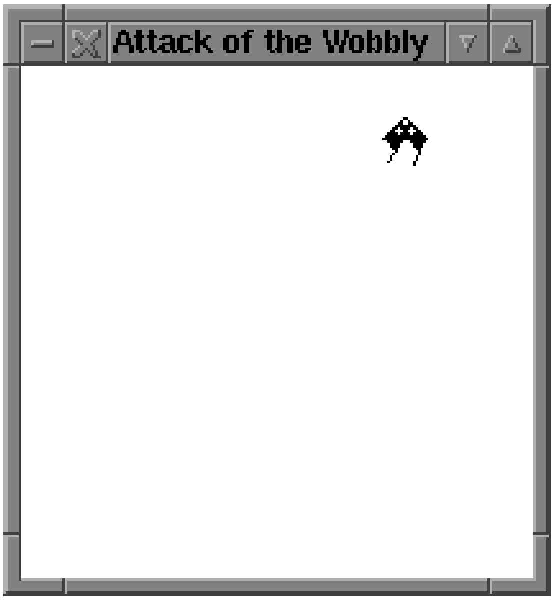


Figure 2.10: The *Wobbly Invaders* in Action

2.8 Resources

The code files mentioned in this chapter are:

tayser.erl
factprg2.erl
ot.erl
wi.erl
thing1.gif
thing2.gif

These files can be retrieved from:

`http://www.serc.rmit.edu.au/~maurice/erlbk/eg/fund`.

2.9 Exercises

1. Alter the *tayser* program so that all variable names are not reused in different clauses. Test the program to convince yourself that the old and new versions of the program work identically.

2. Alter the *Wobbly Invaders* program to use fully qualified functions instead of the import directive.

Chapter 3

Writing Functions

This chapter initially looks at the differences between Erlang and other languages. It then proceeds to describe how functions can be classified and using the classification illustrates how functions can be written in the language.

3.1 Procedural versus Declarative

Languages like C, Pascal, C++, Ada, Java, Cobol and Fortran are **procedural** languages. In these languages the programmer describes in detail *how* to solve a problem. Erlang is a **declarative** language. In a declarative language, the programmer describes *what* the problem is. The difference between the two styles of programming is best seen by example. Two programs have been written to solve the following problem:

> The problem: A person takes a mortgage of $10000 at 6.15% per annum and makes monthly payments of $200. How many months does it take to clear the debt?

The solution in Fortran 77 is given in figure 3.1. This solution is typical for other procedural languages as it uses a loop to calculate the outstanding amount of the loan while the loan is greater than zero dollars. The process of solving the problem is stated more clearly than the objectives of the program.

The solution in Erlang is given in figure 3.2. The Erlang solution differs from the Fortran solution in many ways. The most striking difference is in the *cnt* function. The first clause of the *cnt* function states that a solution has been found when the outstanding amount ($Outs$) has dropped to or below zero. If a solution has not been found the second clause of the function describes where to look for the next result. Naturally there remains a procedural element to the second clause – it describes how to calculate the next step – but the emphasis is on the *what* of the problem. Another significant difference between the Erlang solution and the Fortran solution is the lack of an iteration construct in Erlang. Erlang employs recursion instead. The absence of iteration encourages declarative programming practices. Guards and pattern matching are two methods of clearly describing aspects of the problem.

The output from the two programs is shown in figure 3.3.

```
      PROGRAM MORT

C Calculate the length of a morgage in months
C OUTS - outstanding amount
C INTR - interest charged
C RATE - monthly interest rate
C N - number of months
C REPAY - Repayment
      REAL OUTS, INTR, RATE, REPAY
      INTEGER N

C Initial values
      OUTS = 10000.0
      RATE = (6.15 / 100.0) / 12.0
      REPAY = 200.0
      N = 0

10    INTR = OUTS * RATE
      OUTS = OUTS + INTR
      OUTS = OUTS - REPAY
      N = N + 1
      IF (OUTS .GT. 0.0) GO TO 10

20    WRITE (6, FMT=100) N
100   FORMAT (I5)

      CLOSE(6)
      STOP
      END
```

Figure 3.1: Solution to Mortgage problem in Fortran 77: *mort.f*

```
-module(mort).
-export([mort/4]).

% Calculate the number of repayments required to pay
% a mortgage of Amt repaid at NumRep payments of Repay per
% year with interest taken NumRep times using an annual
% interest rate of Rate

mort(Amt, Rate, NumRep, Repay) ->
        AddRate = Rate / 100.0 / NumRep,
        cnt(Amt, Repay, AddRate, 0).

cnt(Outs, Sub, AddRate, N) when Outs =< 0.0 ->
        N;
cnt(Outs, Sub, AddRate, N) ->
        Add = Outs * AddRate,
        cnt(Outs - Sub + Add, Sub, AddRate, N+1).
```

Figure 3.2: Solution to Mortgage problem in Erlang: *mort.erl*

```
% mort
   58

% erl
Erlang (JAM) emulator version 4.5.3

Eshell V4.5.3  (abort with ^G)
1> mort:mort(10000.0, 6.15, 12, 200.0).
58
```

Figure 3.3: The output of *mort.f* and *mort.erl*

Both the Fortran solution and the Erlang solution presented have used a simulation approach to solving the mortgage problem. With deeper insight into the problem a more direct mathematical solution can be created. This solution is shown in figure 3.4.

```
-module(mort2).
-export([mort/4]).
-import(math, [log/1]).

% Calculate the number of repayments required to pay
% a mortgage of Amt repaid at NumRep payments of Repay per
% year with interest taken NumRep times using an annual
% interest rate of Rate

mort(Amt, Rate, NumRep, Repay) ->
        AddRate = Rate / 100.0 / NumRep,
        repayments(Amt, AddRate, Repay).

repayments(Loan, Rate, Payment)
        when Loan >= 0, Rate == 0, Payment > Loan*Rate ->
        ceiling(Loan/Payment);
repayments(Loan, Rate, Payment)
        when Loan >= 0, Rate > 0, Payment > Loan*Rate ->
        ceiling(-log(1.0 - Rate*Loan/Payment)/log(1.0 + Rate)).

ceiling(X) ->
        N = trunc(X),
        if
                N < X -> N+1;
                N >= X -> N
        end.

```

Figure 3.4: A More Insightful Solution to Mortgage problem in Erlang: *mort2.erl*

3.2 A Taxonomy of Functions

Functions can be collected into groups based on the relation between the volume of their input data to their output data. One set of groups collects functions into the categories:

- Transformation
- Reduction
- Construction
- Reduction and Construction

In general all functions transform their input data into their output data, however, in this classification the volume of data is not changed, it is merely converted into another form. Constructive functions make larger data structures from their input data and reducing functions condense their input data into a smaller data structure. These classifications of functions can be demonstrated with both tuples and lists, although it is somewhat easier to write examples using lists.

3.2.1 Transformation

The sine function *math:sin* in the Erlang math library is a simple example of a transformation.

3.2.2 Reduction

A classic example of a function that reduces its inputs is the *len* function (see figure 3.5) which operates similarly to the *length* function provided by Erlang. This function accepts a list and returns the number of elements in the list.

```
-module(len).
-export([len/1]).

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

Figure 3.5: An implementation of *length* in Erlang: *len.erl*

The reductionist approach taken here stems from knowing what the zero length case looks like; and knowing that a list with one less element can be made by taking the tail of the input list and the current list length is one more than the tail of the list. The list is reduced until it is empty then each stage adds one to the value returned by its reduced list. The top level function returns the length of the list. Figure 3.6 shows a modified program and the output at each stage.

3.2.3 Construction

A function that inserts a node in a Binary Search Tree (BST) is an example of a constructive function. Function *insert* from the *bst* module (see figure 3.7) demonstrates the construction of a tree.

As with reduction, at least one base case needs to be known and induction is used to construct other cases. In the BST example the base case is the construction of a single node tree from an empty tree. All other cases are constructed by locating a suitable *nil* leaf node, replacing it with a newly constructed node and then copying the remainder of the tree.

Reduction type functions have a fairly obvious point at which they are completed, when they have exhausted the resource they are reducing. Construction

```
-module(lens).
-export([len/1]).

len([]) ->
        0;
len(X) ->
        [H|T] = X,
        io:format("~w ~w~n", [H, T] ),
        LenT = len(T),
        io:format("~w ~w ~w ~w~n", [1 + LenT, H, LenT, T] ),
        1 + LenT.
```

```
% erl
Erlang (JAM) emulator version 4.5.3

Eshell V4.5.3  (abort with ^G)
1> c(lens).
{ok,lens}
2> lens:len([a,b,c,d,e,f,g]).
a [b,c,d,e,f,g]
b [c,d,e,f,g]
c [d,e,f,g]
d [e,f,g]
e [f,g]
f [g]
g []
1 g 0 []
2 f 1 [g]
3 e 2 [f,g]
4 d 3 [e,f,g]
5 c 4 [d,e,f,g]
6 b 5 [c,d,e,f,g]
7 a 6 [b,c,d,e,f,g]
7
3>
```

Figure 3.6: *len* in Action: *lens.erl*

```
-module(bst).
-export([insert/2, prefix/1, list_to_bst/1]).

% A BST is composed of nodes {Left, Item, Right}
% an empty tree is denoted by a nil.

% insert(Tree, Item) - Insert an item into a tree

insert(nil, Item) ->
  {nil, Item, nil};
insert({Left, Val, Right}, Item) ->
  if
    Item =< Val ->
      {insert(Left, Item), Val, Right};
    Item > Val ->
      {Left, Val, insert(Right, Item)}
  end.

% prefix(Tree) - Prefix Search

prefix(nil) ->
  nil;
prefix({Left, Val, Right}) ->
  LR = prefix(Left),
  RR = prefix(Right),
  if
    LR =/= nil, RR =/= nil ->
      lists:append(LR, lists:append([Val], RR));
    LR =/= nil ->
      lists:append(LR, [Val]);
    RR =/= nil ->
      lists:append([Val], RR);
    true ->
      [Val]
  end.

% list_to_bst(List) - Convert a list to a bst

list_to_bst(L) ->
  list_to_bst(L, nil).

list_to_bst([], Tree) ->
  Tree;
list_to_bst([H|T], Tree) ->
  list_to_bst(T, insert(Tree, H)).
```

Figure 3.7: *bst.erl*

type functions must be limited to stop constructing larger and larger data structures without limit. Several methods are available including the Reduction / Construction type function provides one method (see section 3.2.4). Two other methods will be discussed here: performing only N steps, and counters.

The *insert* function uses the performing only N steps mechanism. This function creates a tree which has exactly one extra node, one step in the process of creating a tree. There is no risk of this function regressing infinitely.

An example of a function using a counter is the *ndupl* function in figure 3.8. This function creates a list of a specified size with copies of a value.

```
-module(ndupl).
-export([ndupl/2]).

% ndupl(Item, N) - make a list containing N Items

ndupl(_, 0) ->
        [];
ndupl(Item, N) ->
        [Item | ndupl(Item, N-1)].
```

Figure 3.8: *ndupl.erl*

The performing only N steps mechanism differs from the counter mechanism and Reduction / Construction type functions in that there are no counters and no data structures are reduced.

3.2.4 Reduction / Construction

The Reduction / Construction mechanism uses the Reduction mechanism to extract data from one data structure and the Construction mechanism to add the element to a new data structure. Using reduction ensures that the operation is finite.

The function *list_to_bst* from the *bst* module (see figure 3.7) reduces a list and builds a tree.

Another example of this mechanism is the function *flatten* (see figure 3.9) which takes in a list of lists and produces a list containing the elements of the list of lists but without any lists. The work is done in the function *l2:flatten/2*. This function removes elements from its second argument and appends them to its first argument. If the head of the second argument is a list flatten is invoked on it and the result is appended to the first argument. This function can be thought of as stripping the brackets off each list until a single list remains and then appending the single list to the end of an already flattened list. When the second argument is empty the function returns the first argument.

The output of the function *flatten* can be seen in figure 3.10.

```
-module(l2).
-export([flatten/1]).
-import(lists,[append/2]).

flatten(L) ->
        flatten([], L).

flatten(L, []) ->
        L;
flatten(L, [H|T]) when list(H) ->
        flatten(append(L, flatten(H)), T);
flatten(L, [H|T]) ->
        flatten(append(L, [H]), T).
```

Figure 3.9: *l2.erl*

```
% erl
Erlang (BEAM) emulator version 4.6.4

Eshell V4.6.4  (abort with ^G)
1> l2:flatten([1,2,3,[4,5,6,7],8,[9]]).
[1,2,3,4,5,6,7,8,9]
2> l2:flatten([[1,2,3],[[[4],5],6],[],7]).
[1,2,3,4,5,6,7]
3>
```

Figure 3.10: The output of *flatten*

3.3 Resources

The code files mentioned in this chapter are:

mort.f
mort.erl
mort2.erl
len.erl
lens.erl
bst.erl
ndupl.erl
l2.erl
app.erl

These files can be retrieved from:

`http://www.serc.rmit.edu.au/~maurice/erlbk/eg/wrtfn`.

3.4 Exercises

1. Erlang provides a function *lists:append* which joins two lists together, implement your own function *app* that performs the append operation. (**Do NOT peek**; the answer is given in figure 3.11).
2. Write a narrative description of how the *app* function in figure 3.11 works.

```
-module(app).
-export([app/2]).

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

Figure 3.11: The append function (*app*) in *app.erl*

Chapter 4

Choices

In previous chapters selection of which piece of code to execute has generally been made using pattern matching and function head guards. A number of choice functions have been mentioned and used in examples, but without detailed explanation. This chapter will introduce the **if** and **case** functions.

Care must be taken when using choice functions to ensure that the single assignment property of variables in Erlang is not violated. Runtime errors will be generated if an unbound variable is accessed or a bound variable is assigned to.

The fragment of C in figure 4.1 illustrates two choice mechanisms present in the C language. The **if** mechanism is a statement and hence does not return a value. The **?:** mechanism is an operator and behaves like a function in that it returns a value. Languages such as Ada, Java, and Fortran, tend to provide statement based choice mechanisms. In contrast, Erlang's choice mechanisms return values.

```
int max(int a, int b)
{
        if (a > b)
                return a;
        else
                return b;
}

int min(int a, int b)
{
        return a < b ? a : b;
}

```

Figure 4.1: *max* and *min* in C

Figure 4.2 shows the Erlang code that implements the functions *max* and *min* in Erlang.

The following sections will discuss the implementation of choice using **if**, **case** and function heads. A function that determines the correct minimal change to

```
-module(maxmin).
-export([max/2,min/2]).

max(A,B) ->
        if
                A > B -> A;
                true -> B
        end.

min(A,B) ->
        if
                A < B -> A;
                true -> B
        end.
```

Figure 4.2: *max* and *min* in Erlang

be delivered by a vending machine will be used as an example. The cent is the unit of currency in Australia and the available coins are 2 dollar, 1 dollar, 50 cent, 20 cent, 10 cent, and 5 cent. As the 2 cent and 1 cent coins were withdrawn from service the following additional rule applies: values are rounded to the nearest 5 cents. This rule requires 2 and 1 cent amounts be rounded to 0, and that 3 and 4 cent amounts be rounded to 5 cents.

4.1 If

The **if** construct consists of a series of guards and sequences separated by semicolons. The syntax of the **if** construct is shown below:

if
    Guard1 $->$
        Seq1;
    ...
    GuardN $->$
        SeqN
end

The sequence associated with the first matching guard is executed. If no guard matches an error occurs. When the atom *true* is used as a guard it acts as a catch all – a catch all will match any pattern. Figure 4.3 ilustrates the use of the **if** construct.

4.2 Case

The **case** construct consists of a series of patterns – with optional guards – and sequences separated by semicolons. The syntax of the **case** construct is shown below:

```
-module(chg1).
-export([change/1]).

change(X) ->
        BaseSum =  (X div 5) * 5,
        Delta = X - BaseSum,
        if
                Delta >= 3 -> change(BaseSum + 5, []);
                true -> change(BaseSum, [])
        end.

change(X, L) ->
        if
                X >= 200 -> change(X - 200, ['2.00' | L]);
                X >= 100 -> change(X - 100, ['1.00' | L]);
                X >=  50 -> change(X -  50, ['50' | L]);
                X >=  20 -> change(X -  20, ['20' | L]);
                X >=  10 -> change(X -  10, ['10' | L]);
                X >=   5 -> change(X -   5, ['5' | L]);
                true -> L
        end.
```

Figure 4.3: *chg1.erl*

case Expr **of**
Pattern1 [**when** Guard1] – > Seq1;
Pattern2 [**when** Guard2] – > Seq2;
...
PatternN [**when** GuardN] – > SeqN
end

The expression – *Expr* – is evaluated before any of the patterns are evaluated. The sequence associated with the first matching pattern to the output of *Expr* is executed. If no pattern matches the result of the expression a match error will occur. The pattern '_' is a catch all will match anything and can be used to avoid the risk of a match error.

Figure 4.4 ilustrates the use of the **case** construct.

4.3 Function Heads

As mentioned earlier in chapter 2 the clauses of a function can be selected for execution based on the arguments contained in the head of the clause or by guards.

Figure 4.4 ilustrates the example problem implemented code selected using the guards present in the function heads.

```
-module(chg2).
-export([change/1]).

change(X) ->
        BaseSum =  (X div 5) * 5,
        Delta = X - BaseSum,
        case Delta of
                3 -> Rounded = BaseSum + 5;
                4 -> Rounded = BaseSum + 5;
                _ -> Rounded = BaseSum
        end,
        change(Rounded, [200, 100, 50, 20, 10, 5], []).

change(X, VL, L) ->
        case {X, VL} of
                {X, []} ->
                        L;
                {X, [H|T]} when X >= H ->
                        change(X - H, VL, [toatom(H) | L]);
                {X, [H|T]} ->
                        change(X, T, L)
        end.

toatom(X) ->
        case X of
                200 -> '2.00';
                100 -> '1.00';
                50 -> '50';
                20 -> '20';
                10 -> '10';
                5 -> '5'
        end.
```

Figure 4.4: *chg2.erl*

```
-module(chg3).
-export([change/1]).

change(X) ->
        BaseSum =  (X div 5) * 5,
        Delta = X - BaseSum,
        change(round(BaseSum, Delta), []).

round(X, Y) when Y >= 3 ->
        X + 5;
round(X, Y) ->
        X.

change(X, L) when X >= 200 ->
        change(X - 200, ['2.00' | L]);
change(X, L) when X >= 100 ->
        change(X - 100, ['1.00' | L]);
change(X, L) when X >= 50 ->
        change(X - 50, ['50' | L]);
change(X, L) when X >= 20 ->
        change(X - 20, ['20' | L]);
change(X, L) when X >= 10 ->
        change(X - 10, ['10' | L]);
change(X, L) when X >= 5 ->
        change(X - 5, ['5' | L]);
change(X, L) ->
        L.
```

Figure 4.5: *chg3.erl*

4.4 Resources

The code files mentioned in this chapter are:

cch.c
maxmin.erl
chg1.erl
chg2.erl
chg3.erl
erasto.erl

These files can be retrieved from:

`http://www.serc.rmit.edu.au/~maurice/erlbk/eg/choice`.

4.5 Exercises

1. Eratosthenes (276 - 196 BC) a Greek astronomer developed a technique for extracting prime numbers from a set of numbers. A prime number is only divisible by 1 and itself. His technique (The Sieve of Eratosthenes) involves writing down all the numbers from 3 to some value N. He then marked each number that was a multiple of 2. He then located the next unmarked number and removed all multiples of it. This process is repeated until the next unmarked number is greater than the square root of N.

   Implement the *Sieve of Eratosthenes* in Erlang. (Hint you may want to think about what marking means in the algorithm.)

   (**Do NOT peek**; the answer is given in figure 4.6).

2. Examine the examples of *change* in this chapter and identify which choice mechanisms best suit the problem. Discuss the benefits and drawbacks of each mechanism. Write an Erlang implementation which uses the most appropriate choice mechanisms for each decission point in the program.

3. Rewrite the *Sieve of Eratosthenes* shown in figure 4.6 using other choice mechanisms.

```
-module(erasto).
-export([era/1]).

era(N) ->
        [1 | era(math:sqrt(N), fill(2, N))].

fill(L, L) ->
        [L];
fill(L, H) when H > L ->
        [L | fill(L+1, H)].

era(Max, L) when hd(L) =< Max ->
        Prime = hd(L),
        [Prime | era(Max, sieve(Prime, L))];
era(Max, L) ->
        L.

sieve(N, []) ->
        [];
sieve(N, [H|T]) when H rem N == 0 ->
        sieve(N, T);
sieve(N, [H|T]) ->
        [H | sieve(N, T)].
```

Figure 4.6: The Sieve of Eratosthenes: *era* in *erasto.erl*

Chapter 5

Processes and Messages

Erlang provides easy access to lightweight processes, simple but powerful Inter-Processes Communication (IPC) mechanisms, and easy to use distributed processing. This chapter addresses the mechanisms which provide these facilities.

5.1 Processes

A process is the basic unit of execution of an Erlang program. The name of a process or Process IDentifier (PID) is used. Processes are recipients of messages and hold the running state of a thread of execution.

A process is started in Erlang by using the **spawn** function. The simple form of the **spawn** function takes a series of arguments including the module name, the name of the function and a list of arguments to a function. The PID of the new process is returned to the calling process.

The syntax of the spawn function is:

pid = **spawn**(*module*, *function*, [*args* ...])
pid = **spawn**({*module*, *function*}, [*args* ...])

Figure 5.1 shows an interactive way of using the **spawn** function to create a new shell (which takes over terminal IO).

In general, **spawn** creates a process and returns immediately to the calling process. The called process is then independent of its creator.

5.1.1 Finding a Processes Name

Erlang processes can determine their Process ID by calling the *self* function. Figures 5.2 and 5.3 illustrate the use of the **spawn** and **self** functions.

The result of the self function is a data element called a *pid*. PIDs are not human readable, however, there is an on-screen representation that is displayed whenever a pid is output. Typing in this representation in the Erlang shell to use it as a pid will fail.

```
% erl
Erlang (BEAM) emulator version 4.6.4

Eshell V4.6.4  (abort with ^G)
1> F = 2.
2
2> spawn(shell, start, [[]]).
<0.29.0>
Eshell V4.6.4  (abort with ^G)
1> F.
** exited: {{unbound,'F'},{erl_eval,expr,3}} **
2> exit().
** Terminating shell **
3> F.
2
4>
```

Figure 5.1: Spawning a Second Shell From an Erlang Shell

```
-module(spwslf).
-export([start/0, newfn/0]).

start() ->
        MyPid = self(),
        io:format("demo: ~w~n", [MyPid]),
        NewPid = spawn(spwslf, newfn, []),
        io:format("demo: ~w~n", [MyPid]).

newfn() ->
        MyPid = self(),
        io:format("newfn: ~w~n", [MyPid]).
```

Figure 5.2: Spawning and Identifying Processes: *spwslf.erl*

```
% erl
Erlang (JAM) emulator version 4.5.3

Eshell V4.5.3  (abort with ^G)
1> spwslf:start().
demo: <0.20.0>
demo: <0.20.0>
newfn: <0.23.0>
ok
2>
```

Figure 5.3: Exercising *spwslf.erl*

5.1.2 Process Dictionary

Each process has an associative store – known as the Process Dictionary – that is private to the process. Use of the process dictionary is discouraged as the process dictionary breaks aspects of the functional paradigm, and programs which use it tend to be harder to modify and maintain.

The functions **put**, **get**, **erase**, and **get_keys** are used to access the process dictionary.

The function **put** adds a key value pair to the process dictionary. If the key already exists in the process dictionary the key value pair in the dictionary is replaced and the old value is returned, otherwise the atom *undefined* is returned. The contents of the process dictionary can be returned as a set of key value tuples by using **get** with no arguments. The value associate with a particular key can be found by calling **get** with the key. The entire process dictionary can be deleted by calling **erase** with no arguments. The erased contents of the process dictionary are returned. An individual key value pair can be erased by calling **erase** with the key. If the key to be erased exists in the process dictionary, the old value is returned, otherwise the atom *undefined* is returned. A list of all the keys which correspond to a value in the process dictionary can be found using the **get_keys** function with the value as the argument.

The process dictionary is exercised in figure 5.4 and the code is shown in figure 5.5. This example implements a simple rolodex or phone book.

5.1.3 Message Buffer

Associated with each process is a logical buffer which contains all messages that are waiting to be received by the process. Most implementations of Erlang use a common pool of buffer space which is shared by all the processes on a node.

5.2 Messages

Messages are sent to processes using the **!** operator. this operator takes two arguments the PID or a registered name and an expression:

pid ! **expression**

The **!** operator always appears to send a message. If there is no destination or there is no space to queue the message at its destination then the message is discarded. Erlang offers **NO guarantee of message delivery**.

The result of the *expression* is transmitted to the process named by *pid*. The message is stored until the process chooses to receive it by executing a **receive** expression. **Receive** has a similar structure to **case**, except that it operates on the process's message queue.

```
receive
    Message1 [when Guard1]  -> Act1;
    Message2 [when Guard2]  -> Act2;
    ...
    MessageN[when GuardN] -> ActN
after
    TimeOut -> ActT
end
```

```
% erl
Erlang (BEAM) emulator version 4.6.4

Eshell V4.6.4  (abort with ^G)
1> prcdct:teldir().
Enter name and phone number (blank line to end)
name phone> john 92914592
name phone> max 93442156
name phone> fred 9834231
name phone>
Operation
 1) Search 2) Add/Change 3) List Names 4) Delete    0) Quit
op > 3
fred
max
john
op > 4
name > fred
9834231
op > 1
name > fred
undefined
op > 3
max
john
op > 2
Enter name and phone number (blank line to end)
name phone> jane 94514567
name phone>
op > 3
jane
max
john
op > 1
name > jane
94514567
op > 0
true
2>
```

Figure 5.4: Exercising the Process Dictionary

```
-module(prcdct).
-export([teldir/0]).

teldir() -> getdata(), menu(), querylp().

getdata() ->
    io:format("Enter name and phone number ", []),
    io:format("(blank line to end)~n", []),
    getlp().

getlp() ->
    Line = io:get_line('name phone> '),
    Lst = string:tokens(Line, " \n"),
    getlp(Lst).

getlp([Name, Phone]) ->
    put(Name, Phone),
    getlp();

getlp(Lst) when length(Lst) == 0 ->
    true;
getlp(_) ->
    io:format("Error~n"),
    getlp().

menu() ->
    io:format("Operation~n 1) Search 2) Add/Change "),
    io:format("3) List Names 4) Delete    0) Quit~n", []).

querylp() -> querylp(io:fread('op > ', "~d")).

querylp({ok, [0]}) -> true;
querylp({ok, [1]}) -> search(), querylp();
querylp({ok, [2]}) -> getdata(), querylp();
querylp({ok, [3]}) -> lstname(), querylp();
querylp({ok, [4]}) -> delete(), querylp().

getnam() ->
    Line = io:get_line('name > '),
    getnam(string:tokens(Line, " \n")).

getnam([L]) -> L;
getnam(_) -> io:format("Error~n"), getnam().

search() -> io:format("~s~n", [get(getnam())]).

lstname() -> lstname(get()).

lstname([]) -> true;
lstname([{Key, Value}|T]) -> io:format("~s~n", [Key]), lstname(T).

delete() -> io:format("~s~n", [erase(getnam())]).
```

Figure 5.5: Code for Exercising the Process Dictionary: *prcdct.erl*

Unbound variables in *Message* act as wildcards and are bound to values when a suitable match is found. If *Message* is an unbound variable it will match any message and be bound to the contents of the message. An unbound variable acts as a catch all. *ActT* is executed if *TimeOut* milliseconds elapse and there is no message that is matched by a pattern in the receive function. If the value of *TimeOut* is *infinity* then the receive does not time out and *ActT* is not executed. If the value is *0* then all the messages are checked and if none match *ActT* is executed immediately.

The time order of messages from one process to another is preserved and the receive statement selects the oldest message that successfully pattern matches.

It is the responsibility of the process to remove irrelevant or invalid messages from its message buffer. Failure to do this can result in losing valid messages when space is no longer available to store messages. Long running processes usually have a catch all at some point in the program to remove messages which do not match any pattern required by the program.

In section 5.1.2 the use of a processes dictionary was discouraged, the next example illustrates how the functionality of the process dictionary can be achieved in an extensible manner without using the existing process dictionary functions discussed earlier.

Figure 5.6 provides an implementation of Process Dictionary functionality through the use of messages and processes. Only two functions are illustrated: *get* and *put*. The program is exercised in figure 5.7.

A process receiving a message does not know which process sent it. Because messages are anonymous, a process can only reply to a message if the message contains the pid of the sending process. The *self* call is used to gain a process's pid which can then be sent in a message so the destination process can reply to the message.

5.3 Time Delays

Erlang uses a degenerate form of **receive** to generate a time delay. Figure 5.8 shows code that implements a 10 second count down.

5.4 Distribution

Distribution is almost trivially easy in Erlang. Starting a process on another node is a straight forward extension of the syntax of the **spawn** function.

The first task is to start a remote node with a name. This can be done in two ways using the **-sname** option or the **-name** option.

The example in figure 5.9 shows both methods and assumes that the machine the Erlang node is being started on is called *atum.castro.aus.net*. The node name in each case will be *maurice*.

The two methods are almost identical with the exception that the latter generates a fully qualified name.

To start a process on a remote node the node name is introduced into the **spawn** function:

$$pid = \textbf{spawn}(node,\ module,\ function,\ [args\ \ldots])$$

```
-module(dct).
-export([start/0,get/2,put/3,dct/1]).

start() ->
    spawn(dct,dct,[[]]).

get(Pid, Key) ->
    Pid ! {get, self(), Key},
    receive
        X -> X
    end.

put(Pid, Key, Value) ->
    Pid ! {put, self(), Key, Value},
    receive
        X -> X
    end.

dct(L) ->
    NewL = receive
        {put, Pid, Key, Value} ->
            {Code, List} = insert(L, [], Key, Value),
            Pid ! Code,
            List;
        {get, Pid, Key} ->
            Code = find(L, [], Key),
            Pid ! Code,
            L;
        X ->
            L
    end,
    dct(NewL).

insert([], N, Key, Value) ->
    {undefined, [{Key, Value}|N]};
insert([{Hkey, HVal} |T], N, Key, Value) when Key == Hkey ->
    {HVal, lists:append(N,[{Key, Value} | T])};
insert([H|T], N, Key, Value) ->
    insert(T, [H|N], Key, Value).

find([], N, Key) ->
    undefined;
find([{Hkey, HVal}|T], N, Key) when Key == Hkey ->
    HVal;
find([H|T], N, Key) ->
    find(T, [H|N], Key).
```

Figure 5.6: Source code for a Dictionary: *dct.erl*

```
% erl
Erlang (BEAM) emulator version 4.6.4

Eshell V4.6.4  (abort with ^G)
1> P = dct:start().
<0.28.0>
2> dct:put(P,fred,123).
undefined
3> dct:get(P,fred).
123
4> dct:get(P,fred).
123
5> dct:put(P,john,456).
undefined
6> dct:get(P,fred).
123
7> dct:get(P,john).
456
8> dct:put(P,john,678).
456
9> dct:get(P,john).
678
10> dct:get(P,fred).
123
11>
```

Figure 5.7: Exercising *dct.erl*

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

start() ->
        cntdwn(10).

cntdwn(N) when N > 0 ->
        io:format("~w~n", [N]),
        receive
        after 1000 ->
                true
        end,
        cntdwn(N-1);
cntdwn(_) ->
        io:format("ZERO~n").
```

Figure 5.8: Source code for a Count Down: *cntdwn.erl*

```
atum_1% erl -sname maurice
Erlang (JAM) emulator version 4.5.3

Eshell V4.5.3  (abort with ^G)
(maurice@atum)1>


atum_1% erl -name maurice
Erlang (JAM) emulator version 4.5.3

Eshell V4.5.3  (abort with ^G)
(maurice@atum.castro.aus.net)1>
```

Figure 5.9: Starting a Distributed Node

Sending messages to a process on a remote node is identical to sending messages to a process on a local node.

5.5 Registered Names

All the prior examples of message sending have used explicit pids to identify where a message is to be sent. Registering a process allows a symbolic name to be associated with a Process ID. A symbolic name is particularly useful where a process is offering a service and that is widely used as it avoids the need to distribute the pid of that service to its potential users.

The function **register** is used to associate a name on the local node with a pid:

register(*name*, *pid*)

The association can be removed using **unregister**:

unregister(*name*)

And a pid can be recovered for a name using **whereis**, *undefined* is returned if the name is not associated with a pid.

whereis(*name*)

Registered names on the local node can be used instead of a pid in the '!' operation. Names on remote nodes can be accessed using '{*Name*, *Node*} ! *Message*'

5.6 Resources

The code files mentioned in this chapter are:

spwslf.erl
prcdct.erl
dct.erl
cntdwn.erl

These files can be retrieved from:

`http://www.serc.rmit.edu.au/~maurice/erlbk/eg/procmsg`.

5.7 Exercises

1. Modify *dct.erl* (see figure 5.6) to implement all the functions provided by the process dictionary.

2. Modify *prcdct.erl* to use the dictionary code you wrote in the previous exercise.

Chapter 6

Meta-programming

In C it is possible to use a pointer to a function to name a function and evaluate it. Figure 6.1 shows an example of where calling a function through a pointer to the function is used. The *qsort* function provided by C is made flexible by allowing the user to choose their own comparison functions.

The technique of writing functions which deal with a particular structure or problem, but use some function which is supplied by the user to customise the behaviour of the function to the exact situation is sometimes called meta-programming.

Erlang allows functions to be called by name. The **apply** function executes the function named in its arguments with a supplied set of arguments. Two forms of **apply** are supported by Erlang:

$$retval = \mathbf{apply}(module,\ function,\ [args\ \ldots])$$
$$retval = \mathbf{apply}(\{module,\ function\},\ [args\ \ldots])$$

The return value, *retval*, is the value returned by executing the function *module:function* with the arguments *args* .... The **apply** function is particularly useful for transforming lists and other data structures.

The classic example of **apply** is the *map* function. This function applies the supplied function to each element of a list. An implementation of this function is shown in figure 6.2 and a sample of its output is shown in figure 6.3.

6.1 Resources

The code files mentioned in this chapter are:

cptf.c
map.erl

These files can be retrieved from:

`http://www.serc.rmit.edu.au/~maurice/erlbk/eg/meta`.

```c
#include <stdio.h>
#include <stdlib.h>

char strarr[5][10] =
{
    "one", "two", "three", "four", "five"
};

int intarr[5] =
{
    5, 4, 3, 2, 1
};

int strcmp(const void *, const void *);

int intcmp(const void *a, const void *b)
{
    if (* (int *) a > * (int *) b)
        return 1;
    else if (* (int *) a < * (int *) b)
        return -1;
    else
        return 0;
}

int main(void)
{
    int i;
    for (i=0; i < 5; i++)
        printf("%s ", strarr[i]);
    printf("\n");
    qsort(strarr, 5, 10, &strcmp);
    for (i=0; i < 5; i++)
        printf("%s ", strarr[i]);
    printf("\n");
    for (i=0; i < 5; i++)
        printf("%d ", intarr[i]);
    printf("\n");
    qsort(intarr, 5, sizeof(int), &intcmp);
    for (i=0; i < 5; i++)
        printf("%d ", intarr[i]);
    printf("\n");
    return 0;
}
```

Figure 6.1: Sorting with *qsort* in C: *cptf.c*

```
-module(map).
-export([map/2]).

map(L, Fn) ->
        map(L, [], Fn).

map([], N, Fn) ->
        N;
map([H|T], N, Fn) ->
        map(T, [apply(Fn, [H]) | N], Fn).
```

Figure 6.2: Source code for *map*: *map.erl*

```
% erl
Erlang (JAM) emulator version 4.5.3

Eshell V4.5.3  (abort with ^G)
1> map:map([[1,2,3],[],[a,b]],{erlang,length}).
[2,0,3]
2>
```

Figure 6.3: Exercising *map.erl*

6.2 Exercises

1. Write a function which takes a list of lists and applies the head of the inner list as a function to the arguments remaining in the inner list. The results of this function should be returned as a list.

2. Write a version of *map* which applies its function to each element of each list in a potentially infinitely deep list of lists. The list of lists structure must be retained.

Chapter 7

Writing Efficient Code

Erlang is a relatively impoverished programming language in terms of the number of mechanisms it offers the programmer. This limits the complexity of the language, making it easy for programmers to learn and master. However, the absence of mechanisms like iteration make it harder for the programmer to communicate with the compiler the intention of the writer's code and hence make it harder for the compiler to generate efficient code.

This chapter covers methods used to make Erlang code both time and space efficient.

7.1 Last Call Optimisation

A simple model of a computer program consists of a stack, some memory and a set of instructions (*code*) that operate on these elements (see figure 7.1). The stack is used to store context information for functions and procedures. Arguments and a return address are *pushed* onto the stack before a function is called, these arguments are preserved until the function returns. Variables local to a function are also allocated on the stack. This allows a function to call other functions to do work for it without the other functions disturbing values local to the calling function. A stack pointer (*SP*) and a base pointer (*BP*) are used to identify where the stack ends and where the return address is located.

In procedural languages, such as C and Ada, iteration constructs such as loops are used to repeat operations. Erlang does not support iteration directly in the language and, furthermore, prevents the values of variables being reassigned. A naive implementation of Erlang would add a new stack frame to the stack each time a function was called and the system would soon run out of memory.

Fortunately there exists a case in which an Erlang function never returns to its calling function. In this case the calling stack frame can be overwritten (provided the original return address is preserved) with the new stack frame and as a result the stack does not grow.

If the last action of a clause is to call another function then the stack frame of the calling function can be overwritten. The overwriting of the stack frame is called last call optimisation. It is highly desirable to write clauses so that the last action of the clause is a function call as it both saves time and space. Time savings are gained in that an additional stack frame does not need to be

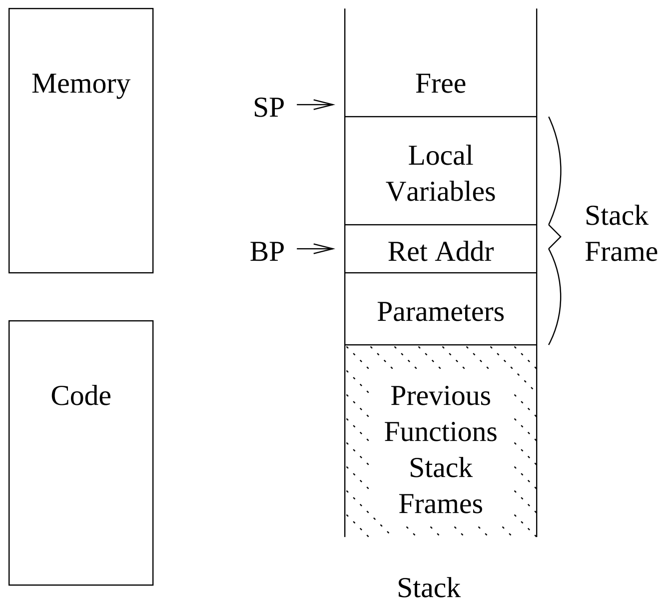


Figure 7.1: A Simple Model of a Program

traversed on returning from a function and space is saved by preventing stack growth.

Figure 7.2 shows two implementations of the *length* function which computes the length of a list. Both implementations are based on the observations that a list is one item larger than a list with the first element removed and a list with no elements has zero length. The first implementation, *len1*, uses a straight forward translation of the observations and as a consequence consumes one stack frame for each element in the list. The second implementation, *len2*, is more conservative in its use of the stack. Each time the final clause is called its stack frame is replaced resulting in more compact execution.

The second implementation can take advantage of last call optimisation since there are no operations remaining in the calling function which must be performed after the call to the new function.

7.2 Hashable Constructions

There are wide range of Erlang compilers deployed. This section will describe some programming practices that will allow suitably equipped Erlang compilers to produce faster code. The emphasis in this section is to discuss techniques which do not unduly complicate code nor slow down code on compilers which are not fitted with these optimisations. Some programming constructions in Erlang are well suited to optimisation.

A common optimisation for pattern matching language compilers is to examine the arguments for the pattern match and generate a hash of the arguments. This hash is used to jump directly to a function clause or case clause rather than performing a sequential match. Some Erlang compilers support this opti-

```
-module(len2).
-export([len1/1, len2/1]).

len1([H|T]) ->
        1 + len1(T);
len1([]) ->
        0.

len2(L) ->
        len2(0, L).

len2(N, []) ->
        N;
len2(N, [H|T]) ->
        len2(N+1, T).
```

Figure 7.2: Two implementations of *length*: *len2.erl*

misation on the first argument of a function head and the first argument of a case head.

Bit stuffing in the HDLC protocol will be used as an example of how a function can be coded to take advantage of these optimisations. HDLC is a link level protocol which uses the sequence 01111110 to designate both the beginning and the end of the message. This sequence cannot appear in the payload of the message on the wire. If sender of the message wants to transmit the data 01111110 to the receiver it is necessary to alter the data transmitted and to recover the original pattern at the receiving end of the link. The algorithm used is:

Transmitter:

If the previous five digits have been ones send a zero then transmit the next bit in the stream

Otherwise, Transmit each bit as it appears

Receiver:

If the previous five digits have been ones and a zero arrives the zero is discarded.

If the previous six digits have been ones and a zero arrives then the end of frame has been encountered.

If the previous six digits have been ones and a one arrives then an error has occurred.

Figure 7.3 shows an unoptimised implementation of the HDLC protocol. Figure 7.4 shows an improved version. The improvements consist of greater use of function heads for decision making and a reordering of the function arguments to allow hashing on the first argument to be exploited. A side effect of the rewrite has been to improve readability, the presence of two states – *message* and *start* – in the decoder has been made clearer.

```
-module(hdlc).
-export([enc/1,dec/1]).

delimit() ->
        [ 0, 1, 1, 1, 1, 1, 1, 0].

enc(L) ->
        X = enc(L, 0),
        lists:append(delimit(), lists:append(X, delimit())).

enc(L, 5) ->
        [0 | enc(L, 0)];
enc([H|T], N) ->
        if
                H == 1 ->
                        [1 | enc(T, N+1)];
                H == 0 ->
                        [H | enc(T, 0)]
        end;
enc([], _) ->
        [].

dec(L) ->
        {Code, List} = dec(L, start, []),
        {Code, lists:reverse(List)}.

dec([0, 1, 1, 1, 1, 1, 1, 0 | T], start, _) ->
        dec(T, message, []);
dec([H | T], start, _) ->
        dec(T, start, []);
dec([], start, _) ->
        {error, []};
dec([0, 1, 1, 1, 1, 1, 1, 0 | T], message, L) ->
        {ok, L};
dec([1, 1, 1, 1, 1, 1 | T], message, L) ->
        {error, L};
dec([1, 1, 1, 1, 1, 0 | T], message, L) ->
        dec(T, message, [1, 1, 1, 1, 1 | L]);
dec([H|T], message, L) ->
        dec(T, message, [H | L]);
dec([], message, L) ->
        {error, L}.
```

Figure 7.3: An Implementation of the HDLC Protocol: *hdlc.erl*

```
-module(hdlc2).
-export([enc/1,dec/1]).

delimit() ->
        [ 0, 1, 1, 1, 1, 1, 1, 0].

enc(L) ->
        X = enc(0, L),
        lists:append(delimit(), lists:append(X, delimit())).

enc(5, L) ->
        [0 | enc(0, L)];
enc(N, [1|T]) ->
        [1 | enc(N+1, T)];
enc(N, [0|T]) ->
        [0 | enc(0, T)];
enc(_, []) ->
        [].

dec(L) ->
        {Code, List} = dec(start, L, []),
        {Code, lists:reverse(List)}.

dec(start, [0, 1, 1, 1, 1, 1, 1, 0 | T], _) ->
        dec(message, T, []);
dec(start, [H | T], _) ->
        dec(start, T, []);
dec(start, [], _) ->
        {error, []};
dec(message, [0, 1, 1, 1, 1, 1, 1, 0 | T], L) ->
        {ok, L};
dec(message, [1, 1, 1, 1, 1, 1 | T], L) ->
        {error, L};
dec(message, [1, 1, 1, 1, 1, 0 | T], L) ->
        dec(message, T, [1, 1, 1, 1, 1 | L]);
dec(message, [H|T], L) ->
        dec(message, T, [H | L]);
dec(message, [], L) ->
        {error, L}.
```

Figure 7.4: An Improved Implementation of the HDLC Protocol: *hdlc2.erl*

7.3 Resources

The code files mentioned in this chapter are:

len2.erl
hdlc.erl
hdlc2.erl

These files can be retrieved from:

`http://www.serc.rmit.edu.au/~maurice/erlbk/eg/eff`.

7.4 Exercises

1. Examine your earlier programs and determine where last call optimisation can be applied to them. Rewrite the programs and confirm that the new programs provide the same answers as the original programs.

Chapter 8

Robust Programs

Running programs can fail for many reasons. Some of these failures can be controlled by the programmer, others are beyond the programmer's control. Robust systems must continue to function in the face of unexpected problems. Robust programs must be able to handle problems such as a lack of a critical resource like memory or disk space. Erlang provides a number of mechanisms that allow robust programs to be written. This chapter describes some of those mechanisms.

8.1 Catch and Throw

Exception mechanisms are provided by many languages including C++, Java, and Ada. Instead of scattering error trapping code throughout a program, the exception mechanism allows error handling code to be centralised. Under the exception paradigm, when a function encounters a problem that it cannot deal with an exception data structure is generated. The function stops and the exception is propagated up through each of the subroutines which are awaiting return values and caused the subroutine where the exception occurred to be called. The calling subroutines can choose to handle the exception and processing resumes in the exception handler with the exception data structure provided as an argument. Figure 8.1 shows a typical implementation of an exception in C++.

Erlang provides **catch** and **throw** mechanism which resembles the exception mechanism. In Erlang, the argument to throw is *thrown* up the chain of calling functions until it is *caught* by a catch. Note: Unlike C++ or Ada where an exception can be handled or passed on to another exception handler, Erlang requires that the first catch encountered handle the result of a throw.

Failures can result in throw like behavior. If a match operation fails, a function is evaluated with an incorrect or unsupported argument, or an arithmetic expression is evaluated with an invalid argument, then the Erlang run time system generates a throw containing a reason for the failure.

If a failure or a throw is not caught then the default behavior of the run time system is to terminate the process that failed abnormally.

Figure 8.2 shows how catch code can be used to protect a divide operation from bad data. Figure 8.3 shows the result of calling the code with various

```
#include <iostream.h>

class DivZeroErr
{
    public:
        DivZeroErr() {}
};

int div(int a, int b)
{
    if (b == 0)
        throw DivZeroErr();
    return(a / b);
}

int main()
{
    int a, b, r;
    cin >> a >> b;
    try
    {
        r = div(a, b);
        cout << r << endl;
    }
    catch (DivZeroErr error)
    {
        cout << "Attempt to Divide by Zero" << endl;
    }
}
```

Figure 8.1: An Example of Exception Handling in C++: *div.C*

correct and incorrect data.

Two types of errors generated by the Erlang run time system are shown in figure 8.3. The first and second errors are the product of the *io:read* function returning something other than the pattern {*ok, [A, B]*}. These resulted in a *badmatch* error. The final error shown was a result of attempting to divide by zero and a *badarith* error was returned.

The syntax of the *catch* operator is unusual in that it and its arguments must be surrounded by parenthesis. The syntax of the operator is shown below:

(**catch** *expr*)

The expression *expr* is evaluated and if no failure or throw occurs in the expression **catch** returns the result of the evaluation. If a failure or a throw occurs then **catch** returns a data structure. This data structure is either the argument of a throw or generated by the run time system.

Throw looks and behaves like a function, except it never returns. The syntax of throw is shown below:

throw(*expr*)

The expression is evaluated and the result is passed to the nearest catch. Throw can be used to simulate failures which would normally be generated by

```
-module(divide).
-export([divide/0]).

divide() ->
    X = (catch getarg()),
    case X of
        {'EXIT', Reason} ->
            io:format("Error ~w~n", [X]);
        {A, B} ->
            Y = (catch divide(A, B)),
            case Y of
                {'EXIT', Reason} ->
                    io:format("Error ~w~n", [Y]);
                Y -> Y
            end
    end.

getarg() ->
    {ok, [A, B]} = io:fread('Enter 2 numbers > ', "~d ~d"),
    {A, B}.

divide(A, B) ->
    D = A div B,
    io:format("~w~n", [D]).
```

Figure 8.2: Protecting Divide: *divide.erl*

```
% erl
Erlang (BEAM) emulator version 4.6.4

Eshell V4.6.4  (abort with ^G)
1> divide:divide().
Enter 2 numbers > 4 2
2
ok
2> divide:divide().
Enter 2 numbers > 1 2
0
ok
3> divide:divide().
Enter 2 numbers > a 1
Error {'EXIT',{{badmatch,error},{divide,getarg,0}}}
ok
4> divide:divide().
Enter 2 numbers > 1 a
Error {'EXIT',{{badmatch,error},{divide,getarg,0}}}
ok
5> divide:divide().
Enter 2 numbers > 2 0
Error {'EXIT',{badarith,{divide,divide,2}}}
ok
6>
```

Figure 8.3: Exercising *divide.erl*

the run time system.

Some common failures generated by the run time system include:

badarg is caused by attempting to use an incorrect argument type with a function. Eg. *math:sin(a)*

badarith is caused by providing an invalid argument to a mathematical operator. Eg. *1 + a*

badmatch is caused by attempting to assign to an incompatible pattern. Eg. *{ok, [A]} = {foo}*

case_clause occurs when no clause of a case statement matches the argument of case. Eg.

```
case {1,2} of
    {A} -> 1;
    {A,B,C} -> 3;
    {A,B,C,D} -> 4
end
```

function_clause occurs when no function clause matches the argument of a the function. Eg. *bin:bin(2)*

```
-module(bin).
-export([bin/1]).

bin(0) -> 0;
bin(1) -> 1.
```

if_clause occurs when no if clause is true. Eg.

```
if
    a == b -> 1;
    b == c -> 3
end
```

nocatch is caused by a throw not being caught.

noproc is caused by attempting to link (see section 8.5) to a non-existent process.

timeout_value is caused by attempting to use a non-integer as a timeout period in a receive. Eg.

```
receive
    X -> X
after 3.4 ->
    timeout
end
```

unbound is caused by attempting to access an unbound variable.

undef is caused by attempting to access a function which has not been defined.
Eg. *string:len("abc","a")*

Sometimes it is not convenient or possible to handle a failure at the first catch encountered. In this case the catch can re-throw the failure data structure to allow the failure to be handled at a higher level.

Figure 8.4 shows a program that adds hexadecimal numbers and outputs the result in decimal. The program uses catch and throw failure detection and generation. The *hexdec* function transforms one type of error (*function_clause*) caused by the presence of a non-hexadecimal digit into a *badchar* error. In addition the function re-throws thrown 2 element tuples containing the *fail* atom.

The output of the program is shown in figure 8.5. The second execution sequence was caused by entering a space at the prompt, resulting in an empty list being passed to the *hexcvt* function.

The **catch** and **throw** construct can be used to generate a non-local return from a function. Results can be passed directly up the call chain, avoiding intermediate functions by throwing the result to a catch. This practice can lead to confusing code and should be used with care.

8.2 Termination

A process can be terminated either by returning from the function that the process was started with or by calling the **exit** function.

The result of an **exit** occurring within the scope of a catch can be trapped using **catch**.

Exit looks and behaves like a function, except it never returns. Exit generates a signal of the form {*'EXIT'*, *Reason*}. Signals are similar to the data thrown by a **throw** to the extent that they can be caught by a **catch**. Signals can be generated by the process receiving them or by another process which knows the pid of the process to which the signal is to be sent. The syntax of exit is shown below:

exit(*expr*)

The expression *expr* is evaluated and the result becomes the reason either caught by a catch or displayed by the interpreter. The atom *normal* is special in that when a process exits normally no error report is displayed by the interpreter.

Another form of the **exit** function operates on processes other than the caller.

exit(*pid*, *expr*)

This function returns to its calling process. The process named by *pid* is signaled with the reason resulting from evaluating *expr*. The named process then behaves as if it had executed **exit**(*expr*).

Note that the second form of **exit** cannot be caught by a **catch** as it occurs outside the scope of the catch.

```
-module(hex).
-export([hexdec/1, hexadd/0]).

hexadd() ->
    X = (catch adder()),
    case X of
        {'EXIT', Reason} ->
            io:format("Error ~w~n", [X]);
        {fail, badchar} ->
            io:format("Error hex digits must are 0-9 a-f~n");
        {fail, nullarg} ->
            io:format("Error value required~n");
        Y -> Y
    end.

adder() ->
    {ok, [A]} = io:fread('Enter first number > ', "~a"),
    {ok, [B]} = io:fread('Enter second number > ', "~a"),
    hexdec(A) + hexdec(B).

hexdec(Atom) ->
    L = atom_to_list(Atom),
    case (catch hexcvt(L)) of
        {'EXIT', {function_clause, _}} ->
            throw({fail, badchar});
        {fail, X} ->
            throw({fail, X});
        R -> R
    end.

hexcvt([]) ->
    throw({fail, nullarg});
hexcvt(L) ->
    hexcvt(L, 0).

hexcvt([], N) ->
    N;
hexcvt([H|T], N) ->
    V = decval(H),
    hexcvt(T, N * 16 + V).

decval($0) -> 0;  decval($1) -> 1;  decval($2) -> 2;
decval($3) -> 3;  decval($4) -> 4;  decval($5) -> 5;
decval($6) -> 6;  decval($7) -> 7;  decval($8) -> 8;
decval($9) -> 9;  decval($a) -> 10; decval($b) -> 11;
decval($c) -> 12; decval($d) -> 13; decval($e) -> 14;
decval($f) -> 15.
```

Figure 8.4: Hex Conversion and Addition: *hex.erl*

```
% erl
Erlang (BEAM) emulator version 4.6.4

Eshell V4.6.4  (abort with ^G)
1> hex:hexadd().
Enter first number > 1
Enter second number > 2
3
2> hex:hexadd().
Enter first number > 1
Enter second number >
Error value required
ok
3> hex:hexadd().
Enter first number > 1
Enter second number > g
Error hex digits must are 0-9 a-f
ok
4> hex:hexadd().
Enter first number > 1
Enter second number > a
11
5>
```

Figure 8.5: Exercising *hex.erl*

8.3 Error Handlers

Erlang defines a number of default behaviors which occur in response to particular types of errors. These behaviors are defined by the *error_handler* module. This module can be replaced by calling **process_flag(error_handler,** *module***)**, where *module* is the name of the module containing functions which implement the interfaces described below.

When an undefined function occurs *error_handler:undefined_function* is invoked:

undefined_function(*module*, *func*, *Args*)

where *module* is the name of the module, *func* is the name of the function and *Args* is the list of arguments.

When an undefined global name occurs *error_handler:undefined_global_name* is invoked:

undefined_global_name(*name*, *message*)

where *name* is the name, and *message* is the message sent to the undefined name.

These functions operate in a complicated and sensitive environment. Changing the default behavior is risky and may lead to system deadlock.

8.4 Defensive Programming

Data provided to an Erlang program can cause a run time failure. Causes for these failures include division by zero or being unable to match a piece of data. Programs need to guard against the problems introduced by bad data. Two strategies are available:

- A *test before use* strategy ensures that data is correctly formatted and meets preconditions before using it.
- A *try and recover if necessary* strategy proceeds as if the data is correct and only worries about error handling if an error occurs.

The former strategy is useful if performing part of an operation without completing is undesirable. The latter strategy is in general more desirable. Error trapping code adds to the complexity and size of the working code, the second strategy reduces the volume of error trapping code and allows it to be separated from the code that handles the general case. If errors tend to be rare events, traversing code to prevent errors adds a cost to each run for an event that occurs infrequently. The second strategy eliminates the test code for the general case.

The programs in figures 8.6 and 8.7 divide 2 integers to produce a dividend. The code in figure 8.6 guards data by testing to see if it is valid before use. A try and recover if necessary strategy is used by the code in figure 8.7.

8.5 Linked Processes

Erlang processes may be linked to other Erlang processes. Links are bidirectional. The linking mechanism is used to notify linked processes of the failure of a process. This notification takes the form of a signal:

'EXIT', *Exiting_PID*, *Reason*

If the *Reason* is not *normal* and a linked process does not handle the exit signal then the linked process will terminate and send exit signals to all its linked processes.

Links can be created when a process is spawned or they can be added or removed after a process has been spawned. The **spawn_link** function creates a process and links the current process to it.

spawn_link(*Module*, *Function*, *Arglist*)

The **spawn_link** function takes the same arguments as **spawn**, after the arguments are evaluated a new process is created and started by calling *Module:Function* with the arguments contained in *Arglist*. The function **link** creates a link between processes:

link(*Pid*)

If a link already exists the **link** function has no effect. If the process does not exist an exit signal of the form {*'EXIT', PID, noproc*} is generated in the process calling link. A link between two processes is removed using the **unlink** function.

unlink(*Pid*)

```
-module(divide2a).
-export([divide/0]).

divide() ->
    io:format("Enter  numbers "),
    {R1, A} = readint(),
    {R2, B} = readint(),
    if
        R1 == ok, R2 == ok ->
            if
                B =/= 0 ->
                    D = A div B,
                    io:format("~w~n", [D]),
                    D;
                true ->
                    io:format("Attempt to divide by zero~n"),
                    divide()
            end;
        true ->
            io:format("Please enter 2 numbers~n"),
            divide()
    end.

readint() ->
    io:format("> "),
    {ok, [L]} = io:fread('', "~s"),
    Len = string:span(L, "0123456789"),
    if
        Len == 0 ->
            {nodata, 0};
        true ->
            V = list_to_integer(string:substr(L,1,Len)),
            {ok, V}
    end.
```

Figure 8.6: Handing Bad Data: *divide2a.erl*

```
-module(divide2b).
-export([divide/0]).

divide() ->
    case (catch getdiv()) of
        {'EXIT', Reason} ->
            io:format("Error ~w~n", [Reason]),
            divide();
        X -> X
    end.

getdiv() ->
    {ok, [A, B]} = io:fread('Enter 2 numbers > ', "~d ~d"),
    D = A div B,
    io:format("~w~n", [D]).
```

Figure 8.7: Handling Bad Data: *divide2b.erl*

This function has no effect if there is no link between the current process and the named process. As links are bidirectional, this call removes both the link from both the caller and the named process.

8.6 Trapping Exits

Two forms of the **exit** are supported in Erlang. An exit executed by a process can be caught by a process using **catch**. Exits from outside a process are caused either by a link or using the two argument form of **exit**. Exits from outside a process cannot be caught by a **catch**. Erlang provides a process flag which allows these external signals to be converted to messages. Executing

process_flag(trap_exit, true)

causes all incoming signals to be converted to messages so that they can be handled.

8.7 Robust Servers

Self healing is a highly desirable property in a long running system. The section describes how a server can be implemented and monitored so that the service can be restarted and hence provide near continuous availability of a service.

Figure 8.8 shows the code for *restart*. This process monitors its children and restarts them when they fail, ensuring service availability. It does not address the issues of preserving state or recovery.

The *restart* process is shown in action with the *mtocon* program. The source for *mtocon* is shown in figure 8.9. The output of the run is shown in figure 8.10.

The code in figures 8.8 and 8.9 employ the following mechanisms and techniques to provide a robust service to their clients:

- Registered Names: Both programs employ registered names to provide easy access to the service. The *mtocon* relies on the use of registered names to provide continuity of reference to the service after it is restarted.
- Defensive Programming: is employed by the *restart* program. Bad data is guarded against when a process is created using **catch** on lines 21 and 57. Failures are ignored by returning an unchanged list. Success results in the list of processes being changed.
- Exit Trapping: is used by the *restart* program to determine when its clients have failed and to initiate an attempt to restart them.

```
-module(restart).
-export([start/0, init/0, add/3, remove/3]).

start() ->
    spawn(restart, init, []).

init() ->
    process_flag(trap_exit, true),
    register(restart, self()),
    manage([]).

add(M, F, A) ->
    restart ! {add, M, F, A}.

remove(M, F, A) ->
    restart ! {remove, M, F, A}.

manage(ListProc) ->
    NewList = receive
        {add, Moda, Funa, Arga} ->
            case (catch newproc(Moda, Funa, Arga, ListProc)) of
                X when list(X) ->
                    X;
                _ ->
                    ListProc
            end;
        {remove, Modr, Funr, Argr} ->
            remproc(Modr, Funr, Argr, ListProc);
        {'EXIT', Pid, Reason} ->
            restart(Pid, ListProc);
        Unknown ->
            ListProc
    end,
    manage(NewList).

newproc(M, F, A, L) ->
    Pid = spawn_link(M, F, A),
    [{M, F, A, Pid} | L].

remproc(M, F, A, L) ->
    remproc(M, F, A, L, []).

remproc(M, F, A, [], L) ->
    L;
remproc(M, F, A, [{HM, HF, HA, HP}|T], L)
        when M==HM, F==HF, A==HA ->
    remproc(M, F, A, T, L);
remproc(M, F, A, [H|T], L) ->
    remproc(M, F, A, T, [H|L]).

restart(P, L) ->
    restart(P, L, []).
```

```

restart(P, [], L) ->
    L;
restart(P, [{HM, HF, HA, HP}|T], L) when P==HP ->
    NL = case (catch newproc(HM, HF, HA, T)) of
        X when list(X) ->
            X;
        _ ->
            T
    end,
    lists:append(NL, L);
restart(P, [H|T], L) ->
    restart(P, T, [H|L]).
```

Figure 8.8: A Program to Restart Processes: *restart.erl*

```
-module(mtocon).
-export([start/1]).

start(N) ->
    register(N, self()),
    io:format("Starting ~w~n", [N]),
    loop(N).

loop(N) ->
    receive
        {exit, Reason} ->
            io:format("Exiting ~w because ~w~n", [N, Reason]),
            exit(Reason);
        X ->
            io:format("~w received ~w", [N, X])
    end,
    loop(N).
```

Figure 8.9: A Program That Outputs its Messages: *mtocon.erl*

8.8 Generic Servers

The *gen_server* module provides the basic services required to implement a server. The module uses callback functions to provide the specific behavior required for the service. A callback is achieved by passing the module and function names to the generic server. The use of generic servers is encouraged as it allows the programmer to focus on writing the sequential parts of the service, and eliminates the need to test the shared server component each time a server is written.

```
% erl
Erlang (BEAM) emulator version 4.6.4

Eshell V4.6.4  (abort with ^G)
1> restart:start().
<0.28.0>
2> restart:add(mtocon,start,[one]).
{add,mtocon,start,[one]}
Starting one
3> restart:add(mtocon,start,[two]).
{add,mtocon,start,[two]}
Starting two
4> one ! hi.
one received hihi
5> two ! hi.
two received hihi
6> one ! {exit, normal}.
Exiting one because normal
{exit,normal}
Starting one
7> one ! hi.
one received hihi
8> restart:remove(mtocon,start,[one]).
{remove,mtocon,start,[one]}
9> one ! hi.
one received hihi
10> one ! {exit, done}.
Exiting one because done
{exit,done}
11> one ! hi.

=ERROR REPORT==== 10-Jun-1998::10:29:17 ===
<0.21.0> error in BIF send/2(one,hi)
<0.21.0> error: badarg in erl_eval:eval_op/3
** exited: {badarg,{erl_eval,eval_op,3}} **
12> two ! hi.
two received hihi
13> restart:add(mtocon,stop,[one]).
{add,mtocon,stop,[one]}
14> restart:add(mtocon,start,[three]).
{add,mtocon,start,[three]}
Starting three
15> three ! hi.
three received hihi
16> restart:remove(mtocon,start,[two]).
{remove,mtocon,start,[two]}
17> restart:remove(mtocon,start,[three]).
{remove,mtocon,start,[three]}
18> three ! {exit, done}.
Exiting three because done
{exit,done}
19>
```

Figure 8.10: Exercising *restart.erl* with *mtocon.erl*

8.9 Resources

The code files mentioned in this chapter are:

div.C
divide.erl
hex.erl
divide2a.erl
divide2b.erl
restart.erl
mtocon.erl

These files can be retrieved from:

`http://www.serc.rmit.edu.au/~maurice/erlbk/eg/robust`.

Further information on the *gen_server* module can be found in the 'The Standard Erlang Libraries: Reference Manual' in 'Open Telecom Platform (OTP)' documentation set by Ericsson Software Technology AB, Erlang Systems. This documentation is provided in HTML and Postscript form with the Erlang distribution.

8.10 Exercises

1. Identify some of the faults which *restart* (figure 8.8) does not address. Suggest alterations to the code that would allow *restart* to address these faults.

2. The Roman satirist Juvenal reflected 'Quis custodiet ipsos custodes?' which may be translated as 'Who is to guard the guards themselves?' Describe the problems likely to be encountered in building a robust server and identify a strategy for constructing a robust server.

Chapter 9

Code Replacement

Some commercial systems run for many years at a time with no opportunity for downtime. Telephone exchanges and power grids are examples of systems where a design objective is to have one hundred percent availability. Based on experiences drawn from other commercial computing endeavors, it would be unreasonable to expect that code used in these systems would be correct in all aspects and not require alteration during the life of the system. The normal mode of code replacement, bringing down the system and restarting it with new code is unacceptable for this type of system. Erlang was designed with telephone exchanges as an application, and provides a mechanism for replacing parts of the program code while the system remains running.

This chapter discusses the mechanisms that Erlang provides for code loading and replacement.

9.1 Loading and Linking

Erlang groups functions into collections called modules. Loading is the process of reading a module into the systems memory for use. Linking is the process of resolving names to addresses. Linking can be carried out once when a program is compiled (static linking) or it can be deferred until a program is running (dynamic linking).

Modules are loaded into memory when a function in that module is first named.

Erlang dynamically links a module when a fully qualified function name is used for a function contained in that module. A fully qualified name consists of the module name and the function name separated by a colon. Each time a fully qualified function name is encountered the code transfers execution to the latest instance of the module loaded.

Functions named in a module, that are referred to only by the function name (not fully qualified) are statically linked at compile time. Static linkages cannot be altered at run time.

9.2 Code Replacement

Modules are the base unit of code replacement. More than 1 image of a module may be present in memory at one time (current systems support a maximum of 2 images). When a fully qualified call is made the function named is executed from the latest version of the module currently in memory. Partially qualified references, whether internal (static) or from an import clause refer to the version of the module present when the fully qualified call was made.

An artificial example of code replacement is shown in figures 9.1 and 9.2. The code in figure 9.1 is modified half way through the output shown in figure 9.2 changing the version number from *1* to *2*.

```
-module(coderep).
-export([msglp/0]).

vers() ->
        1.

msglp() ->
        Msg = receive
                X -> X
        end,
        io:format("~w ~w~n", [Msg, vers()]),
        coderep:msglp().
```

Figure 9.1: Source code for *coderep* (initial version): *coderep.erl*

In the example a new version of the code is created at the 6th step in figure 9.2 by recompiling the altered module. The new code is first used at the end of the 7th step when the fully qualified function call *coderep:msglp* is executed. When that statement is executed the new module is loaded into memory and control is transfered to it. All references to the function *vers* are statically linked at compile time. The result of the code change is seen at the 8th step when the version number is shown as *2* .

9.3 Limitations

In section 9.2 it was noted that only 2 images of a module are supported concurrently. If a process requires an old image and it is moved out of memory, the process can no longer run. Figure 9.3 illustrates a process that uses old code and figure 9.4 shows how it fails.

When the process attempts to access the old function *stale* in the original module, it fails as the code has been displaced by 2 newer instances of the module.

9.4 Code Management

Erlang provides a set of functions to manage module images and the processes running old images. These functions can be used to avoid the problem demon-

```
% erl
Erlang (BEAM) emulator version 4.6.4

Eshell V4.6.4  (abort with ^G)
1> c(coderep).
{ok,coderep}
2> Pid=spawn(coderep,msglp,[]).
<0.32.0>
3> Pid ! 1.
1 1
1
4> Pid ! 2.
2 1
2
5> Pid ! 3.
3 1
3
6> c(coderep).
{ok,coderep}
7> Pid ! 1.
1 1
1
8> Pid ! 2.
2 2
2
9> Pid ! 3.
3 2
3
10>
```

Figure 9.2: Demonstrating Code Replacement

strated in section 9.3. The interface to the management functions is found in the *code* module. The functions found in this module include:

The **load_file** function attempts to load the named Erlang module. If the module loaded replaces an existing module the existing module is made old and any other copies of the module are removed from memory.

load_file(Module)

The **delete** function makes the code for *Module* old. New invocations of the module will not be able to access the deleted module. It returns *true* on success, and *false* on failure.

delete(Module)

The **purge** function removes the code of the named module marked as old from the system. Processes running the old module code will be killed. If a process has been killed the function returns *true*, otherwise *false*.

purge(Module)

```
-module(stale).
-export([start/0, stale/0, vers/0]).

vers() ->
    1.

start() ->
    spawn(stale, stale, []).

stale() ->
    receive
        startver ->
            io:format("Start Version ~w~n", [vers()]);
        curver ->
            io:format("Current Version ~w~n", [stale:vers()])
    end,
    stale().
```

Figure 9.3: Source for *stale*: *stale.erl*

The **soft_purge** function is similar to **purge** except it will not purge a module if it is currently been used by a module. If a process is using old code the function returns *false*, otherwise *true*.

soft_purge(Module)

The **is_loaded** function returns a tuple containing the atom *file* and either the filename from which the module was loaded or the atoms *preloaded* or *interpreted* for loaded modules. If the module is not loaded then it returns *false*.

is_loaded(Module)

The **all_loaded** function returns a list of tuples containing the name of the module, and either the filename from which the module was loaded or the atoms *preloaded* or *interpreted* for all loaded modules.

all_loaded()

Using these functions and another mechanism of invoking the compiler it is possible to write a program that changes its code when required. The program in figure 9.5 uses a new version of itself only after a *new* message is sent to it. Figures 9.6 shows that code can be changed and compiled but does not become active until a message is sent. The *compile:file* function is similar to *c* but it does not automatically load the compiled module.

```
% erl
Erlang (BEAM) emulator version 4.6.4

Eshell V4.6.4  (abort with ^G)
1> c(stale).
{ok,stale}
2> P=stale:start().
<0.32.0>
3> P ! startver.
Start Version 1
startver
4> P ! curver.
Current Version 1
curver
5> c(stale).
{ok,stale}
6> P ! startver.
Start Version 1
startver
7>  P ! curver.
Current Version 2
curver
8> c(stale).
{ok,stale}
9> P ! startver.
startver
10>
```

Figure 9.4: Process Failure Caused by Code Replacement

9.5 Resources

The code files mentioned in this chapter are:

coderep.erl
stale.erl
nofstale.erl

These files can be retrieved from:

`http://www.serc.rmit.edu.au/~maurice/erlbk/eg/coderep`.

9.6 Exercises

1. Write some programs that use the functions described in this chapter to explore code replacement.

```
-module(nofstale).
-export([start/0, nofailstale/0, vers/0]).

vers() ->
    3.

start() ->
    spawn(nofstale, nofailstale, []).

nofailstale() ->
    Msg = receive
        X -> X
    end,
    nofailstale(Msg).

nofailstale(new) ->
    code:purge(nofstale),
    code:load_file(nofstale),
    nofstale:nofailstale();
nofailstale(ver) ->
    io:format("Start Version ~w~n", [vers()]),
    nofailstale().
```

Figure 9.5: Source for *nofailstale*: *nofstale.erl*

```
% erl
Erlang (BEAM) emulator version 4.6.4

Eshell V4.6.4  (abort with ^G)
1> compile:file(nofstale).
Erlang BEAM Compiler 4.6.4/22
Object file: nofstale.beam
{ok,nofstale}
2> P=nofstale:start().
<0.32.0>
3> P!ver.
Start Version 1
ver
4> P!ver.
Start Version 1
ver
5> compile:file(nofstale).
Erlang BEAM Compiler 4.6.4/22
Object file: nofstale.beam
{ok,nofstale}
6> P!ver.
Start Version 1
ver
7> P!new.
new
8> P!ver.
Start Version 2
ver
9> P!ver.
Start Version 2
ver
10> compile:file(nofstale).
Erlang BEAM Compiler 4.6.4/22
Object file: nofstale.beam
{ok,nofstale}
11> P!new.
new
12> P!ver.
Start Version 3
ver
13> P!ver.
Start Version 3
ver
14>
```

Figure 9.6: *nofailstale* in Action

Chapter 10

Programming Style

This chapter provides an introduction to good programming style and formatting in Erlang. Readers are referred to the reference section (section 10.6) for a more comprehensive guide.

Many of the suggestions made in this chapter will be for particular conventions. Conventions are agreed means for handling specified circumstances. They need not be enforced and may be arbitrary in nature. The general use of conventions in programming is to improve the readability and understandability of code. Conventions can be used to provide additional information about the program to programming tools that is not present in the compiled code. This information often relates to the intentions of the authors or the anticipated use of the code.

10.1 Comments and Documentation

Erlang provides two mechanisms for internal documentation: comments and attributes . Figure 10.1 illustrates the documentation conventions and implements a semaphore.

10.1.1 Comments

Comments are introduced using a '%' in Erlang. A commenting convention describes how and where comments should be used in source code. In general, a compiler does not enforce a convention, however, tools can be written that check a convention is at least being partly met. Furthermore, a comment convention can be exploited to assist in the automatic generation of documentation.

The following is the recommended convention for comments in Erlang:

- Module descriptions should begin with 3 percent signs ('%%%')
- Function descriptions should begin with 2 percent signs ('%%')
- Comments within a function should begin with 1 percent sign ('%'). The preferred practice is to place the comment at the end of the line that is being commented on. If a comment does not fit it should be placed on the line above.

```
-module(doc).
-author('Maurice Castro').
-copyright('Copyright (c) 1998').
-vsn('$Id: doc.erl,v 1.4 1998/06/22 02:33:07 maurice Exp $').
-modified('Mon Jun 22 08:38:16 EST 1998').
-modified_by('maurice@serc').
-modified('Mon Jun 22 12:17:35 EST 1998').
-modified_by('maurice@serc').

-export([start/0, start/1, p/1, v/1, msglp/2]).

%%% ---------------------------------------------------------------
%%% This module illustrates documentation conventions described in
%%% 'Erlang in Real Time' by Maurice Castro
%%% It implements a Counting Semaphore as defined by Hwang K,
%%% 'Advanced Computer Architecure' McGraw Hill, 1993, p638.
%%% Warning this code assumes no messages are lost.
%%% ---------------------------------------------------------------

%% ----------------------------------------------------------------
%% This is the special case of a binary semaphore, use general
%% start routine to perform start.
%% ----------------------------------------------------------------

start() ->
        start(1).

%% ----------------------------------------------------------------
%% The start/1 function starts a server and returns its pid.
%% This is the general case
%% ----------------------------------------------------------------

start(N) ->
        % spawn the message loop with initial data of N
        spawn(doc, msglp, [N, []]).


%% ----------------------------------------------------------------
%% P(s): if s > 0, then s = s - 1; else put in wait queue
%% ----------------------------------------------------------------

p(S) ->
        S ! {semaphore, p, self()},     % cont requires process name
        % wait for a continue message to indicate exiting queue
        receive
                {semaphore, cont} ->
                        true
        end.

%% ----------------------------------------------------------------
%% V(s): if wait queue not empty, wake up one; else s = s + 1
%% ----------------------------------------------------------------
```

```

v(S) ->
        S ! {semaphore, v}.     % Server handles v

%% -------------------------------------------------------------
%% The msglp function handles   cases:
%% -------------------------------------------------------------

msglp(S, L) ->
        {NewS, NewL} = receive
                {semaphore, p, Pid} ->
                        if
                                % P(s): if s > 0, then s = s - 1;
                                S > 0 ->
                                        Pid ! {semaphore, cont},
                                        {S - 1, L};
                                true ->
                                        % else put in wait queue
                                        {S, [Pid, L]}
                        end;
                % V(s): if wait queue not empty,
                {semaphore, v} when length(L) =/= 0 ->
                        [H|T] = L,
                        % wake up one;
                        H ! {semaphore, cont},
                        {S, T};
                % if the list is empty on a v
                {semaphore, v} ->
                        % else s = s + 1
                        {S+1, L}
        end,
        msglp(NewS, NewL).
```

Figure 10.1: Documentation Example - Semaphore Source Code: *doc*

10.1.2 Attributes

A module attribute appears at the beginning of an Erlang module, before any Erlang code. An attribute consists of a '-' a name and a bracketed Erlang term.

-*name*(*term*).

A number of module attributes have special meanings such as **-module()**, **-export()**, and **-import()**. Other attributes can be used for documentation purposes such as identifying the creator and those who have influenced code.

10.2 Modules

This section provides a series of recommendations that apply to Erlang modules.

- **Minimise the number of functions exported from a module.** Reduces the possibility for coupling between modules to a small number of functions. This minimises the number of interface functions that need to be maintained.

- **Use functions to encapsulate common code.** Avoid cut and paste programming, instead write a function to represent the repeated aspect of the code.

- **Do not presume that the data structures provided by a module are unchanging.** A module should provide a sufficient interface to carry out all required operations on the data structures it generates. If a user knows the structures generated by a module they should avoid access the elements of those structures directly.

- **Use interface functions.** Use functions as an interface where possible. Avoid sending messages directly.

10.3 Functions

This section provides a series of recommendations that apply to Erlang functions.

- **Avoid side effects.**

- **Do not assume what the user of a function wants to do with its results.** The act of printing an error message when an error occurs in a function assumes that the user wants an error displayed. If the error were returned silently then the user of the function could choose to display the error or not.

10.4 Messages

This section provides a series of recommendations that apply to Erlang messages.

- **Tag messages.** This reduces the sensitivity of messages to order.

- **Dispose of unknown messages.** Every server should incorporate a match all pattern in at least one of its receives to ensure that unhandleable messages are removed from the message queue.

10.5 General

This section provides a series of general recommendations.

- **Avoid defensive programming.** The majority of a systems programs should trust that their inputs are correct. Exceptions can be caught where necessary. Only a small fraction of the code in a system should check its inputs.

- **Separate handling of error cases from normal code.**

- **Write declaritive code.** Use guards in function heads where possible to declare the applicability of the code rather than hiding the choices in **if** and **case** operations.

10.6 Resources

The code files mentioned in this chapter are:

doc.erl

These files can be retrieved from:

`http://www.serc.rmit.edu.au/~maurice/erlbk/eg/style`.

Further information on programming style can be found in Eriksson, K., Williams, M., Armstrong, J., 'Program Development Using Erlang - Programming Rules and Conventions', 1995, `http://www.erlang.se/erlang/sure/main/news/programming_rules.ps.gz` or `http://www.erlang.se/erlang/sure/main/news/programming_rules.shtml`.

10.7 Exercises

1. Study the style guidelines and identify a rationale for each guideline.

2. Rewrite *filecnt* (figure 1.5) using the style recommendations made in this chapter.

3. Examine the examples in this book and recommend style improvements.

Chapter 11

Graphics

Although Erlang supports at least 2 graphics subsystems (gs and pxw) this chapter will limit itself to describing the portable graphics subsystem known as *gs*. This subsystem was used in the previous examples of graphical code (figures 1.3 and 2.9).

11.1 Model

The gs subsystem is built on an event model. Messages are sent between the controlling process and a gs server to cause or report events. Figure 11.1 shows the relationship between the components of the system, the notification of an event and an operation being performed.

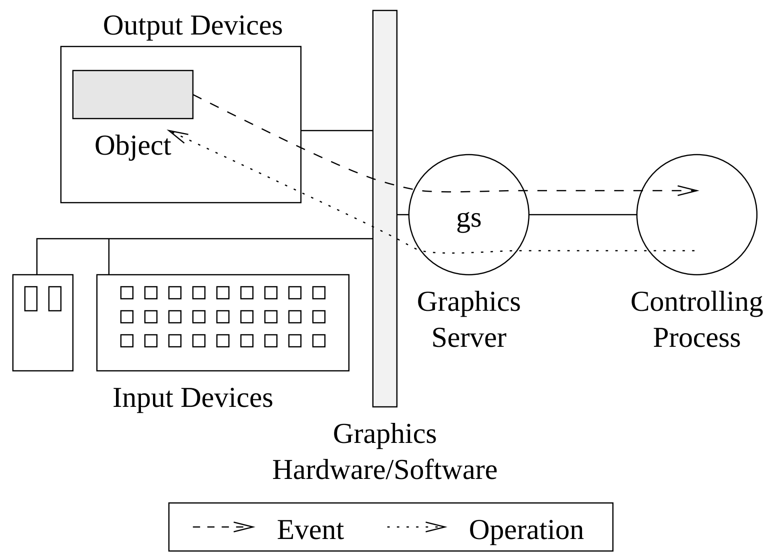


Figure 11.1: The Components of the GS Graphical Model

Objects are created in a hierarchy. Each object has a parent and may have

```
1  -module(thrbut).
2  -export([init/0]).
3
4  init() ->
5      Server = gs:start(),
6      Win = gs:create(window, Server, [{width, 300}, {height, 80}]),
7      Display = gs:create(label, Win, [{label, {text, "0"}},
8          {x, 0}, {y, 0}, {width, 200}, {height, 50}]),
9      Plus = gs:create(button, Win, [{label, {text, "+"}},
10         {x, 0}, {y, 50}]),
11     Minus = gs:create(button, Win, [{label, {text, "-"}},
12         {x, 100}, {y, 50}]),
13     Quit = gs:create(button, Win, [{label, {image, "q.xbm"}},
14         {x, 200}, {y, 50}]),
15     gs:config(Win, {map, true}),
16     event_loop(0, Display, Plus, Minus, Quit).
17
18 event_loop(N, D, P, M, Q) ->
19     receive
20         {gs, P, click, Data, Args} ->
21             RP = N+1,
22             gs:config(D, {label, {text, RP}}),
23             event_loop(RP, D, P, M, Q);
24         {gs, M, click, Data, Args} ->
25             RM = N-1,
26             gs:config(D, {label, {text, RM}}),
27             event_loop(RM, D, P, M, Q);
28         {gs, Q, click, Data, Args} ->
29             gs:stop(),
30             N
31     end.
```

Figure 11.2: Source Code for *thrbut.erl*

one or more children. When an object is created an object identifier is returned to the creator. Objects can be created with a specified name, allowing the program to control the choice of identifier.

Figures 11.2 and 11.3 show a simple example of a server with 3 buttons and a text display. This example illustrates:

- Starting and stopping the graphics server (lines 5 and 29) using **gs:start()** and **gs:stop()**
- Creating a window (line 6) using **gs:create**
- Creating buttons inside a window (lines 9 to 14)
- Creating a label inside a window (lines 7 and 8)
- Changing the options connected with a graphical object (lines 15, 22 and 26) using **gs:config**
- Handling events generated in an even loop (lines 18 to 31)

Figure 11.3: Display from *thrbut.erl*

The example has a simple two level hierarchy which is shown in figure 11.4 (Note: the actual names of the entities are not shown in the diagram, only the variables in which the names of the graphical objects were initially stored). The hierarchy is constructed by naming the parents when the **gs:create** function is called. The hierarchy is used to describe the layout relationships between objects, for instance it allows objects to be grouped so that they can moved as part of a single entity.

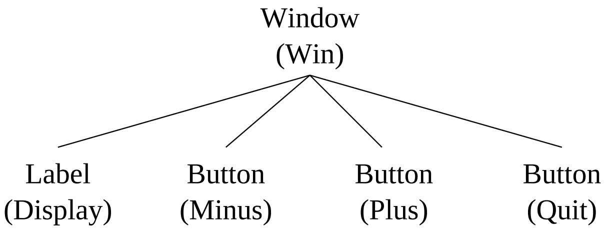


Figure 11.4: The GS Hierarchy Found in *thrbut.erl*

11.2 Interface

11.2.1 Functions

The interface to gs is built on 6 basic functions: **gs:start**, **gs:stop**, **gs:create**, **gs:config**, **gs:destroy**, and **gs:read**.

The **gs:start** function starts the gs server. The function takes no arguments. An identifier is returned which is used as the parent for top level objects (windows). If the function is called more than once it returns the same identifier.

gs:start()

The **gs:stop** function stops the gs server and closes any windows opened by the server. The function takes no arguments.

gs:stop()

The **gs:create** function is used to make graphical objects. The function has several forms. The types of the arguments used in the forms are: *Objtype* – atom identifying object type; *Parent* – identifier returned by parent; *Options* – list of options to be set for object; *Option* – option to be set for object; and *Name* – identifier to be used to refer to object. An identifier for the new object is returned.

> **gs:create**(*Objtype*, *Parent*)
> **gs:create**(*Objtype*, *Parent*, *Options*)
> **gs:create**(*Objtype*, *Parent*, *Option*)
> **gs:create**(*Objtype*, *Name*, *Parent*, *Options*)
> **gs:create**(*Objtype*, *Name*, *Parent*, *Option*)

The **gs:config** function sets an option for an object. The function has two forms. It takes an object identifier or a name and either a single option value tuple or a list of option value tuples as arguments. It returns *ok* on success or {*error, Reason*} on failure.

> **gs:config**(*Identifier*, {*Option, Value*})
> **gs:config**(*Identifier*, [{*Option, Value*} ...])

The **gs:destroy** function destroys a graphical object and its children. The function takes an object identifier or a name as an argument.

> **gs:destroy**(*Identifier*)

The **gs:read** function reads a value from a graphical object. The function takes an object identifier or a name and a key as arguments. It returns the value read on success or {*error, Reason*} on failure.

> **gs:read**(*Identifier*, *Key*)

11.2.2 Objects

The gs subsystem provides a large number of built in graphics objects which can be used to construct displays. A selection of the available objects is discussed below.

A **window** is a screen object which contains other screen objects. Only windows are allowed to be top level objects. All other objects must be descendents of a top level object. A window may have a window or the server as its parent. The atom *window* is used to denote this object type.

A family of button objects is supported. A **button** is an object which may be selected with a mouse. It may be selected or unselected. A button may have a window or a frame as a parent. The atom *button* is used to denote a simple button, *radiobutton* is used to denote a button type where only one member of a group of buttons may be pressed at one time, and *checkbutton* denotes a button type where many buttons may be selected in a group at one time.

A **label** is used to display either a text message or a bitmap[1]. A label may have a window or a frame as a parent. The atom *label* is used to denote this object type.

A **frame** is a container used to group objects. A frame may have a window or a frame as a parent. The atom *frame* is used to denote this object type.

An **entry** allows a single line of text to be entered. An entry may have a window or a frame as a parent. The atom *entry* is used to denote this object type.

A **listbox** displays a list of strings and allows zero or more to be selected. A listbox may have a window or a frame as a parent. The atom *listbox* is used to denote this object type.

A **canvas** is a drawing area. The following objects may be present in a canvas: *arc*, *image*[2], *line*, *oval*, *polygon*, *rectangle*, and *text*. A canvas may have a window or a frame as a parent. The atom *canvas* is used to denote this object type.

A collection of elements have been provided that can be used to construct menus. A **menu** is a recursive graphical structure which is used to present options and actions. These choices can be selected. A menu may have as a parent: a menubutton, a menuitem with an itemtype of cascade, a window or a frame. A menuitem may have a menu as a parent. A menubar may have a frame or a window as a parent. The atoms *menu*, *menuitem*, *menubutton* and *menubar* are used to denote objects used to construct menus.

The subsystem also provides facilities to construct tables, a multi-line editor and to select values from a scale (the interface resembles a sliding potentiometer).

11.2.3 Events

Events are represented by tuples which are sent as messages to the controlling process. These messages have the form:

$$\{gs,\ Identifier,\ EventType,\ Data,\ Args\}$$

The atom *gs* is used to tag gs related messages. The *Identifier* field contains either an object identifier returned by create or a name (objects created with the name form of create use names here). The *EventType* defines the class of event that has occurred. The *Data* field is used to return user defined data associated with an object that is generating an event. The *Args* field contains event specific information.

All objects return the following events (generic events):

A *buttonpress* is generated when a mouse button is pressed over an object. A *buttonrelease* is generated when a mouse button is released over an object. They both return in *Args*:

[ButtonNo, X, Y | _]

An *enter* is generated when the mouse pointer enters an object area. A *leave* is generated when the mouse pointer leaves an object. A list is returned in *Args* for both these events.

[1] At the time of writing only monochrome X11 bitmaps were supported

[2] At the time of writing GIFs and BMP image files were supported

A *focus* event is generated when the keyboard focus changes. The *Int* in the structure shown below is *1* if the focus has been gained and *0* if the focus has been lost.

[Int | _]

A *keypress* event is generated for each keypress. The *Args* field contains:

[KeySym, KeyCode, Shift, Control | _]

where *KeySym* contains an atom describing the key pressed, *KeyCode* contains the key number for the depressed key and *Shift* and *Control* contain *1* if the modifier keys are depressed and *0* otherwise.

A *motion* event is generated when the mouse moves inside an object. The *Args* field contains:

[X, Y | _]

There are two object specific events: *click* and *double-click*. The argument lists returned for these events depend on the type of object that was clicked on.

11.3 Example

The program *chboard.erl* draws a draughts / checkers board and allows moves to be made. Button 1 on the mouse is used to move pieces, button 2 kings pieces, and button 3 deletes pieces. This example illustrates rubber banding (in movement mode), the use of data elements associated with graphical items, menus, and the event driven nature of the interface. Sample output is shown in figure 11.5 and the code is shown in figure 11.6.

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

start() ->
  Server = gs:start(),
  Win = gs:create(window, Server, [{width, 8 * sx()},
    {height, 100 + 8 * sy()}]),
  menu(Win),
  Canvas = newgame(Win),
  gs:config(Win, {map, true}),
  event_loop(Win, Canvas, normal, {}).

event_loop(Win, Canvas, Mode, StateData) ->
  receive
    {gs, exit, click, _, _} ->
      gs:stop(),
      exit(normal);
    {gs, new, click, _, _} ->
      gs:destroy(Canvas),
      event_loop(Win, newgame(Win), normal, {});
    {gs, Id, buttonpress, {X, Y, Base, Color, man, I},
      [ 3, _, _ | _ ]} when Mode == normal ->
      gs:destroy(I),
      gs:config(Id, [{data, {X, Y, Base, empty, empty, noimage}}]),
      event_loop(Win, Canvas, Mode, StateData);
    {gs, Id, buttonpress, {X, Y, Base, Color, man, I},
      [ 2, _, _ | _ ]} when Mode == normal ->
      gs:destroy(I),
      piece(Id, X, Y, Base, Color, king),
      event_loop(Win, Canvas, Mode, StateData);
    {gs, Id, buttonpress, {X, Y, Base, Color, Piece, I},
      [ 1, _, _ | _ ]} when Mode == normal, Piece =/= empty ->
      gs:destroy(I),
      gs:config(Id, [{data, {X, Y, Base, empty, empty, noimage}}]),
      event_loop(Win, Canvas, move, {Id, X, Y, Base, Color, Piece});
    {gs, Id, enter, _, _ } when Mode == move ->
      gs:config(Id, {bg, red}),
      event_loop(Win, Canvas, move, StateData);
    {gs, Id, leave, {X, Y, Base, Color, Piece, I},
      _ } when Mode == move ->
      gs:config(Id, {bg, Base}),
      event_loop(Win, Canvas, move, StateData);
    {gs, Id, buttonpress, {X, Y, Base, Color, Piece, I},
      [ 1, _, _ | _ ]} when Mode == move, Piece == empty ->
      {OId, _, _, B, C, M} = StateData,
      gs:config(Id, [{bg, Base}]),
      piece(Id, X, Y, Base, C, M),
      event_loop(Win, Canvas, normal, {});
    X ->
      event_loop(Win, Canvas, Mode, StateData)
  end.
```

```
menu(Win) ->
  MenuBar = gs:create(menubar, Win, []),
  FileBut = gs:create(menubutton, MenuBar, [{label,
    {text, "File"}}]),
  FileMenu = gs:create(menu, FileBut, []),
  gs:create(menuitem, new, FileMenu, [{label, {text, "New"}}]),
  gs:create(menuitem, exit, FileMenu, [{label, {text, "Exit"}}]).

newgame(Win) ->
  Frame = gs:create(frame, Win, [{width, 8 * sx()},
    {height, 8 * sy()}, {bw, 1}, {x, 0}, {y, 100}]),
  drawch(Frame, 8, 8),
  setboard(Frame),
  Frame.

setboard(Frame) ->
  StartPos = [
  {1, 0, white, man}, {3, 0, white, man}, {5, 0, white, man},
  {7, 0, white, man}, {0, 1, white, man}, {2, 1, white, man},
  {4, 1, white, man}, {6, 1, white, man}, {1, 2, white, man},
  {3, 2, white, man}, {5, 2, white, man}, {7, 2, white, man},
  {0, 5, black, man}, {2, 5, black, man}, {4, 5, black, man},
  {6, 5, black, man}, {1, 6, black, man}, {3, 6, black, man},
  {5, 6, black, man}, {7, 6, black, man}, {0, 7, black, man},
  {2, 7, black, man}, {4, 7, black, man}, {6, 7, black, man}
  ],
  SqLst = gs:read(Frame, children),
  setboard(SqLst, StartPos).

setboard(SqLst, []) ->
  true;
setboard([Sq | T], Pieces) ->
  SqD = gs:read(Sq, data),
  setboard(T, setsq(Sq, SqD, Pieces, [])).

setsq(Sq, SqD, [], L) ->
  L;
setsq(Sq, {SqX, SqY, SqB, SqC, SqM, I}, [{X, Y, C, M} | T], L) ->
  if
    SqX == X, SqY == Y ->
      piece(Sq, SqX, SqY, SqB, C, M),
      lists:append(T, L);
    true ->
      setsq(Sq, {SqX, SqY, SqB, SqC, SqM, I}, T,
        [{X, Y, C, M} | L])
  end.

drawch(Frame, Nx, Ny) ->
  drawch(Frame, Nx, Ny, Nx-1, Ny-1, white).

drawch(F, Nx, Ny, 0, 0, BW) ->
  sq(F, 0, 0, BW);
drawch(F, Nx, Ny, Px, 0, BW) ->
  sq(F, Px, 0, BW),
```

```
  drawch(F, Nx, Ny, Px-1, Ny-1, BW);
drawch(F, Nx, Ny, Px, Py, BW) ->
  sq(F, Px, Py, BW),
  drawch(F, Nx, Ny, Px, Py-1, oppcolor(BW)).

sq(F, X, Y, Color) ->
  Xs = sx(),
  Ys = sy(),
  gs:create(canvas, F, [{x, X * Xs}, {y, Y * Ys},
    {width, Xs}, {height, Ys}, {bg, Color},
    {enter, true}, {leave, true}, {buttonpress, true},
    {data, {X, Y, Color, empty, empty, noimage}}]).

piece(Canvas, X, Y, Base, Color, Piece) ->
  File = case {Color, Piece} of
    {white, man} -> "whtbit.gif";
    {black, man} -> "blkbit.gif";
    {white, king} -> "whtking.gif";
    {black, king} -> "blkking.gif"
  end,
  I = gs:create(image, Canvas, [{load_gif, File}]),
  gs:config(Canvas, {data, {X, Y, Base, Color, Piece, I}}).

oppcolor(white) ->
  black;
oppcolor(black) ->
  white.

sx() -> 50.

sy() -> 50.
```

Figure 11.6: *Checker Board* Source Code (*chboard.erl*)

11.4 Resources

The code files mentioned in this chapter are:

thrbut.erl
q.xbm
chboard.erl
blkbit.gif
blkking.gif
whtbit.gif
whtking.gif

These files can be retrieved from:

`http://www.serc.rmit.edu.au/~maurice/erlbk/eg/graph`.

Further information on the *gs* module can be found in the 'The Graphics System (GS): GS User's Guide' in 'Open Telecom Platform (OTP)' documentation set by Ericsson Software Technology AB, Erlang Systems. This documentation is provided in HTML and Postscript form with the Erlang distribution.

11.5 Exercises

1. Add comments to the *Checker Board* source code (*chboard.erl*) from figure 11.6. Describe features and functions provided by the code.

2. Modify the *event_loop* in *Checker Board* so that it can be easily extended and is more readable. Pay particular attention to making it possible to add new features. Suggest methods for checking the correctness of moves.

3. Modify *Checker Board* to allow two players on different Erlang nodes to play against each other.

4. Make *Wobbly Invaders* (figure 2.9) into a playable game. Only one invader and one defender is required. Consider using a process for each graphical object.

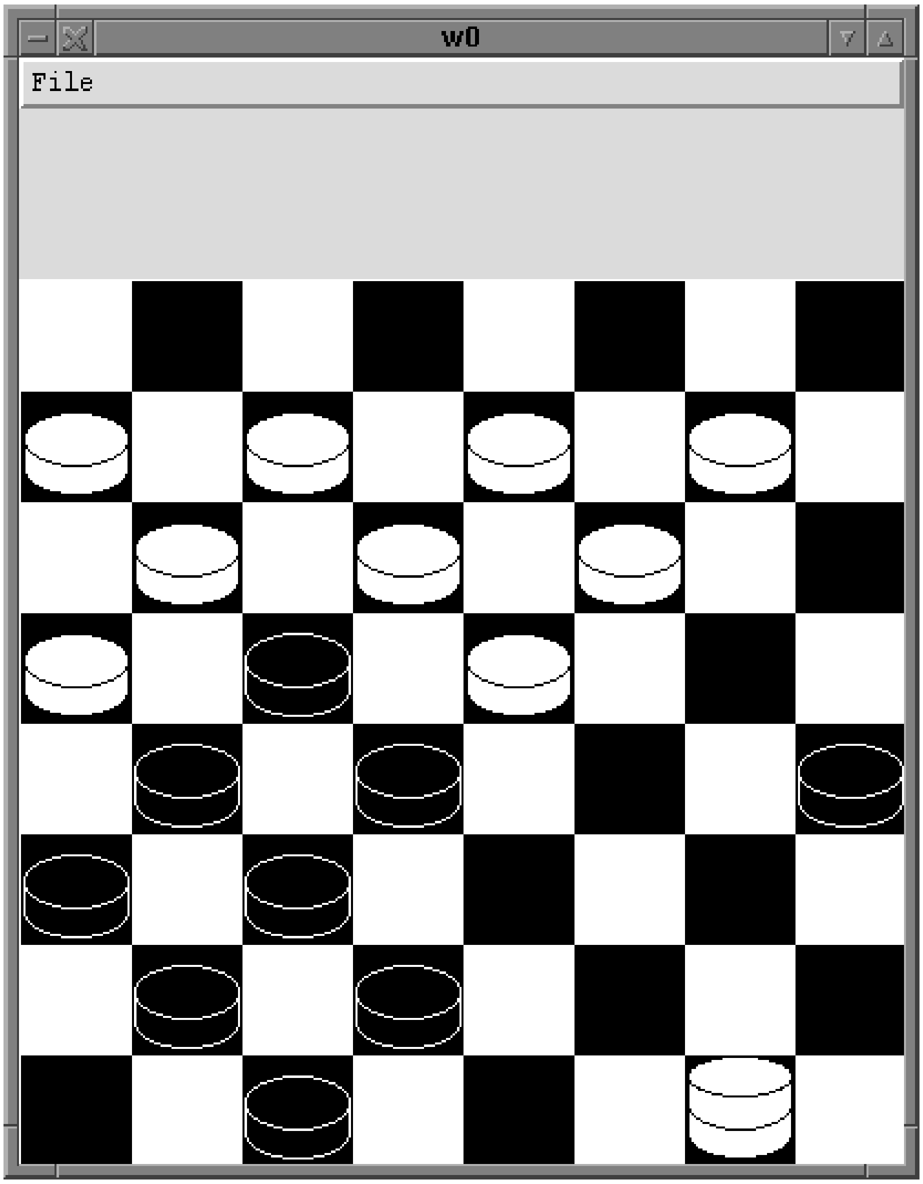


Figure 11.5: A game in play on the *Checker Board*

Chapter 12

Internet

Internet based applications and interfaces are rapidly becoming de rigueur for systems. Many designers choose to use a TCP/IP based interface because of: the simplicity and ubiquity of the WWW interface; the wide availability of the TCP/IP protocol; and the high degree of interoperability between systems provided by the protocol.

The Erlang support libraries provide easy access to TCP/IP functionality, allowing Erlang programmers to implement both clients and servers easily. This chapter describes one of the library interfaces used to access TCP/IP sockets, the *gen_tcp* module. Similar functionality for UDP or datagram sockets is provided by the *gen_udp* module.

12.1 Basic Functions

The *gen_tcp* module provides many functions including: **accept**, **close**, **connect**, **listen**, **recv**, and **send**. The named functions are sufficient to setup both client and server programs. The functions are described below:

accept accepts an incoming connection request on a listen socket.

close closes an open socket.

connect makes a TCP/IP connection to a specified server on a specified port.

listen sets up a listen socket to which clients can connect.

recv receives a packet.

send transmits a packet.

12.2 A Simple Web Server

WARNING: This web server described in this section is **not secure**. It performs no checking on file names and hence can be exploited by third parties to view your files.

Figure 12.1 illustrates a simple WWW server written in Java. In this section this program will be rewritten in Erlang.

```
import java.net.*; import java.io.*; import java.util.*;

// Based on TinyHttpd from Niemeyer P, Peck J, Exploring Java,
// O'Reilly & Associates, 1996, pp 244-245

public class httpd
{
    public static void main(String argv[]) throws IOException
    {
        ServerSocket svrsock =
            new ServerSocket(Integer.parseInt(argv[0]));
        while (true)
        {
            Socket consock = svrsock.accept();
            new httpdconnection(consock);
        }
    }
}

class httpdconnection extends Thread
{
    Socket sock;

    public httpdconnection(Socket s)
    {
        sock = s;
        setPriority(NORM_PRIORITY - 1);
        start();
    }

    public void run()
    {
        try
        {
            OutputStream out = sock.getOutputStream();
            PrintWriter outw =
                new PrintWriter(sock.getOutputStream());
            InputStreamReader inr =
                new InputStreamReader(sock.getInputStream());
            BufferedReader in = new BufferedReader(inr);
            String req = in.readLine();
            System.out.println("req " + req);
            StringTokenizer st = new StringTokenizer(req);
            if ((st.countTokens() >= 2) &&
                (st.nextToken().equals("GET")))
            {
                req = st.nextToken();
                if (req.startsWith("/"))
                    req = req.substring(1);
                if (req.endsWith("/") || req.equals(""))
                    req = req + "index.html";
                try
```

```
                {
                    FileInputStream fin =
                        new FileInputStream(req);
                    byte [] data = new byte[fin.available()];
                    fin.read(data);
                    out.write(data);
                }
                catch(FileNotFoundException e)
                {
                    outw.println("404 Not Found");
                }
            }
            else
                outw.println("400 Bad Request");
            sock.close();
        }
        catch (IOException e)
        {
            System.out.println("IO error " + e);
        }
    }
}
```

Figure 12.1: A Java WWW Server Implementation: *httpd.java*

The Erlang version can be seen in figure 12.2.

The Erlang and the Java HTTP servers take the same approach to the task of serving web pages. Both programs setup a listening socket on which they accept incoming connections. When a connection request arrives it is accepted and a new process or thread is started to handle the request. The first line of the data sent from the web browser to the server is parsed by the process or thread and the second argument on the line names the file to be sent. The file is sent to the browser and the connection closed.

The most notable features of the Erlang version is the duality of lists and binaries through out the code, and the use of regular expression routines to perform the name manipulations.

The *gen_tcp* module allows the contents of a stream to be seen as either a collection of binary objects or as strings. In this implementation we have chosen to use binary objects as when used in conjunction with the *file:read_file* function it allows particularly easy transmission of whole files back to the browser. Note that binaries must be converted to strings for easy pattern matching and manipulation.

Regular expressions are provided by the *regexp* module. Using the *regexp:gsub* function it was possible to rewrite requests into an acceptable form without using conditional statements.

Both programs have been written with clarity as the primary objective, rather than making maximum use of language features to reduce code volume. Considering this objective it should be noted that the Erlang version is slightly shorter, yet fairly easy to read.

```
-module(httpd).
-export([start/1,server/1,reqhandler/1]).

start(Port) ->
   spawn(httpd, server, [Port]).

server(Port) ->
   {ok, Lsock} = gen_tcp:listen(Port, [binary, {packet, 0},
      {active, false}]),
   serverloop(Lsock).

serverloop(Lsock) ->
   {ok, Sock} = gen_tcp:accept(Lsock),
   spawn(httpd,reqhandler,[Sock]),
   serverloop(Lsock).

reqhandler(Sock) ->
   ReqStr = getreq(Sock),
   [FirstArg, SecondArg | Tail] = string:tokens(ReqStr, " \n\t"),
   if
      FirstArg =/= "GET" ->
         gen_tcp:send(Sock, list_to_binary("400 Bad Request\r\n"));
      true ->
         {ok, BaseName, _} = regexp:gsub(SecondArg, "/$|^$",
             "/index.html"),
         {ok, File, _} = regexp:sub(BaseName, "^/+", ""),
         sendfile(Sock, File)
   end,
   gen_tcp:close(Sock).

getreq(Sock) ->
   getreq(Sock, []).

getreq(Sock, OrigStr) ->
   {ok, Pack} = gen_tcp:recv(Sock, 0),
   RecStr = binary_to_list(Pack),
   NewStr = lists:append(OrigStr, RecStr),
   Pos = string:str(NewStr, "\r\n"),
   if
      Pos =/= 0 ->
         string:substr(NewStr, 1, Pos-1);
      true ->
         getreq(Sock, NewStr)
   end.

sendfile(Sock, Filename) ->
   case file:read_file(Filename) of
      {ok, Binary} ->
         gen_tcp:send(Sock, Binary);
      _ ->
         gen_tcp:send(Sock, list_to_binary("404 Not Found\r\n"))
   end.
```

Figure 12.2: An Erlang WWW Server Implementation: *httpd.erl*

12.3 Resources

The code files mentioned in this chapter are:

httpd.java
httpd.erl

These files can be retrieved from:

`http://www.serc.rmit.edu.au/~maurice/erlbk/eg/inet`.

Further information on the *gen_tcp* and *gen_udp* modules can be found in the 'The Kernel: Kernel Reference Manual' in 'Open Telecom Platform (OTP)' documentation set by Ericsson Software Technology AB, Erlang Systems. This documentation is provided in HTML and Postscript form with the Erlang distribution.

12.4 Exercises

1. Write a client in Erlang that can read a Web page from a server and store the page in a file. It should have the following interface:

   gethttp:gethttp(Url, Filename)

2. Extend the web server in figure 12.2 to handle CGI scripts (written in Erlang) using the GET method.

3. Extend the web server in figure 12.2 to handle CGI scripts (written in Erlang) using the POST method.

Chapter 13

Reliable Communications

Messages are not guaranteed to be delivered (see chapter 5). This chapter discusses several techniques which can be used to deal with unreliable communication.

13.1 No Recovery

Sometimes it may not be desirable or possible to recover from the loss of a message. Circumstances when this could arise include:

- Informational messages - Some messages convey information that is not required for the continuing safe or correct operation of a process. These messages may be discarded.
- Timely messages - Some information has a short useful lifetime. Losing this data may be less harmful than getting delayed data.
- Results of a functional conversion of data - This data can be regenerated at any time by supplying the same set of input data. Responsibility for recovery of this data can often be deferred to the initiator of the operation.

13.2 Backward Error Correction

Backward error correction is a family of techniques that are used to recover data after an error in the data has been discovered. The basic mechanism used in these techniques is to retransmit data after it has been determined that an error has occurred in the transmitted data.

13.2.1 Simple Retransmission

This retransmission mechanism is sometimes known as 'Stop and Wait ARQ' . The protocol employs a frame number and a timeout. Figure 13.1 shows an error free execution and the types of failures that this mechanism can handle.

The simple retransmission scheme relies on the initiator of a transmission re-sending a frame if an ACK (acknowledgment) is not received within the timeout. The frame number and the matching acknowledgment numbers are

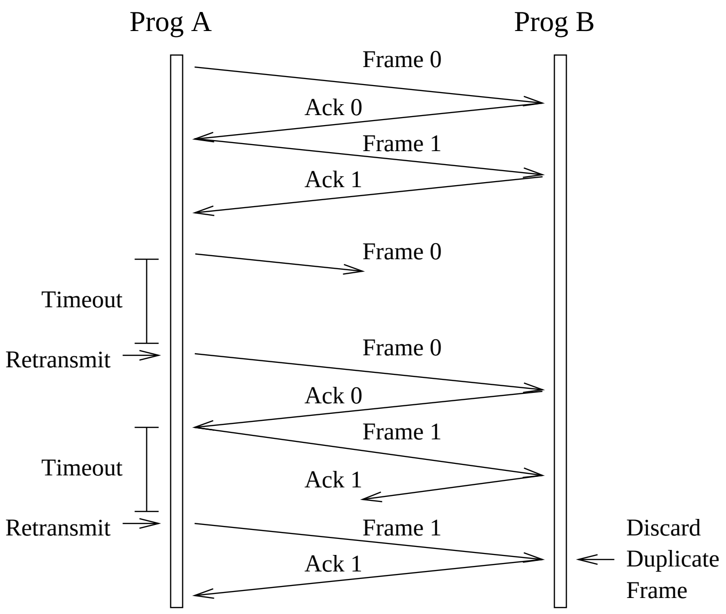


Figure 13.1: Stop and Wait ARQ

used to discover and eliminate duplicate transmissions. In the example only 2 frame and acknowledgment numbers are used as this is sufficient to discover any repeated transmission with at most one outstanding transmission. When this scheme is applied in data communications a NAK (negative acknowledgment) is often used to allow corrupted packets to be resent before the timeout expires. As Erlang communications are always correct or not present at all NAKs are not required.

Features of the simple retransmission scheme:

- Timeout delay must be greater than the round trip time of the message and the ACK.

- Timeout delay encountered before recovery can occur.

- State must be held until an ACK occurs otherwise recovery cannot occur.

- If a process is communicating with many other processes, the communication must be uniquely identified.

Figure 13.2 shows one implementation of this recovery mechanism. The program in figure 13.3 and a test run in figure 13.4 show how the *sawarq.erl* module can be used. The test program implements a tallier for adding up billing records. Billing information should not be lost so a reliable transmission mechanism is used to transfer the information. Recovering the total from the tallier requires that the information be timely, so an unreliable mechanism for transferring the data is used.

```
-module(sawarq).
-export([open/3, xmit/2, recv/1, procexists/1]).

open(Dest, Timeout, Retry) ->
    RefId = make_ref(),
    {Dest, RefId, 0, Timeout, Retry}.

xmit({Dest, RefId, N, Timeout, Retry}, Mesg) ->
    Dest ! {self(), RefId, N, Mesg},
    receive
        {RefId, N, ack} ->
            {Dest, RefId, (N+1) rem 2, Timeout, Retry};
        {RefId, OtherN, ack} ->
            error
    after Timeout ->
        case procexists(Dest) of
            true ->
                receive
                after Retry ->
                    ok
                end,
                xmit({Dest, RefId, N, Timeout, Retry}, Mesg);
            _ ->
                error
        end
    end.

recv(N) ->
    receive
        {Sender, RefId, N, Mesg} ->
            Sender ! {RefId, N, ack},
            {(N+1) rem 2, Mesg};
        {Sender, RefId, _, Mesg} ->
            error
    end.

procexists(Pid) when pid(Pid) ->
    Nd = node(Pid),
    ThisNd = node(),
    ListProc = if
        Nd == ThisNd ->
            processes();
        true ->
            rpc:call(Nd, erlang, processes, [])
    end,
    lists:member(Pid, ListProc);
procexists(Name) ->
    case whereis(Name) of
        undefined -> false;
        _ -> true
    end.
```

Figure 13.2: Source code for implementing ‘Stop and Wait ARQ’: *sawarq.erl*

```
-module(total).
-export([start/0, read/1, add/2, tallier/2]).

timeout() -> 10000.
retry() -> 100000.

start() ->
        Pid = spawn(total, tallier, [0, 0]),
        sawarq:open(Pid, timeout(), retry()).

read(Connect) ->
        Connectp = sawarq:xmit(Connect, {read, self()}),
        Result = receive
                X -> X
        after timeout() ->
                error
        end,
        {Connectp, Result}.

add(Connect, T) ->
        sawarq:xmit(Connect, {add, T}).

tallier(N, Total) ->
        {Np, Msg} = sawarq:recv(N),
        case Msg of
                {read, Pid} ->
                        Pid  ! Total,
                        tallier(Np, Total);
                {add, T} ->
                        tallier(Np, Total + T);
                _ ->
                        tallier(Np, Total)
        end.
```

Figure 13.3: Tallier: *total.erl*

```
% erl
Erlang (BEAM) emulator version 4.6.4

Eshell V4.6.4  (abort with ^G)
1> P = total:start().
{<0.28.0>,#Ref,0,10000,100000}
2> {Pp, R} = total:read(P).
{{<0.28.0>,#Ref,1,10000,100000},0}
3> Ppp = total:add(Pp, 10).
{<0.28.0>,#Ref,0,10000,100000}
4> {Pppp, Rp} = total:read(Ppp).
{{<0.28.0>,#Ref,1,10000,100000},10}
5> Ppppp = total:add(Pppp, 10).
{<0.28.0>,#Ref,0,10000,100000}
6> {Pppppp, Rpp} = total:read(Ppppp).
{{<0.28.0>,#Ref,1,10000,100000},20}
7>
```

Figure 13.4: Testing the Tallier

13.2.2 Retransmission with Windows

The mechanism described in section 13.2.1 works well if communication times are short. In environments with long communication times the ACKs slow down the protocol. A modification of the protocol allows a number of frames to be outstanding, this set of frames is called the window. The sender transmits frames until he reaches the window size. When the first frames in the window are acknowledged the window slides and more frames are transmitted. Frames which do not get acknowledged are retransmitted. This approach allows longer latencies in the response, but without compromising the throughput as much as 'Stop and Wait ARQ'. These protocols are collectively refered to as 'sliding window' protocols.

13.3 Forward Error Correction

Forward error correction is a family of techniques that are used to recover data when errors occur. Unlike backward error correction these techniques do not wait for an error to be discovered. Instead these techniques are based on transmitting additional redundant information with the transmitted data. The redundant information is used to reconstruct the data if it is corrupted or lost.

These techniques can be broken into two classes. The first class transmits sufficient data to reconstruct the lost data exactly. The second class transmits some collective property of the data that can be used for approximating the lost data. The second class tends to be more compact, but can only be used where an exact representation is not required.

13.4 Resources

The code files mentioned in this chapter are:

sawarq.erl
total.erl

These files can be retrieved from:

`http://www.serc.rmit.edu.au/~maurice/erlbk/eg/relcom`.

Further information can be found on constructing reliable communications on unreliable media in text books on data communications. One which has been used by the author is: Stallings, W., 'Data and Computer Communications', 4 Ed, Macmillan, 1994.

13.5 Exercises

1. Examine the tallier in figure 13.3 for errors which will cause it to fail and lose the total.
2. Using the code in figure 13.2 as a basis write an implementation of a sliding window protocol with a user specifiable window size

Chapter 14

Reliability and Fault Tolerance

Real time systems are often used in safety critical situations – situations where failure of the system may lead to loss of life – and situations where the timeliness of data or actions is critical to the commercial viability of a business. In these circumstances the failure of a program can have drastic consequences. This chapter focuses on how reliable and fault tolerant systems can be built. Techniques and approaches are introduced that allow systems to recognise and handle faults.

14.1 Terminology

A number of terms will be used in this chapter to describe the behavior of a malfunctioning system:

Failure – The deviation of a system's behavior from the specification

Error – An instance of a deviation from specification

Fault – The mechanical or algorithmic cause of an error

Faults can be classified by their temporal characteristics:

Transient Fault – A fault that starts at some time, remains in the system for a time, and then disappears from the system

Permanent Fault – A fault that starts at some time, and remains in the system until it is repaired

Intermittent Fault – A transient fault that recurs from time to time

Failures in real time systems can be classified into two modes:

Value failure – an incorrect value is returned

Time failure – a service occurs at the wrong time

14.2 Fault Prevention

Fault prevention is divided into fault avoidance and fault removal. Fault avoidance focuses on writing fault free programs. Techniques such as rigorous or formal specification of requirements; and the use of proven design methodologies are used to limit the introduction of errors into programs. Fault removal uses code reviews and system testing to detect faults and remove them.

System testing is imperfect:

- a test cannot detect the absence of an error
- realistic test conditions cannot always be created
- requirements stage errors often cannot be detected until the system is made operational

The functional nature of Erlang provides the system tester with the advantage of being able to test functions individually. Furthermore, if functions have a functional behavior they can be used with other functions and result in a known outcome. Programming languages which allow state to be stored with a function or procedure do not have this property.

14.3 Fault Tolerance

A fault tolerant system can provide either full service or some reduced degree of service after a fault occurs. Systems can be grouped by the degree of service provided:

Full fault tolerance – no loss of service (either functionality or performance) in the presence of errors

Graceful degradation (a.k.a. Fail soft) – system continues to operate, but with some level of service degradation, until the system either recovers or is repaired

Fail safe – system ensures its integrity but stops delivering services

14.3.1 Redundancy

To provide service in the presence of errors redundancy is introduced into the system. Extra elements are added to the system to allow the system to recover from faults. Redundancy can be either static or dynamic.

Static Redundancy

Parts of the system are replicated in order to continue to provide service in the event of the failure of a replicated component.

Triple Modular Redundancy (TMR) is a special case of N Modular Redundancy (NMR). N Modular Redundancy (see figure 14.1) replicates a system N times and employs a voting system to determine the result given. The replicated systems are not permitted to interact with each other. This approach can lead to a less reliable system as it increases the complexity of the system

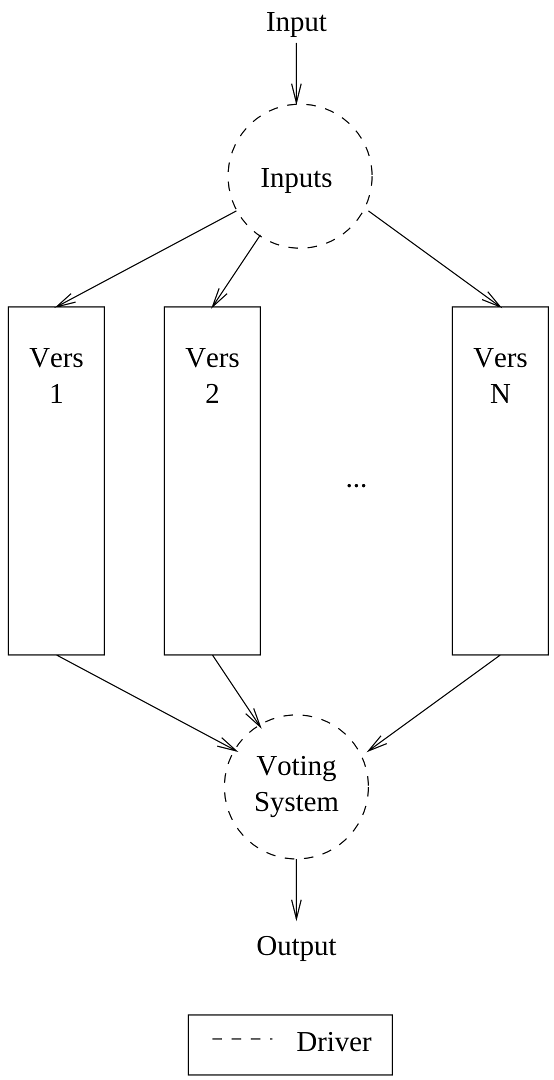


Figure 14.1: N Module Redundancy

and the voting scheme may introduce errors (common problems include failures in synchronisation and failures in comparisons). Each of the versions of the program must differ in some way such that the faults in one version are not related to faults in another version. This difference may be achieved through using different development teams, different algorithms, or different compilers.

The *driver* provides both the required inputs and compares the results returned by the versions. Two typical implementations of the *driver* process exist. The simpler implementation if for one-shot operations. In this implementation, the driver invokes each process, wait for results and acts on the results. The more complex implementation allows for continuous operation. Continuous interaction is achieved by invoking the processes, providing inputs as required, collecting results when comparison points are reached, and generating actions. As calculation errors can accumulate in some systems leading to correct programs diverging, some implementations may resynchronise the state of the versions at the comparison points to ensure that any divergence in results is due to a fault in the system rather than accumulated calculation errors.

The time between comparisons (the granularity of the comparison process) is significant as a longer period tends to increase the divergence of the results returned by each version. However, a shorter period results in increased overhead associated with the collection of results from each of the versions.

Another influence on the time between comparisons is the required accuracy of a result and the rate a result is required. There is a class of numerical algorithms called iterative techniques. These algorithms take an approximation of a value and perform an operation on the value to improve the approximation. The number of iterations and the quality of the initial guess determine the rate at which the approximation converges with the real target value. Too few iterations results in a wide variance from the desired target. Too many iterations may return a result more accurate than required wasting machine cycles.

The method of vote comparison is critical to the implementation of NMR. Where results are integers or strings, comparisons are a straight forward matter of comparing votes and returning the majority decision. Comparing floating point (real) values is more complex.

Measurements of real systems usually return results which are more accurately represented than measured. This typically results in a spread of results being returned for the same real result measured. Although two calculations may be equivalent in exact arithmetic, their results may differ when performed with the finite arithmetic used to express floating point numbers. Thus a spread of results is also possible when differing algorithms and implementations are used to perform calculations using floating point numbers.

One technique used for comparing floating point numbers is to compare values with a threshold and use the result of this comparison to return a symbolic result which can be used in an exact voting scheme (see figureinexact). This method performs well when the inexact results are not near the threshold. When results are close to the threshold (within the error in the calculation or measurement) the symbolic result is unreliable. Adding a tolerance to the result does not solve the problem, it merely moves the unreliability from results about the threshold value to results near the tolerance values (see figureinextol).

With vote comparison based on inexact data disagreement is possible without the event of an error.

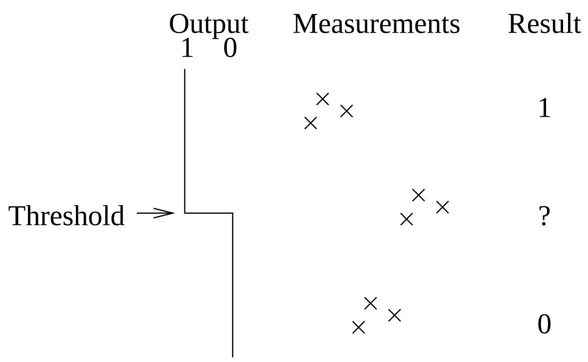


Figure 14.2: Inexact Comparisons of Measured Data

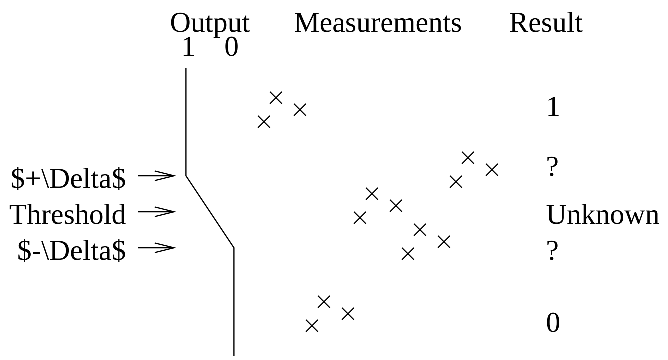


Figure 14.3: Inexact Comparisons of Measured Data – with Tolerance

Dynamic Redundancy

Static redundancy duplicates components that are used regardless of whether a fault has occurred or not. In dynamic redundancy, the redundant components are only used when a fault has been detected.

The dynamically redundant technique for implementing a fault tolerant system introduced here consists of four phases: error detection, damage confinement and assessment, error recovery, and fault treatment and continued service.

Error detection is critical to the success of fault tolerance as the majority of faults eventually lead to errors and no fault tolerant scheme can operate until an error is detected.

Environmental detection of errors relies on the environment in which a program executes to alert the program to a failure. Erlang uses error trapping, catch and linked processes to report the errors described in section 8.1. Other languages rely on the operating system to generate exceptions (in Unix these are called signals) when a program exceeds the restrictions provided by the environment.

In the application detection approach an application detects errors itself. Some techniques which can be used include:

- Replication Checks – Use of NMR to compute and compare results. Typically 2 versions are used and a disagreement indicates a fault.

- Timing Checks: Watch Dog Timer – A timer is associated with a component. The timer is reset on correct interactions with the component. If the timer expires the component is assumed to be in error. This is related to a heart beat . A component uses a timer to trigger periodic signals to a process monitoring it. Failure of these signals to arrive indicates an error in the component.

- Timing Checks: Deadlines – Where timely response is required, missing a deadline is an error.

- Reversal Checks – Where there is a one to one relationship between the inputs and the outputs of a component, an output value can be used to compute the value of the input. Comparing the input with the calculated value allows the operation of the component to be checked.

- Error Detecting Codes – The integrity of data can be checked by using an error detecting code to provide redundant data. Common examples include checksums and parity.

- Reasonableness Checks – Using knowledge of the design and construction of the system, programmers can construct tests that values or the state of the system are reasonable. These tests can be subsetted into: consistency tests which check that related values fall within the expected relationship; and constraint tests which check that values fall inside an expected set of values. These tests can either be explicitly coded or implicitly represented. In C these test are often explicitly coded as assertions. Types and subtypes can be used in Ada to implicitly code an expected set of values. In Erlang, reasonableness checks are typically explicitly coded with no additional language support.

- Structural Checks – The integrity of data structures such as lists, queues and trees can be checked by adding redundant information to the structure. Common methods are to include element counts or redundant pointers. Erlang does not support the second method as it does not support pointers.
- Dynamic Reasonableness Checks – The output from a component can be related to previous outputs from a component. If there is a relationship between consecutive outputs a bound can be placed on the difference between each output. If the outputs are too dissimilar an error can be assumed to have occurred.

Only some of the techniques may be feasible or useful in a given situation or program.

The damage confinement and assessment phase occurs after an error has been detected. This phase assesses how corrupt the system has been made by the error. Factors involved in this assessment are: the type of error encountered, the period of time between the fault occurring and the error being detected, and how well the system contains the spread of an error.

Design techniques are used to confine damage. Interfaces can test data passed to them to ensure reasonableness and contain errors. A modular decomposition defines a set of interfaces through which information is passed. Data transfers which avoid these interfaces should be eliminated. In Erlang these data transfers would take place as either: messages sent directly rather than using a function provided by a module to correctly format the message; or, directly accessing a data structure passed back by a function when the module has functions for manipulating the data structure. The defined set of interfaces eases the task of determining the impact of an error. Implementing shifts from one consistent state to another consistent state using atomic actions can confine an error to a single process or state within a process.

Error recovery is performed after the damage has been assessed.

Forward error recovery attempts to take a system from a damaged state into a correct state by applying corrections to selected elements of the system state.

Backward error recovery techniques restore a system to a safe state before the error occurred and an alternative section of the program is then executed.

Checkpointing is a technique where the system state is stored. These recovery points can then be used to restore the system state if an error is detected. Storing the whole system state can be expensive or impractical. Incremental checkpointing can be used to reduce the cost by storing only the changes in state from a stable state. Where multiple processes are employed care must be taken to ensure that the checkpoints used give a consistent state for the whole system, where processes have communicated it may be necessary to undo the effect of the communication.

Although error recovery has removed the fault from the system, the possibility of the fault recurring exists. Fault treatment and continued service is concerned with removing the cause of the fault. Logging should be used to provide information to locate the fault and the component should be repaired to prevent the fault recurring. Unlike many other languages Erlang is well suited to fault repair as its code modules can be replaced on the fly (see chapter 9).

In section 14.3.1 the *driver* was introduced as an implementation of static redundancy. The recovery block approach will be introduced as an approach to implementing dynamic redundancy. The approach is a backward error recovery technique. Recovery blocks can be nested. The technique is illustrated as a flow chart in figure 14.4

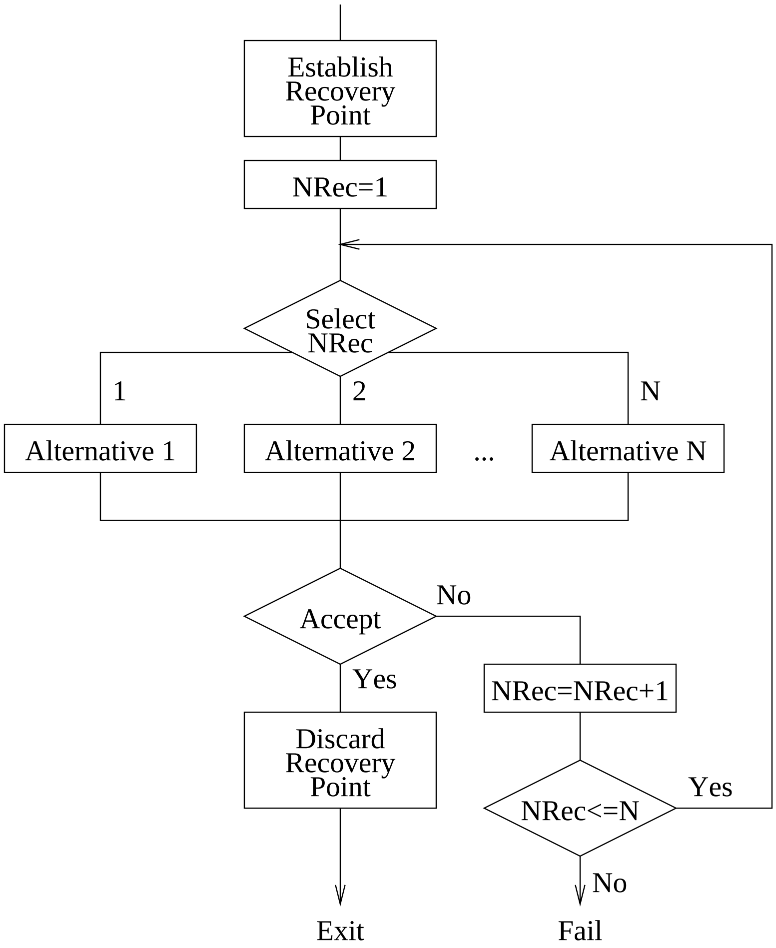


Figure 14.4: Algorithm for Recovery Blocks

The essence of the recovery block approach is to save a recovery point and try a series of implementations until the acceptance test is met. It is important to note that an acceptance test is used not a correctness test. The test is present to ensure the stability of the system not to test the correctness of the system.

14.4 When Recovery is Undesirable

There are some circumstances when recovery is not desirable in these cases the recovery action either complicates the system without providing gain, or recovery would come too late to be of benefit. An example of this is the failure of a telephone exchange. Attempting to restore the calls that were present at the time of the failure is difficult as the phone users who were cut off are probably not expecting to have their calls restored. Thus a lot of work would be done for people who no longer require the service. It should be noted that within this system there are still components that would require recovery. The billing system is an example of this as the owner of the exchange wants to be paid for the service provided to the phone users.

14.5 Resources

The code files mentioned in this chapter are:

exrev1.erl
exrev2.erl

These files can be retrieved from:

`http://www.serc.rmit.edu.au/~maurice/erlbk/eg/relflt`.

Further information on the topics discussed in this chapter can be found in Burns, A., Wellings A., 'Real-Time System and Programming Languages', 2 ed, Addison Wesley Longman 1997.

14.6 Exercises

1. Devise a real time system where there are several algorithms which are applicable to solving the same problem.
2. Implement a driver process in Erlang for the system you devised.
3. Implement recovery blocks in Erlang for the system you devised.
4. Discuss the advantages and disadvantages of driver processes and recovery blocks.
5. Examine the functions in figure 14.5 determine if they are suitable for a reversal check and if possible design a reversal check for it.

```
-module(exrev1).
-export([f/1]).

f(X) ->
        1 + math:sqrt(X).
```

```
-module(exrev2).
-export([f/1]).

f(X) ->
        0.8 * math:cos(X).
```

Figure 14.5: *exrev1.erl* and *exrev2.erl*

Index

Glossary

ACK Acknowledgment

ARQ Automatic repeat request

Arity Number fundamentally associated with an entity. In Erlang, the arity of a function is its number of arguments

Atom a constant name

BIF are members of a special class of functions known as Built In Funtions. These functions are built in to the interpreter in an interpreted environment.

BST Binary Search Tree

Backward Error Correction recovers data by having the transmitter re-send lost or corrupted data

Backward Error Recovery A class of error recovery techniques that restore a system to a safe state before the error occurred and execute an alternative section of the program

Binding The association of a variable name to the contents of the variable.

Catch all matches any pattern.

Convention A convention is an agreed means for handling a specified circumstance. It need not be enforced and may be arbitrary in nature. The general use of conventions in programming is to improve the readability and understandability of code.

Elements the items that a tuple or list are constructed from

Erlang Shell An environment which allows users to directly interact with Erlang functions

Error An instance of a deviation from specification

Fail safe System ensures its integrity but stops delivering services

Fail soft see graceful degradation

Failure The deviation of a system's behavior from the specification

Fault The mechanical or algorithmic cause of an error

Float a number with a fractional part (ie. no decimal point)

Forward Error Correction recovers data by augmenting the data from the transmitter with additional data that can be used to reconstruct lost or corrupted data

Forward Error Recovery A class of error recovery techniques that attempt to take a system from a damaged state into a correct state by applying corrections to selected elements of the system state

Full fault tolerance No loss of service (either functionality or performance) in the presence of errors

Fully qualified name consists of the module name a colon and the function name.

Graceful degradation System continues to operate, but with some level of service degradation, until the system either recovers or is repaired

Granularity Size of an element or component of a calculation

IPC Inter-Process Communication

Induction a process of constructing a general result for a given problem from a given set of facts relating to the problem

Integer a positive or negative number with no fractional part (ie. no decimal point)

List a variable length collection of elements

NAK Negative acknowledgment

NMR N Modular Redundancy

Pid a process identifier

Process Dictionary A process's private associative store

Proper list a proper list has an empty list ([]) as its last element

Recursion A function which calls itself or calls a function or series of functions which call the original function

Redundant Additional component which does not contribute to normal operation

Reference a unique value that can be copied or passed but cannot be generated again

Scope the scope of an entity is the section of program in which an entity can be accessed or named.

Side Effect an interaction between a function and its environment other than through its input parameters or its output value

Single assignment a variable may be assigned (bound) exactly once

Static Redundancy Replication of a component or service

TMR Triple Modular Redundancy

Term A value. An integer, float, atom, pid, reference, list, or tuple and any of list or tuple composed of these types

Tuple a fixed length collection of elements

Well formed list a well formed list has an empty list ([]) as its last element