Erlang Example
Programs

Joe Armstrong

November 30, 1998
Contents

1 Introduction ................................................. 3

2 FTP .................................................................. 5
   2.1 The FTP client ........................................ 5
   2.2 The FTP server ........................................ 7
   2.3 Problems ............................................... 9
   2.4 Exercises ............................................. 10

3 Utilities ........................................................ 11
   3.1 find ........................................................ 11
   3.2 Ermake ................................................... 15
   3.3 Makefile format ........................................ 15
   3.4 How make works ...................................... 16

4 Parser Tutorial .................................................. 21
   4.1 Parsing with yecc and leex .......................... 21
       4.1.1 Overview ........................................ 21
       4.1.2 An example program .......................... 22
       4.1.3 A sample session ................................ 23
       4.1.4 The yecc grammar .............................. 24
       4.1.5 The leex grammar .............................. 26

5 Cryptography ...................................................... 29
   5.1 A numerical diversion ............................... 29
   5.2 Lin ....................................................... 29
   5.3 Primes .................................................... 33
   5.4 RSA ....................................................... 36
CONTENTS

6 Sos 39
  6.1 A Simple Operating System .......................... 39
  6.2 Example programs ...................................... 41
  6.3 sos .................................................. 43
  6.4 The sos main program .................................. 44
  6.5 Server support ......................................... 45
  6.6 Code server ............................................ 47
  6.7 Error Logger ........................................... 48
  6.8 Halt demon ............................................ 49
  6.9 IO server .............................................. 49
  6.10 Environment server ................................. 50
  6.11 Global process support ............................... 51
  6.12 Support for processes ............................... 52
  6.13 Utilities ............................................ 53
  6.14 Error Handler ........................................ 54
Chapter 1

Introduction

This is a collection of Erlang example programs. The document is also available as a postscript or pdf file. All the programs referred to are available in the archive examples-2.0.tgz.

The programs attempt to do useful things, thus you will find complete source code for a number of useful Erlang utilities such as make, find ftp etc. which model the behaviours of the Unix programs with the same names.
Chapter 2

FTP

This chapter describes how FTP (File Transfer Protocol) could have been implemented if the implementation language had been Erlang. Instead of following RFC 959 we will implemented the core functionality of a file transfer protocol in two Erlang modules `ftp_client` and `ftp_server`.

2.1 The FTP client

`ftp_client` exports the following function:

```
connect(hostname(), user(), password()) -> {ok, handle()}|error
```

Tries to open an FTP session on `hostname()` with the given user name and password. If successful this returns a handle that can be used to communicate with the FTP server.

```
pwd(handle()) -> string()
```

Returns the current working directory of the FTP server.

```
file(handle(), dir()) -> {ok, dir()} | error
```

Changes the working directory of the FTP server.

```
ls(dir()) -> [file()|dir()]
```

Returns a listing of the contents of the working directory at the remote FTP server.

```
put(handle(), file()) -> ok | {error, Reason}
```

Transfers `file()` to the remote FTP server.

```
get(handle(), file()) -> ok | {error, Reason}
```

Fetches a file from the remote FTP server.
1cd(dir()) -> ok | error
Changes the local current working directory to dir().

1pwd() -> dir()
Returns the local current working directory.

1ls() -> [file()|dir()].
Lists the contents of the local current working directory.

The implementation of these commands is straightforward, firstly, connect/3, which tries to establish a connection to the FTP server:

connect(Host, User, Password) ->
   {ftp_server, Host} ! {connect, self(), User, Password},
   receive
   {ftp_server, Reply} -> Reply;
   Other -> Other
after 10000 ->
   timeout
end.

This just sends a connect request to the server and waits for a reply. pwd/1, cd/2, ls/1, get/2 and quit/1 are all implemented as remote procedure calls:

pwd(Handle)       -> remote(Handle, pwd).
cd(Handle, Dir)   -> remote(Handle, {cd, Dir}).
ls(Handle)        -> remote(Handle, ls).
get(Handle, File) -> remote(Handle, {get, File}).
quit(Handle)      -> remote(Handle, quit).

Where:

remote(Handle, Op) ->
   Handle ! {self(), Op},
   receive
   {ftp_server, Any} ->
   Any
after 1000 ->
   timeout
end.
which just sends a message to the server and waits for a reply.
The other commands `lcd/1`, `lpwd/0` and `lls/0` are purely local:

\[
\begin{align*}
\text{lcd}(\text{Dir}) & \rightarrow \text{file:} \text{set_cwd}(\text{Dir}), \text{lpwd}(). \\
\text{lpwd}() & \rightarrow \text{cwd}(). \\
\text{lls}() & \rightarrow \text{element}(2, \text{file:} \text{list_dir}()().) \\
\end{align*}
\]

Finally `put/2` reads the file locally as an atom action and then calls `remote/2` to store the file on the remote FTP server.

\[
\text{put}(	ext{Handle, File}) \rightarrow \\
\quad \text{case } \text{file:read_file}(\text{File}) \text{ of} \\
\quad \quad \{\text{ok, Contents}\} \rightarrow \\
\quad \quad \quad \text{remote}(	ext{Handle, } \{\text{put, File, Contents}\}()); \\
\quad \quad \quad \text{Other} \rightarrow \\
\quad \quad \quad \quad \text{Other} \\
\quad \quad \end{align*}
\]

\section{2.2 The FTP server}

The FTP server is started by evaluating `ftp_server:start/0`:

\[
\begin{align*}
\text{start}() & \rightarrow \\
\quad \text{case } (\text{catch register}(\text{ftp_server}, \\
\quad \quad \text{spawn}(\text{?MODULE, internal, []}))) \text{ of} \\
\quad \quad \{\text{EXIT}, \_\} & \rightarrow \text{already_started}; \\
\quad \quad \text{Pid} & \rightarrow \\
\quad \quad \quad \text{ok} \\
\quad \quad \end{align*}
\]

which calls:

\[
\begin{align*}
\text{internal}() & \rightarrow \\
\quad \text{case } \text{file:consult}("\text{users}\") \text{ of} \\
\quad \quad \{\text{ok, Users}\} & \rightarrow \\
\quad \quad \quad \text{process_flag}(\text{trap_exit, true}), \\
\quad \quad \quad \text{loop}(\text{Users, 0}) \\
\quad \quad \quad \_ & \rightarrow \\
\end{align*}
\]
CHAPTER 2. FTP

exit(no_users_allowed)
end.

internal() reads the password file called user, a typical file might be:

{"joe", "zapme"}.
{"ftp", "ftp"}.
{"jane", "hospan"}.

which contains valid {User, Password} pairs.
The main loop of the server is:

loop(Users, N) ->
    receive
        {connect, Pid, User, Password} ->
            io:format("connection request from:~p~p~p~n", [Pid, User, Password]),
            case member({User, Password}, Users) of
                true ->
                    Max = max_connections(),
                    if
                        N > Max ->
                            Pid ! {ftp_server, {error, too_many_connections}},
                            loop(Users, N);
                        true ->
                            New = spawn_link(?MODULE, handler, [Pid]),
                            Pid ! {ftp_server, {ok, New}},
                            loop(Users, N + 1)
                    end;
                false ->
                    Pid ! {ftp_server, {error, rejected}},
                    loop(Users, N)
            end;
        {'EXIT', Pid} ->
            io:format("Handler ~p died~n", [Pid]),
            loop(Users, lists:max(N-1, 0));
        Any ->
            io:format("received:~p~n", [Any]),
            loop(Users, N)
    end.
Here **Users** is the content of the password file and **N** is the current number of active connections to the server. When a connect request arrives at the server, the server checks to see if the username and password are correct and that the maximum number of simultaneous connections has not been exceeded. If everything is OK the connection is accepted and a new process **spawn_link**ed to handle the connection, otherwise the connection is rejected.

The other point to note about the server loop is that if the server receives an exit signal it decrements the number of active connections, this can only happen if a handler process dies (note the main loop of the server was set to trap exits).

The individual handlers just perform the remote operations that were requested by the client:

```erlang
handler(Pid) ->
  receive
    {Pid, quit} ->
      Pid ! {ftp_server, ok};
    {Pid, Op} ->
      io:format("got:"p "n",[Pid, Op]),
      Pid ! {ftp_server, do_op(Op)},
      handler(Pid)
  end.

where

do_op({cd, Dir}) -> file:set_cwd(Dir), cwd();
do_op(ls) -> element(2, file:list_dir(cwd()));
do_op(pwd) -> cwd();
do_op({get_file, File}) -> file:read_file(File).
```

That’s it.

### 2.3 Problems

On my Linux box at home I had the following problem:

- Distributed Erlang worked when I was connected to the net at work, but
- Distributed Erlang didn’t work when I was not connected to the net
The fix: as root give the following commands:

/sbin/ifconfig dummy 145.222.16.27 up
/sbin/route add -host 145.222.16.27 dummy

These can be added to /etc/rc.d/rc.local (I think).

2.4 Exercises

The ftp client (39 lines) and ftp server (64 lines) were kept deliberately simple, so as not to obscure the essence of an FTP client/server. Here are a few simple ideas for extensions that could be added to the system.

- Implement a better password protection scheme
  In our program {User, Password} pairs are sent in plain-text over the net. Implement a scheme where the password is never stored, instead store the MD5 checksum of the password and transmit this over the net.

- All users have the same rights
  Implement a scheme whereby different users are restricted to which directories they may access.

- Files are sent as atomic actions
  Files are read, transmitted and written as atomic actions. This may not work if the files become very large. Implement a scheme for sending the files in smaller chunks. Implement a scheme whereby an FTP transfer can be aborted and restarted in the case where we transfer very large files.

- ftp uses file:set_cwd which globally changes the current working directory. If two FTP clients simultaneously access the FTP server weird things will start happening. Changes to the working directory made by the first client will be visible to the second client etc. Correct this behaviour.
Chapter 3

Utilities

This chapter describes `find` and `ermake` which are Erlang versions of the familiar Unix utilities with similar names.

3.1 find

`find` exports the following functions:

files(Dir, ReExpr, Recursive) -> [File]
    Find regular files starting from Dir which match ReExpr. If Recursive
    is true then call find recursively on all sub-directories.
    For example, `find(".", ".*erl", false)` will find all Erlang files
    in the current directory

out_of_date(Dir, SrcExt, ObjExt)
    Finds all "out of date files" in Dir.
    For example, `out_of_date(".", ".erl", ".jam")` finds all out of
    date files in the current directory

`find:files/3` is pretty straightforward:

files(Dir, Re, Flag) ->
    Re1 = string:re_sh_to_awk(Re),
    find_files(Dir, Re1, Flag, []).
find_files(Dir, Re, Flag, L) ->
case file:list_dir(Dir) of
  {ok, Files} -> find_files(Files, Dir, Re, Flag, L);
  {error, _} -> L
end.

And finally

find_files([File|T], Dir, Re, Recursive, L) ->
  FullName = Dir ++ [$/|File],
  case file_type(FullName) of
    regular ->
      case string:re_match(FullName, Re) of
        {match, _, _} ->
          find_files(T, Dir, Re, Recursive, [FullName|L]);
        _ ->
          find_files(T, Dir, Re, Recursive, L)
      end;
    directory ->
      case Recursive of
        true ->
          L1 = find_files(FullName, Re, Recursive, L),
          find_files(T, Dir, Re, Recursive, L1);
        false ->
          find_files(T, Dir, Re, Recursive, L)
      end;
    error ->
      find_files(T, Dir, Re, Recursive, L)
  end;
  find_files([], _, _, _, L) ->
    L.

In addition we need something that classifies the file type:

file_type(File) ->
case file:file_info(File) of
  {ok, Facts} ->
    case element(2, Facts) of
      regular -> regular;
      directory -> directory;
      _ -> error
end;
_  ->
  error
end.

The other utility function in find.erl is out_of_date/3, this is as follows:

\[
\text{out_of_date(Dir, In, Out)} \rightarrow
\]
\[
\begin{align*}
\text{case file:list_dir(Dir) of} \\
\{\text{ok, Files0}\} \rightarrow
\text{Files1 = filter(fun(F) ->} \\
\text{suffix(In, F)}
\text{end, Files0),}
\text{Files2 = map(fun(F) ->} \\
\text{sublist(F, 1,}
\text{length(F)-length(In))}
\text{end, Files1),}
\text{filter(fun(F) -> update(F, In, Out) end,Files2);}
\_  ->
\[]
\end{align*}
\]
end.

Which calls:

\[
\text{update(File, In, Out)} \rightarrow
\]
\[
\begin{align*}
\text{InFile } &= \text{File ++ In,}
\text{OutFile } &= \text{File ++ Out,}
\text{case is_file(OutFile) of} \\
\text{true } &\rightarrow
\text{case writeable(OutFile) of} \\
\text{true } &\rightarrow
\text{outofdate(InFile, OutFile);}
\text{false } &\rightarrow
\text{%% can't write so we can't update}
\text{false}
\text{end;}
\text{false } &\rightarrow
\text{%% doesn't exist}
\text{true}
\text{end.}
\end{align*}
\]
This calls the predicates `is_file` and `writeable`:

```prolog
is_file(File) ->
    case file:file_info(File) of
        {ok, _} ->
            true;
        _ ->
            false
    end.

writeable(F) ->
    case file:file_info(F) of
        {ok, {_,_,_,read_write,_,_,_}} -> true;
        {ok, {_,_,_,write_,_,_,_}} -> true;
        _ -> false
    end.
```

and then checks if the file needs to be updated by calling `outofdate`:

```prolog
outofdate(In, Out) ->
    case {last_modified(In), last_modified(Out)} of
        {T1, T2} when T1 > T2 ->
            true;
        _ ->
            false
    end.
```

Finally we need the last modified date of the file:

```prolog
last_modified(F) ->
    case file:file_info(F) of
        {ok, {_,_,_,_,Time,_,_}} ->
            Time;
        _ ->
            exit({last_modified, F})
    end.
```
3.2 Ermake

\texttt{ermake} is a make utility for Erlang. \texttt{ermake} exports the following functions:

\begin{verbatim}
all()
\end{verbatim}

Makes the first target in \texttt{Makefile}.

\begin{verbatim}
file(File)
\end{verbatim}

Makes the first target in \texttt{File}.

\begin{verbatim}
target(Target)
\end{verbatim}

Makes \texttt{Target} in \texttt{EMakefile}.

\begin{verbatim}
target(File, Target)
\end{verbatim}

Makes \texttt{Target} in \texttt{File}.

3.3 Makefile format

A typical makefile looks like:

\begin{verbatim}
include("$(MAKEDIR)/suffix\_rules").
\end{verbatim}

\begin{verbatim}
JAMS = a.jam, b.jam, c.jam.
\end{verbatim}

\begin{verbatim}
all when $(JAMS) ->
\end{verbatim}

\begin{verbatim}
true.
\end{verbatim}

Entries in the makefile may span several lines. Each entry is terminated by a dot followed by a whitespace. Three different types of entity are permitted.

- Macro definitions
  These are line of the form \texttt{VAR = string}.

- Include files
  These are written \texttt{include("File")}.

- Make commands
  These are written with the syntax:

\begin{verbatim}
T1, T2, T3, ..., Tn when D1, D2, ..., Dn ->
\end{verbatim}

\begin{verbatim}
Expressions.
\end{verbatim}
In addition the notation $\$(VAR)\$ means the value of the variable \VAR, $\$>\$ means the root name of the current target, and the pre-defined variable \MAKEDIR\ contains the root directory of the \ermake\ program.

The file \suffix\_rules\ is as follows:

Suffix \texttt{.erl.jam} \rightarrow
\hspace{1cm} \texttt{c:c(\$\textgreater\$)}.

Suffix \texttt{.tex.dvi} \rightarrow
\hspace{1cm} \texttt{unix:cmd("latex \$\textgreater\$.\text{tex}")}.

Suffix \texttt{.ehtml.html} \rightarrow
\hspace{1cm} \texttt{ehtml:file("\$\textgreater\$")}.

### 3.4 How make works

Make is a fairly complex program which is built from three modules:

\texttt{ermake\_line\_reader.\texttt{erl}}

This provides character level input for utilities such as make, lex, yacc etc. This module reads lines up to dot whitespace. Include files and named macros are expanded in place.

\texttt{ermake\_parse.\texttt{erl}}

 Parses the lines in the makefile. This is very simple (too simple!).

\texttt{ermake.\texttt{erl}}

The guts of the make program. This is where all the fun is.

\texttt{ermake.\texttt{erl}} is a long and complex program (232 lines). Rather than explain all the details I'll just give a brief overview of how it works. Once the principles of make have been understood the details are easy to understand.

We start with a simple example. Suppose we have a makefile consisting of the following:

all when a,b \rightarrow
\hspace{1cm} C1.

a when d,e \rightarrow
\hspace{1cm} C2.

b when x \rightarrow
The initial target in this makefile is the goal all. The first step in generating the makefile dependencies is to generate the set of pairs \( \{I, J\} \) where \( I \) depends upon \( J \). For the above makefile this is the set:

\[
L = \{\{\text{all, a}\}, \{\text{all, b}\}, \{a, d\}, \{a, e\}, \{b, x\}, \{c, y\}\}
\]

This can be read as \( \text{all depends upon a, all depends upon b, etc.} \)

Now we compute the transitive closure of the dependency graph as follows:

\[
\text{transitive:closure([all], L).}
\]

\[
[d,e,x,b,a,all]
\]

This is the set of all targets that are reachable from the start symbol all. Note that the symbols \( c \) and \( y \) cannot be reached.

Next we remove any pairs from the dependency graph that cannot be reached, this results in the list:

\[
L1 = \{\{\text{all, a}\}, \{\text{all, b}\}, \{a, d\}, \{a, e\}, \{b, x\}\}
\]

The makefile ordering is now found by first reversing the order of each pair and then performing a topological sort of the result:

\[
> L2 = \text{lists:map(fun(\{I,J\}) -> \{J, I\} end, L1).}
\]

\[
[[a,all],\{b,all\},\{d,a\},\{e,a\},\{x,b\}]
\]

\[
> \text{topological_sort:sort(L2).}
\]

\[
\{\text{ok, [x,e,d,b,a,all]}\}
\]

Yielding the final order in which things have to be made, namely:

\[
[x,e,d,b,a,all]
\]

Now that we know the order all we have to do is look back to the original rules to figure out which rule to apply.

Computing the order in which to build the targets is performed by \text{ermake:make_everything/2}:
make_everything(Targets, Rules) ->
  Ps = [{T, D} || {make, Ts, Ds, _} <- Rules, T <- Ts, D <- Ds],
  L0 = transitive_closure( Targets, Ps),
  L = delete(true, L0),
  Ps1 = filter(fun({D, T}) ->
    member(D, L) or member(T, L)
  end, Ps),
  L = delete(true, L0),
  reverse the order to build the bottom up tree
  Ps2 = map(fun({I, J}) -> {J, I} end, Ps1),
  % order the result
  case topological_sort(sort(Ps2)) of
    {ok, Order0} ->
      Order = delete(true, Order0),
      % order is the absolute order to build things
      Cmds = map(fun(I) -> select_rule(I, Rules) end, Order),
      foldl(fun do_cmd/2, [], Cmds),
      true;
    {cycle, Cycle} ->
      exit({makefile, contains, cycle, Cycle})
  end.

[Note:  ermake also handles make rules where the when alternative is omitted, to handle these then pseudo dependent true is created, this has to be removed from the closure and topologically ordered sets]

The transitive closure was computed by:

  closure( RootSet, Pairs) ->
  closure_list( RootSet, Pairs, RootSet).

  closure( Start, [], L) ->
    L;
  closure( Start, Pairs, Reachable) ->
    {Next, Rest} = next(Start, Pairs),
    closure_list(Next, Rest, Next ++ Reachable).

  closure_list( [], Pairs, Reachable) ->
    Reachable;
  closure_list([H|T], Pairs, Reachable) ->
    Reachable1 = closure(H, Pairs, Reachable),
3.4. HOW MAKE WORKS

```
closure_list(T, Pairs, Reachable).

next(Start, Pairs) ->
  next(Start, Pairs, [], []).

next(Start, [], Reach, NoReach) ->
  {Reach, NoReach};
next(Start, [{Start|Next}|T], Reach, NoReach) ->
  next(Start, T, [Next|Reach], NoReach);
next(Start, [T|T], Reach, NoReach) ->
  next(Start, T, Reach, [H|NoReach]).

And the topological sort by:

sort(Pairs) ->
  iterate(Pairs, [], all(Pairs)).
iterate([], L, All) ->
  {ok, remove_duplicates(L ++ subtract(All, L))};
iterate(Pairs, L, All) ->
  case subtract(lhs(Pairs), rhs(Pairs)) of
    [] ->
      {cycle, Pairs};
    Lhs ->
      iterate(remove_pairs(Lhs, Pairs), L ++ Lhs, All)
  end.
all(L) -> lhs(L) ++ rhs(L).
lhs(L) -> map(fun({X, _}) -> X end, L).
rhs(L) -> map(fun({_, Y}) -> Y end, L).

%% subtract(L1, L2) -> all the elements in L1 which are not in L2
subtract(L1, L2) ->
  filter(fun(X) -> not member(X, L2) end, L1).
remove_duplicates([H|T]) ->
  case member(H, T) of
    true -> remove_duplicates(T);
    false -> [H|remove_duplicates(T)]
  end;
remove_duplicates([]) ->
```


% remove_pairs(L1, L2) -> L2' L1 = [X] L2 = [{X,Y}]
% removes all pairs from L2 where the first element
% of each pair is a member of L1

remove_pairs(L1, L2) -> filter(fun({X,Y}) -> not member(X, L1) end, L2).

The remainder of ermake.erl fills in the details not covered here.

Exercises

- add suffix expansions in macro expansion, i.e. if OBJS = a.jam, b.jam.
  then $(OBJS:.jam.erl) should evaluate to a.erl, b.erl.
- improve the parser making it bomb proof.
Chapter 4

Parser Tutorial

4.1 Parsing with yecc and leex

Making a parser is something which I do sufficiently infrequently that each time I’ve figured out how to do it I immediately forget all the nitty gritty details.

For my example I will make a compiler for a language called ecc. Ecc stands for Erlang Compiler Compiler and is an experiment in automatic compiler generation.

[Note: leex.erl has not yet been officially released, but an inofficial copy has been included in this directory so that you can run the examples. You should be aware of the fact that the leex input grammar may be subject to change without notice]

4.1.1 Overview

In our example we use four files:

ecc_parse.erl
This contains code which compiles and builds the parser.

- ecc_parse:make() compiles and builds the parser.
- ecc_parse:file("ebnf.ecc") tests the parser by trying to parse the file ebnf.ecc

ecc.yrl
Has the yecc grammar for the mini-language.
ecx.xml
Has the lex grammar for the mini-language.

ebnf.ecc
Is a test program written in ecc.

To run the example in this tutorial do as follows:

1. Copy all the files in this directory to a temporary directory
2. Give the commands:
   - `ecc_parse:make()` which makes the compiler.
   - `ecc_parse:file("ebnf.ecc")` which runs the generated compiler.

### 4.1.2 An example program
This is an example program written in the language ecc:

```
COMPILER ebnf.

CHARACTERS
small = "abcdefghijklmnopqrstuvwxyz";
big = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
alpha = small + big;
dig = "0123456789";
blank = CHR(9) + CHR(10) + CHR(32);
noQuote = ANY - "".

COMMENTS
FROM "(" TO "")" NESTED.

TOKENS
Nonterminal = small {alpha | dig};
Terminal = big {alpha | dig};
White = blank {blank};
String = "" { noQuote } "".

IGNORE
White + Comment.

PRODUCTIONS
```
### 4.1.3 A sample session

Here is an example of how to build the parser and some sample output from the parser.

```erlang
c> erl
Erlang (JAM) emulator version 4.5.3

Eshell V4.5.3 (abort with ^G)
c> c(ecc_parse).
{ok,ecc_parse}
c> ecc_parse:make().
Parsing input file ...
Computing states and goto table ...
Computing parse actions ...
Writing file ...
ecc_yecc.erl
./ecc_yecc.erl:589: Warning: function return_error/2 not called
./ecc_yecc.erl:589: Warning: function simplify/1 not called
Parsing file ecc.xrl, contained 20 rules.
NFA contains 131 states, DFA contains 62 states, minimised to 40 states.
Writing file ecc_lex.erl, ok
./ecc_lex.erl:687: Warning: function remove_brackets/1 not called
{ok,ecc_lex}
c> ecc_parse:file("ebnf").
Parsing ebnf.ecc
{ok,[{compiler,"ebnf"},
     {characters,[["small",{string,"abcdefghijklmnopqrstuvwxyz"}],
                  {"big",{string,"ABCDEFGHIJKLMNOPQRSTUVWXYZ"}}]}
```
4.1.4 The yecc grammar

The yecc grammar which describes the language is in the file ecc.yrl. The interesting part of the grammar is the following:

**Nonterminals**

production
form lhs factor
nested syntax char_prods charline char_rhs char_prim
ignore moreignore expr term
4.1. PARSING WITH YECC AND LEEX

Terminals
atom var string quote ']' '=' '{' '{' '(' ')' '[' ']' 
'COMPILER' 'CHARACTERS' 'COMMENTS' 'FROM' 'TO' 'TOKENS' 
'IGNORE' 'PRODUCTIONS' 'END' 'NESTED' 'EOL' 'CHR' 'ANY' integer comment 
'+,-,' 

Rootsymbol form.

form -> 'COMPILER' atom : {compiler, unwrap('2')}.  
form -> 'CHARACTERS' char_prods : {characters, '2'}.  
form -> 'COMMENTS' 'FROM' string 'TO' string nested : {comments,unwrap('3'),unwrap('5'), 
       '6'}.  
form -> 'TOKENS' syntax : {tokens, '2'}.  
form -> 'IGNORE' ignore : {ignore, '2'}.  
form -> 'PRODUCTIONS' syntax : {syntax, '2'}.  
form -> 'END' atom : {theend, '2'}.  
form -> comment.  

nested -> 'NESTED' : nested. 
nested -> 'EOL' : eol. 
nested -> '$empty' : not_nested.  

% Character syntax

char_prods -> charline ';' char_prods : ['1'|'3'] . 
char_prods -> charline : ['1'] . 

charline -> atom '=' char_rhs : {unwrap('1'), '3'}. 
char_rhs -> char_prim '+' char_rhs : {plus, '1', '3'}. 
char_rhs -> char_prim '-' char_rhs : {minus, '1', '3'}. 
char_rhs -> char_prim : '1' . 
char_prim -> 'CHR' '(' integer ')' : {chr, unwrap('3')}. 
char_prim -> string : {string, unwrap('1')}. 
char_prim -> quote : {string, unwrap('1')}. 
char_prim -> atom : {atom, unwrap('1')}. 
char_prim -> 'ANY' : any.
ignore -> var moreignore : [unwrap('$1'), '$2'].
moreignore -> '+' ignore : '$2',
moreignore -> '$empty' : [].
syntax -> production ';' syntax : ['$1', '$3'],
syntax -> production : ['$1'],
production -> lhs '=' expr : {prod, '$1', '$3'},
lhs -> var : unwrap('$1'),
lhs -> atom : unwrap('$1'),
expr -> term : '$1',
expr -> term '|' expr : {alt, '$1', '$3'},
term -> factor : '$1',
term -> factor term : {seq, '$1', '$2'},
factor -> atom : {nt, unwrap('$1')},
factor -> var : {ta, unwrap('$1')},
factor -> string : {ts, unwrap('$1')},
factor -> quote : {tq, unwrap('$1')},
factor -> '[' expr ']': {one, '$2'},
factor -> '{' expr '}' : {star, '$2'},
factor -> '(' expr ')' : {bracket, '$2'},

4.1.5 The leex grammar

The leex grammar which describes the language is in the file ecc.xr1. The interesting part of the grammar is the following:

Definitions.

Dig = [0-9]
Big = [A-Z]
Small = [a-z]
WS = \[\000-\s]\n
COMMENT = \[\\(*)(([^\*])|[^\*])\]|[^\*])\] *\*\*\*
STRING = "\\\".\".\".\".*"
QUOTE = "\\\".\".\".\".*"
4.1. PARSING WITH YECC AND LEEX

Rules.

\[
\{\text{Small}\}\{\text{Small}\}\{\text{Big}\}\{\text{Dig}\}\{\_\}^* : \{\text{token}, \{\text{atom, YYline, YYtext}\}\}\.
\]

\[
\{\text{Big}\}\{\text{Small}\}\{\text{Big}\}\{\text{Dig}\}\{\_\}^* : \{\text{token, special(YYtext, YYline)}\}\.
\]

\[
\{\text{Dig}\}\{\text{Dig}\}^* : \{\text{token, \{integer, YYline, list_to_integer(YYtext)\}}\}\.
\]

\[
\{\text{STRING}\} : \%\% \text{Strip quotes.} \\
S = \text{lists:s sublist(YYtext, 2, length(YYtext) - 2)}, \\
\{\text{token, \{string, YYline, string_gen(S)\}}\}\.
\]

\[
\{\text{QUOTE}\} : \%\% \text{Strip quotes.} \\
S = \text{lists:s sublist(YYtext, 2, length(YYtext) - 2)}, \\
\{\text{token, \{quote, YYline, string_gen(S)\}}\}\.
\]

\[
\{\text{COMMENT}\} : . \\
= : \{\text{token, \{`=`, YYline}\}}. \\
\+: \{\text{token, \{`+`, YYline}\}}. \\
\- : \{\text{token, \{`-`, YYline}\}}. \\
\; : \{\text{token, \{`;`, YYline}\}}. \\
\} : \{\text{token, \{`}`, YYline\}}. \\
\{ : \{\text{token, \{`{`, YYline\}}. \\
\[ : \{\text{token, \{`[`, YYline\}}. \\
\] : \{\text{token, \{`]`, YYline\}}. \\
\( : \{\text{token, \{`(', YYline\}}. \\
\) : \{\text{token, \{`)`, YYline\}}. \\
\| : \{\text{token, \{`|`, YYline\}}. \\
\:\ : \{\text{token, \{`:\`, YYline\}}.
\]

\[
(./\text{|\n}) : \text{skip_token}. \\
\text{.\textbackslash s\textbackslash t\textbackslash n} : \{\text{end_token, \{`$end`, YYline\}}\}.
\]
Chapter 5

Cryptography

5.1 A numerical diversion

This chapter describes a number of simple numerical algorithms which make use of Erlang bignums.

lin.erl
Contains some simple functions for linear algebra.

primes.erl
Contains routines for generating large primes.

rsa_key.erl
Shows how a public key crypto system could have been implemented in Erlang.

5.2 Lin

lin exports the following functions:

\[ \text{pow}(A, B, M) -> V \]
Computes \( V = (A^B) \mod M \)

\[ \text{inv}(A, B) -> C | \text{no_inverse} \]
Computes \( C \) such that \( A \cdot C \mod B = 1 \) if such \( C \) exists.

\[ \text{solve}(A, B) => \{X, Y\} | \text{insoluble} \]
Solves the linear congruence \( A \cdot X - B \cdot Y = 1 \) if it is solvable.
\texttt{str2int(Str)}

Converts a string to a base 256 integer.

\texttt{int2str(N)}

Converts a base 256 integer to a string.

\texttt{gcd(A, B)}

Computes the greater common denominator of \(A\) and \(B\).

Some of these are pretty simple, \texttt{pow(A, B, M)} computes \(A^B \mod M\). To compute this we proceed as follows: if \(B\) is even we compute \texttt{pow(A, B \div 2, M)} and square the result (modulo \(M\)). If \(B\) is odd and greater than one we compute \(P = \texttt{pow}(A, (B-1) \div 2, M)\) and then \(P \cdot P \cdot A \mod M\):

\begin{verbatim}
pow(A, 1, M) ->
    A \mod M;
pow(A, 2, M) ->
    A*A \mod M;
pow(A, B, M) ->
    B1 = B \div 2,
    B2 = B - B1,
    % B2 = B1 or B1+1
    P = pow(A, B1, M),
    case B2 of
        B1 -> (P*P) \mod M;
        _ -> (P*P*A) \mod M
    end.
\end{verbatim}

\texttt{gcd} is also easy:

\begin{verbatim}
gcd(A, B) when A < B -> gcd(B, A);
gcd(A, 0) -> A;
gcd(A, B) ->
    gcd(B, A \mod B).
\end{verbatim}

As are conversions between strings and integers:

\begin{verbatim}
str2int(Str) -> str2int(Str, 0).
str2int([H|T], N) -> str2int(T, N*256+H);
str2int([], N) -> N.
\end{verbatim}
5.2. LIN

\[
\text{int2str}(N) \rightarrow \text{int2str}(N, \left[\right]).
\]

\[
\text{int2str}(N, L) \text{ when } N = \leq 0 \rightarrow L;
\]

\[
\text{int2str}(N, L) \rightarrow
\]

\[
N1 = N \div 256,
\]

\[
H = N - N1 \times 256,
\]

\[
\text{int2str}(N1, \left[H\right|L\right]).
\]

**solve/2** requires some thought, before launching into the code we give some examples:

Solve \(12x - 5y = 1\) for integer \(x\) and \(y\), solution \(x = -2\) and \(y = -5\) (check \(12 \cdot -2 - 5 \cdot -5 = 1\) as required.

Solve \(28x - 25y = 1\) for integer \(x\) and \(y\), solution \(x = -8\) and \(y = -9\) (check \(28 \cdot -8 - (25 \cdot -9) = 1\) as required.

These solutions are computed as follows:

\[
\text{> lin:solve(12,5).}\n\]

\[
\{-2,-5\}\n\]

\[
\text{> lin:solve(28,25).}\n\]

\[
\{-8,-9\}\n\]

To see how to solve these congruences we give a simple example:

\[
\begin{align*}
\text{solve} & \quad 12x - 5y = 1 \quad \text{(1)} \\
\text{or} & \quad (2 \cdot 5 + 2)x - 5y = 1 \\
\text{regrouping} & \quad 2x + 5(2x - y) = 1 \\
\text{let } a &= 2x - y \quad \quad \quad \quad \quad \quad \text{(2)} \\
\text{then} & \quad 2x + 5a = 1 \quad \quad \quad \quad \text{(3)} \\
\text{or} & \quad 2x + (2 \cdot 2 + 1)a = 1 \\
\text{regrouping} & \quad 2(x + 2a) + a = 1 \\
\text{let } b &= x + 2a \quad \quad \quad \quad \quad \text{(4)} \\
\text{then} & \quad 2b + a = 1 \quad \quad \quad \quad \quad \text{(5)} \\
\quad \text{A solution to this is } b = 1, \ a = -1
\end{align*}
\]

Then from (4) \(x = b - 2a = 1 - 2(-1) = 3\) \quad \text{(6)}

and from (2) \(y = 2x - a = 2 \cdot 3 - (-1) = 7\) \quad \text{(7)}
So a solution is \((x, y) = (3, 7)\)

Check \(12\times 3 - 5\times 7 = 1\) as required

This gives us the key idea as to how to solve linear congruences.

In order to solve \(12x - 5y = 1\) (equation 1) we make a substitution (equation 2) to reduce this to a simpler form, then we have to solve the simpler sub problem which is \(2x + 5a = 1\) (equation 3). This is a simpler problem because the magnitude of the arguments are less. Eventually the process terminates when a trivial subproblem (equation 5) is encountered. Having found the solution to the sub-problem we back substitute (equations 6 and 7) to obtain the final result.

Note that some linear congruences are not solvable; \(Ax - By = 1\) is not soluble if \(A \mod B = 0\)

The above algorithm is easily encoded as:

```erlang
solve(A, B) ->
  case catch s(A, B) of
    insoluble -> insoluble;
    {X, Y} ->
      case A * X - B * Y of
        1 -> {X, Y};
        Other -> error
      end
  end.

s(A, 0) -> throw(insoluble);
s(A, 1) -> {0, -1};
s(A, -1) -> {0, 1};
s(A, B) ->
  K1 = A div B,
  K2 = A - K1*B,
  {Tmp, X} = s(B, -K2),
  {X, K1*X - Tmp}.
```

Fortunately Erlang has bignums so that:

```erlang
> lin:solve(2812971937912739173, 2103789173917397193791739173).
{-9973085600128182424840842000, -13334099116160294237}
```

Finally \(\text{inv}(A, B)\) which computes \(C\) such that \(A*C \mod B = 1\) if such an inverse exists.
5.3. PRIMES

Many applications require the use of large prime numbers. The module primes.erl can be used to generate large primes and for primality testing.

primes.erl exports the following:

make(N) -> I
Generates a random integer I with N decimal digits.

is_prime(N) -> bool()
Tests if N is a prime number

make_prime(K) -> P
Generates a random prime P with at least K decimal digits.

make(N) is easy:
make(N) -> new_seed(), make(N, 0).

make(0, D) -> D;
make(N, D) ->
    make(N-1, D*10 + (random:uniform(10)-1)).

where:

new_seed() ->
    {_, _, X} = erlang:now(),
    {H, M, S} = time(),
    H1 = H * X rem 32767,
    M1 = M * X rem 32767,
    S1 = S * X rem 32767,
put(random_seed, {H1,M1,S1}).

is_prime(N) is more interesting. We use a probabilistic method for primality testing which is based on Fermat’s little theorem.

Fermat’s little theorem states that if \( N \) is prime then \( A^N \mod N = A \). So to test if \( N \) is prime we choose some random \( A \) which is less than \( N \) and compute \( A^N \mod N \). If this is not equal to \( A \) then \( N \) is definitely not a prime. If the test succeeds then \( A \) might be a prime (certain composite numbers pass the Fermat test, these are called pseudo-primes), if we perform the test over and over again then the probability of mis-classifying the number reduces by roughly one half each time we perform the test. After (say) one hundred iterations the probability of mis-classifying a number is approximately \( 2^{-100} \). So we can be fairly sure that the classification is correct.

The classification code is as follows:

```prolog
is_prime(D) ->
  new_seed(),
  is_prime(D, 100).

is_prime(D, Ntests) ->
  N = length(integer_to_list(D)) - 1,
  is_prime(Ntests, D, N).

is_prime(0, _, _) -> true;

is_prime(Ntest, N, Len) ->
  K = random:uniform(Len),
  A = make(K),
  if A < N ->
    case lin:pow(A, N, N) of
      A -> is_prime(Ntest-1, N, Len);
       _ -> false
    end;
  true ->
    is_prime(Ntest, N, Len)
  end.
```

make_prime(K) is now easy. We generate a random integer \( N \) of \( K \) decimal digits and test to see if it is a prime. If it is not a prime we test \( N+1 \) etc. until we finally get a prime.
This process is known to terminate. 
[aside] 'Bertrand’s Postulate’ is that for every $N > 3$ there is a prime $P$ such that $N < P < 2N - 2$. Tchebychef proved this result in 1850 and Erdos shocked the world with a simple proof in 1932.

```erlang
make_prime(K) when K > 0 ->
    new_seed(),
    N = make(K),
    if N > 3 ->
        io:format("Generating a ",[K]),
        MaxTries = N - 3,
        P1 = make_prime(MaxTries, N+1),
        io:format("n",[]),
        P1;
        true ->
            make_prime(K)
    end.
make_prime(0, _) ->
    exit(impossible);
make_prime(K, P) ->
    io:format(".",[]),
    case is_prime(P) of
        true -> P;
        false -> make_prime(K-1, P+1)
    end.
```

Since generating large primes can take a long time we print out a string of dots so we have something to watch while the program runs!

```erlang```
> L1 = primes:make_prime(40).
Generating a 40 digit prime .......
723978482126573505109791832675872726791
69> L2 = primes:make_prime(40).
Generating a 40 digit prime .......
9748970581585015676886932557477060213727
70> primes:is_prime(L1*L2).
false
```

[Note: in 1912 Carmichael discovered that certain composite numbers $N$ have the property that $A^N \mod N = A$ for all $A$ less than $N$. The smallest such Carmichael number is $561 = 3 * 11 * 17$.]
> primes:is_prime(561).
true

So our primality test will fail if \( N \) is a Carmichael number. Fortunately Carmichael numbers are exceedingly rare, and it is an open problem whether there exist infinitely many Carmichael numbers. The largest known Carmichael number has 3710 digits so if you stick to bigger numbers than this you probably won't run into problems!

## 5.4 RSA

This section illustrates the principles behind public key cryptography, I will start with a few definitions and then a simple example.

Definitions: The RSA public keys system is characterized by two two-tuples of integers \( \{ N, A \} \) and \( \{ N, B \} \) one tuple is called the public key and may be openly published in a catalogue, the other is called a private key and is kept secret.

To encrypt the message \( M \) with the public key \( \{ N, A \} \) we compute \( E = M^A \mod N \) (\( E \) is now the encrypted message), to decrypt the message we compute \( E^B \mod N \) which amazingly gets you back to where you started!

Example: (from the Erlang shell)

```
1 > A = 117298167.
117298167.
2 > B = 412261863.
412261863
3 > N = 2315306317.
2315306317
4 > M = "cat".
"cat"
5 > I = lin:st2int(M).
6513012
337586
7 > D = lin:pow(E, B, N).
6513012
8 > lin:int2str(D).
"cat"
```

In lines 1-3 we define the variables used in the public key \( \{ A, N \} \) and the private key \( \{ B, N \} \). The secret message "cat" is encoded into an integer
(lines 4-5), line 6 encodes the message, line 7 decodes the message and line 8 decodes it.

Warning in some countries writing a program that evaluates $M^A \mod N$ might be illegal. Check with your local policeman before doing this!

All that remains is to figure out how to compute suitable values for the public and private keys.

The "book [Cryptography, Theory and Practice, Douglas R. Stinson, CRC Press, 1995]" says ...

1. Bob generates two large primes $p$ and $q$.
2. Bob computes $n = pq$ and $\phi(n) = (p-1)(q-1)$.
3. Bob chooses a random $b (0 < b < \phi(n))$ such that $\gcd(b, \phi(n)) = 1$
4. Bob computes $a = b^{(-1)} \mod \phi(n)$ using the Euclidean algorithm
5. Bob publishes $n$ and $b$ in a directory as his public key.

The implementation follows in a straightforward manner from the specification:

```make
make_sig(Len) ->
  %% generate two <Len> digit prime numbers
  P = primes:make_prime(Len),
  io:format("P = "p"n", [P]),
  Q = primes:make_prime(Len),
  io:format("Q = "p"n", [Q]),
  N = P*Q,
  io:format("N = "p"n", [N]),
  Phi = (P-1)*(Q-1),
  %% now make B such that B < Phi and gcd(B, Phi) = 1
  B = b(Phi),
  io:format("Public key (B) = "p"n", [B]),
  A = lin:inv(B, Phi),
  io:format("Private key (A) = "p"n", [A]),
  {B,N},{A,N}).

b(Phi) ->
  io:format("Generating a public key B ",)
  K = length(integer_to_list(Phi)) - 1,
  B = b(1, K, Phi),
```
io:format("\n", []),
B.

b(Try, K, Phi) ->
io:format(".
"),
B = primes:make(K),
if
    B < Phi ->
    case lin:gcd(B, Phi) of
    1 -> B;
       _ -> b(Try+1, K, Phi)
    end;
true ->
b(Try, K, Phi)
end.

Which we can test as follows:

> rsa_key:make_sig(20).
Generating a 20 digit prime ....................
P = 7390162411637407513
Generating a 20 digit prime ....................
Q = 78532450632904297229
N = 580367564761059672626985909764983207677
Generating a public key B ...
Public key (B) = 761323469064175805955496463001193982357
Private key (A) = 3282348550950508596270008789422146776573
{761323469064175805955496463001193982357, 580367564761059672626985909764983207677},
{3282348550950508596270008789422146776573, 580367564761059672626985909764983207677}
Chapter 6

Sos

6.1 A Simple Operating System

This chapter describes a simple operating system called sos which is designed for running simple Erlang programs. sos has been designed for fast loading from the command line prompt and makes no use of the standard Erlang libraries. It can be studied as an example of a complete Erlang application which makes very few assumptions about its environment.

sos exports the following functions:

main() 
 Starts the system.

load_module(Mod) 
 Loads the module Mod.

log_error(Error) 
 Prints the value of Error on standard output.

make_server(Name, FunState, FunHandler) 
 Creates a permanent server called Name. The initial state of the server is determined by evaluating FunState() and the "handler" function for the server is the fun FunHandler (we will say more about this later).

rpc(Name, Query) 
 Make a remote procedure call Query to the server Name.

change_behaviour(Name, FunHandler) 
 Change the behaviour of the server called Name by sending it a new handler function FunHandler.
keep_alive(Name, Fun)
  Makes sure that there is always a registered process called Name. This
  process is started (or restarted) by evaluating Fun().

make_global(Name, Fun)
  Make a global process with registered name Name. The process itself
  spawns the fun Fun().

on_exit(Pid, Fun)
  Monitor the process Pid. If this process exits with reason {'EXIT', Why}
  then evaluate Fun(Why).

on_halt(Fun)
  Set up a condition such that if a request to stop the system is made
  then Fun() will be evaluated. In the case where multiple funs have
  been specified all these funs will be called.

stop_system(Reason)
  Stop the system with reason Reason.

every(Pid, Time, Fun)
  As long as Pid has not terminated, evaluate Fun() every Time
  milliseconds.

spawn_fun(Fun)
  Creates a parallel process which evaluates Fun().

spawn_link(Fun)
  Creates and link to a parallel process which evaluates Fun().

lookup(Key, [{Key, Val}]) -> {found, Val} | not_found
  Look up a key in an dictionary.

map(F, [A]) -> [F(A)]
  Map a function over a list.

reverse([A]) -> [A]
  Reverse the order of elements in a list.

read() -> [string()] | eof
  Read the next line from standard input.

write([string()]) -> ok
  Write string to standard output.

eval(Name)
  Returns the value of the environment variable Name.

These functions can be used by simple Erlang programs.
6.2 Example programs

All our example are normal Erlang modules which must export the function main().

The simplest of all programs is sos_test1.

```
-module(sos_test1).
-doc("").
-export([main/0]).

main() ->
    sos:write("Hello world\n").
```

To run this we must first compile sos_test1 using the compiler in the Erlang development environment. Once we have done this then we can run the program as follows:

```
unix> sos sos_test1
Hello world
```

We can time this as follows:

```
> time sos sos_test1
Hello world
0.09u 0.04s 0:00.37 35.1
```

Which tells us that it took 0.09 seconds to load and run the program.

Here are few more simple programs, sos_test2 tests that autoloading works:

```
-module(sos_test2).
-doc("").
-export([main/0]).

main() ->
    X = lists:reverse("Hellow world"),
    sos:write([X,"\n"]).
```

Thus:

```
> sos sos_test2
dlrow wolleH
```
sos_test3 is an Erlang program which copies everything it sees on standard input to standard output (This is how to write a unix pipe process)

```
-module(sos_test3).
-doc(none).
-export([main/0]).
-import(sos, [read/0, write/1]).

main() ->
    loop().

loop() ->
    case read() of
    eof ->
        true;
    {ok, X} ->
        write(X),
        loop()
    end.
```

For example:

```
> cat sos.html | cksum
2409969489  23031
> cat sos.html | sos sos_test3 | cksum
2409969489  23031
sos_test3 tests error handling:
-module(sos_test4).
-doc(none).
-export([main/0]).

main() ->
    sos:write("I will crash now
"),
    i = 2,
    sos:write("This line will not be printed
")
```

So for example:

```
> sos sos_test4
I will crash now
{stopping_system,{{badmatch,2},{sos_test4,main,[]}}}
6.3  sos

The shell script **sos** which starts everything is as follows:

```bash
#!/bin/sh
erl -boot /home/joe/erl/example_programs-2.0/examples-2.0/sos -environment 'printenv' -load
```

This just starts an erlang system from the bootfile **sos.boot**, the bootfile and the **sos** shell script are created by evaluating *make_scripts*/0

```erl
make_scripts() ->
    {ok, Cwd} = file:get_cwd(),
    Script =
        [{script, "sos", "1.0"},
         [{preLoaded, [init, erl_prim_loader]},
         {progress, preloaded},
         {path, 
             [".",
              Cwd,
              "$ROOT/lib/" ++ lib_location(kernel) ++ "/ebin",
              "$ROOT/lib/" ++ lib_location(stdlib) ++ "/ebin"]},
         {primLoad, 
          [erl_open_port, 
           erlang, 
           error_handler, 
           sos 
           ]},
         {kernel_load_completed},
         {progress, kernel_load_completed},
         {progress, started},
         {apply, [sos, main, []]}
         ]},
    file:write_file("sos.boot", term_to_binary(Script)),
    {ok, Stream} = file:open("sos", write),
    io:format(Stream, 
        "#!/bin/sh 
        erl -boot /home/joe/erl/example_programs-2.0/examples-2.0/sos 
        -environment 'printenv' 
        -load 
        
        " 
        ),
    file:close(Stream),
    unix:cmd("chmod a+x sos"),
    true.
```
lib_location(Lib) ->
    filename:basename(code:lib_dir(Lib)).

make_scripts/0 must be evaluated from within the Erlang development environment since it calls code filename and file which are not available to sos programs. Exactly how this function works is beyond the scope of this paper, the gory details can be found in the full version of the Erlang/OTP documentation.

The important point to note about the script is the last line, {apply, {sos,main,[]}} this is function which is called when the system is started.

### 6.4 The sos main program

When the system is started using the bootfile described in the last section the function sos:main() is evaluated and when this terminates the system will halt:

main() ->
    make_server(io,
        fun start_io/0, fun handle_io/2),
    make_server(code,
        const([init,erl_prim_loader]),
        fun handle_code/2),
    make_server(error_logger,
        const(0), fun handle_error_logger/2),
    make_server(halt_demon,
        const([]), fun handle_halt_demon/2),
    make_server(env,
        fun start_env/0, fun handle_env/2),
    load_module(error_handler),
    Mod = get_module_name(),
    load_module(Mod),
    run(Mod).

This starts five servers (io, code...), loads the error handler, finds the name of the module which is to be run, loads this module and then runs the code in the module.

run(Mod) spawn links Mod:main() and waits for it to terminate. When it terminates stop_system is called:
Main starts a lot of different servers, before going into the details of how the individual servers work we explain the generic framework that is used to build client-servers.

### 6.5 Server support

To create a server we call `make_server(Name, Fun1, Fun2)`. *Name* is the global name of the server, *Fun1()* is expected to return *State1* the initial state of the server. Evaluating `Fun2(State, Query)` should return `{Reply, State1}` if called with a remote procedure call or simply `State1` if called with a cast. `make_server` is:

```erlang
make_server(Name, FunD, FunH) ->
  make_global(Name,
              fun() ->
                Data = FunD(),
                server_loop(Name, Data, FunH)
              end).
```

The server loop is simple:

```erlang
server_loop(Name, Data, Fun) ->
  receive
    {rpc, Pid, Q} ->
      case (catch Fun(Q, Data)) of
        {EXIT, Why} ->
          Pid ! {Name, exit, Why},
          server_loop(Name, Data, Fun);
        {Reply, Data1} ->
          Pid ! {Name, Reply},
          server_loop(Name, Data1, Fun)
      end;
    {cast, Pid, Q} ->
      case (catch Fun(Q, Data)) of
        {EXIT, Why} ->
          exit(Pid, Why),
```
To query the server we use \texttt{rpc} (short for \textit{Remote Procedure Call}) which is:

\begin{verbatim}
Rpc(Name, Q) ->
  Name ! \{rpc, self(), Q\},
  receive
    \{Name, Reply\} ->
      Reply;
    \{Name, exit, Why\} ->
      exit(Why)
  end.
\end{verbatim}

Note how the code in the server loop and \texttt{rpc} interacts. The handler fun in the server is protected by a \texttt{catch} if an exception is raised in the server a \{\texttt{Name, exit, Why}\} message is sent back to the client. If this message is received by the client an exception is raised by evaluating \texttt{exit(Why)} in the client.

The net effect of this is to raise an exception in the client. Note that in the case where the client sent a query to the server that the server could not process the server continues with its old state.

Thus remote procedure calls function as \textit{transactions} as far as the server is concerned. Either they work completely, or the server is rolled back to the state it was in before the remote procedure call was made.

If we just want to send a message to the server and are not interested in the reply we call \texttt{cast/2}:

\begin{verbatim}
Cast(Name, Q) ->
  Name ! \{cast, self(), Q\}.
\end{verbatim}

Amazingly we can change the behaviour of a server by sending it a different fun to use in the server loop:
Recall that when we started the server the initial data structure is often a constant. We can define \texttt{const(C)} which returns a function which when evaluated evaluates to C.

\[
\text{const(C)} \rightarrow \text{fun}() \rightarrow C \end{equation}

Now we will turn our attention to the individual servers.

\section{Code server}
Recall that the code server was started by evaluating:

\[
\text{make_server(code, const([init,erl_prim_loader]), fun handle_code/2)},
\]

\texttt{load_module(Mod)} is implemented as a remote procedure call to the code server:

\[
\text{load_module(Mod)} \rightarrow \text{rpc(code, \{load_module, Mod\})}.
\]

The global state of the code server is simple [\texttt{Mod}] that is, a list of all modules which have been loaded (the initial value of this is [\texttt{init, erl_prim_loader}] - these are \textit{preloaded} i.e. compiled into the Erlang run-time systems kernel); for details read the Erlang/OTP documentation).

The server handler function \texttt{handle_code/2} is thus:

\[
\text{handle_code(\{load_module, Mod\}, Mods) \rightarrow}
\]

\[
\begin{array}{ll}
\text{case member(Mod, Mods) of} \\
\text{true} \rightarrow \\
\{\text{already_loaded, Mods}\}; \\
\text{false} \rightarrow \\
\text{case primLoad(Mod) of} \\
\text{ok} \rightarrow \\
\{\{\text{ok,Mod}, [\text{Mod}|\text{Mods}]\}; \\
\text{Error} \rightarrow \\
\{\text{Error, Mods}\}
\end{array}
\]
and `primLoad` does the loading:

```erlang
primLoad(Module) ->
    Str = atom_to_list(Module),
    case erl_prim_loader:get_file(Str ++ ".jam") of
        {ok, Bin, FullName} ->
            case erlang:load_module(Module, Bin) of
                {module, Module} -> ok;
                {_, _} ->
                    {error, wrong_module_in_binary};
                Other ->
                    {error, {bad_object_code, Module}}
            end;
        error ->
            {error, {cannot_locate, Module}}
    end.
end.
```

### 6.7 Error_logger

`log_error(What)` logs the error `What` on standard output, this is implemented as a cast:

```erlang
log_error(Error) -> cast(error_logger, {log, Error}).
```

the corresponding server handler function being:

```erlang
handle_error_logger({log, Error}, N) ->
    erlang:display({error, Error}),
    {ok, N+1}.
```

Note that the global state of the error handler is an integer `N` meaning the total number of errors which have occurred.
6.8 Halt demon

The halt demon is called when the system is halted. Evaluating on_halt(Fun) sets up a condition such that Fun() will be evaluated when the system is halted. Halting the system is done by calling the function stop_system():

```
on_halt(Fun) -> cast(halt_demon, {on_halt, Fun}).
stop_system(Why) -> cast(halt_demon, {stop_system, Why}).
```

The server handler code for this is:

```
handle_halt_demon({on_halt, Fun}, Funs) ->
    {ok, [Fun|Funs]};
handle_halt_demon({stop_system, Why}, Funs) ->
    case Why of
        normal -> true;
        _ -> erlang:display({'stopping_system',Why})
    end,
    map(fun(F) -> F() end, Funs),
    erlang:halt(),
    {ok, []}.
```

6.9 IO server

The IO server allows access to STDIO, read() reads a line from standard input and write(String) writes a string to standard output:

```
read() -> rpc(io, read).
write(X) -> rpc(io, {write, X}).
```

The initial state of the IO server is obtained by evaluating start_io():

```
start_io() ->
    Port = open_port({fd,0,1}, [eof, binary]),
    process_flag(trap_exit, true),
    {false, Port}.
```

And the IO handler is:
handle_io(read, {true, Port}) ->
    {eof, {true, Port}};
handle_io(read, {false, Port}) ->
    receive
        {Port, {data, Bytes}} ->
            {{ok, Bytes}, {false, Port}};
        {Port, eof} ->
            {eof, {true, Port}};
        {'EXIT', Port, badsig} ->
            handle_io(read, {false, Port});
        {'EXIT', Port, Why} ->
            {eof, {true, Port}}
    end;
handle_io({write, X}, {Flag, Port}) ->
    Port ! {self(), {command, X}},
    {ok, {Flag, Port}}.

The state of the IO server is {Flag, Port} where Flag is true if eof has been encountered, otherwise false.

6.10 Environment server

The function env(E) is used to find the value of the environment variable E:

    env(Key) -> rpc(env, {lookup, Key}).

The server is:

    handle_env({lookup, Key}, Dict) ->
        {lookup(Key, Dict), Dict}.

The initial state of the server is found by evaluating:

    start_env() ->
        Env = case init:get_argument(environment) of
            {ok, [L]} ->
                L;
            error ->
6.11 \textit{Global process support}

We need a few routines for keeping processes alive and registering global names.

\texttt{\texttt{keep\_alive(name, Fun)} makes a registered process called Name it is started by evaluating Fun() and if the process dies it is automatically restarted.}

\begin{verbatim}
keep_alive(Name, Fun) ->
    Pid = make_global(Name, Fun),
    on_exit(Pid,
        fun(Exit) -> keep_alive(Name, Fun) end).
\end{verbatim}

\texttt{\texttt{make\_global(Name, Fun)} checks if there is a global process with the registered name Name. If there is no process it spawns a process to evaluate Fun() and registers it with the name Name.}

\begin{verbatim}
make_global(Name, Fun) ->
    case whereis(Name) of
        undefined ->
            Self = self(),
            Pid = spawn_fun(fun() ->
                make_global(Self,Name,Fun)
            end),
            receive
                {Pid, ack} ->
                    Pid
            end;
        Pid ->
            Pid
    end.
\end{verbatim}
6.12 Support for processes

spawn_fun(Fun) and spawn_link(Fun) spawns (and links) a new process and evaluates Fun() within the newly spawned process.

\[
\text{spawn_fun}\{\text{'fun'}, \text{Mod}, \text{Arity}, \text{Chksum}, \text{Env}\}\to\\
\text{spawn}(?\text{MODULE}, \text{internal\_call},\\
[\text{Mod}, \text{Arity}, \text{Chksum}, [], \text{Env}]).
\]

\[
\text{spawn_link\_fun}\{\text{'fun'}, \text{Mod}, \text{Arity}, \text{Chksum}, \text{Env}\}\to\\
\text{spawn}(?\text{MODULE}, \text{internal\_call},\\
[\text{Mod}, \text{Arity}, \text{Chksum}, [], \text{Env}]).
\]

You are not expected to understand how the above two routines work!

on_exit(Pid, Fun) links to Pid. If Pid dies with reason Why then Fun(Why) is evaluated:

\[
\text{on\_exit}(\text{Pid}, \text{Fun})\to\\
\text{spawn\_fun}(\text{fun}()\to\\
\text{process\_flag}(\text{trap\_exit}, \text{true}),\\
\text{link}(\text{Pid}),\\
\text{receive}\\
\{\text{'EXIT'}, \text{Pid}, \text{Why}\}\to\\
\text{Fun}(\text{Why})\\
\text{end}\\
\text{end}).
\]

every(Pid, Time, Fun) links to Pid then every Time Fun() is evaluated. If Pid exits, this process stops.

\[
\text{every}(\text{Pid}, \text{Time}, \text{Fun})\to\\
\text{spawn\_fun}(\text{fun}()\to\\
\text{process\_flag}(\text{trap\_exit}, \text{true}),\\
\text{link}(\text{Pid}),\\
\text{every\_loop}(\text{Pid}, \text{Time}, \text{Fun})\\
\text{end}).
\]

every_loop(Pid, Time, Fun) ->

receive
6.13 Utilities

get_module_name() gets the module name from the command line.

```erlang
get_module_name() ->
    case init:get_argument(load) of
        {ok, [[Arg]]} ->
            module_name(Arg);
        error ->
            fatal({missing, '-load Mod'})
    end.
```

Finally a few small utilities, such as might be found in lists, are included here so as to make sos "self contained" ...

```erlang
lookup(Key, [{Key,Val}|[]]) -> {found, Val};
lookup(Key, [_|T]) -> lookup(Key, T);
lookup(Key, []) -> not_found.

member(X, [X|_]) -> true;
member(H, [_|T]) -> member(H, T);
member(_, []) -> false.

map(F, [H|T]) -> [F(H)|map(F, T)];
map(F, []) -> [].

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

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

module_name(Str) ->
    case (catch list_to_atom(Str)) of
```
And something to report errors:

    fatal(Term) ->
        log_error({fatal, Term}),
        stop_system({fatal, Term}).

6.14 Error_handler

Sos uses its own error handler (not the standard handler) this is as follows:

    undefined_function(sos, F, A) ->
        erlang:display({'error_handler', undefined_function, sos,F,A}),
        exit(oops);
    undefined_function(M, F, A) ->
        case sos:load_module(M) of
            {ok, M} ->
                case erlang:function_exported(M,F,length(A)) of
                    true ->
                        apply(M, F, A);
                    false ->
                        sos:stop_system({undef,{M,F,A}})
                end;
            {ok, Other} ->
                sos:stop_system({undef,{M,F,A}});
            already_loaded ->
                sos:stop_system({undef,{M,F,A}});
            {error, What} ->
                sos:stop_system({load, error, What})
        end.
    undefined_global_name(Name, Message) ->
        exit({badarg,{Name,Message}}).
We bomb if the undefined function is in sos itself.
If the error handler is called for any other module we call sos:load_module
to try and load the module.

```javascript
undefined_global_name(Name, Message) ->
   exit({badarg,{Name,Message}}).
```