

## Erlang Shell

erl -mode Mode - Mode is embedded or interactive.

Help() - help on erlang shell  
c(Module) - compile module  
b() - show all variables  
f() - remove all variable bindings  
i() - list processes  
memory() - print memory information  
q() - quit  
regs() - registered processes  
rr(Module) - load record definitions from module  
rd(name, {field1, field2, ...}) - define module  
rl() - list all record definitions  
pwd() - return current working directory  
cd(Dir) - change working directory  
appmon:start() - application monitor

pman:start() - start process manager (GUI)  
toolbar:start() - start toolbar  
tv:start() - start ETS table browser from erl shell

+P x set maximum number of processes

==	equal to	/=	not equal to
==:	exactly equal to	==/	exactly not equal to
=<	less than or equal	<	less than
>=	greater than or equal	>	greater than

## Booleans

true, false atoms.

not	unary logical not	or	logical or
and	logical and	xor	logical xor
andalso	short-circuit and	orelse	short-circuit or

## Numbers

base#value - integer with given base

## Strings

A = "hello world".  
B = [104,101,108,108,111,32,119,111,114,108,100].  
C = [\$h,\$e,\$l,\$l,\$o,\$ , \$w,\$o,\$r,\$l,\$d].  
D = <<"this string consumes one byte per character">>.

\$char - ASCII value of the character char.

String module:

len(String) -> Length	Returns the number of characters in the string
equal(String1, String2) -> bool	Tests whether two strings are equal

concat(String1, String2) -> String3	Concatenates two strings to form a new string.
chr(String, Character) -> Index	Returns the index of the first/last occurrence of Character in String. 0 is returned if Character does not occur.
rchr(String, Character) -> Index	
str(String, SubString) -> Index	Returns the position where the first/last occurrence of SubString begins in String. 0 is returned if SubString does not exist in String.
rstr(String, SubString) -> Index	
substr(String, Start) -> SubString	Returns a substring of String, starting at the position Start, and ending at the end of the string or at length Length.
substr(String, Start, Length) -> SubString	
tokens(String, SeparatorList) -> Tokens	Returns a list of tokens in String, separated by the characters in SeparatorList.
join(StringList, Separator) -> String	Returns a string with the elements of StringList separated by the string in Separator.
strip(String) -> Stripped	Returns a string, where leading and trailing blanks have been removed.
sub_string(String, Start) -> SubString	
sub_string(String, Start, Stop) -> SubString	Returns a substring of String, starting at the position Start to the end of the string, or to and including the Stop position.
to_float(String) -> {Float,Rest}   {error,Reason}	Convert string to float. Remaining characters in the string after the float are returned in Rest.
to_integer(String) -> {Int,Rest}   {error,Reason}	Convert string to integer. Remaining characters in the string after the integer are returned in Rest.
to_lower(String) -> Result	Convert case.
to_lower(Char) -> CharResult	
to_upper(String) -> Result	
to_upper(Char) -> CharResult	

## Modules

```
-module(Name).  
-vsn(Version).           % define module version, MD5 checksum by default.  
-export([fun/arity, ..]).  
-compile(export_all)
```

or

```
c(Module,[export_all]).
```

```
$ erlc module.erl
```

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

To add another include directory: c(Module, [{i, Dir}]).

Module:Function(Arg1, ..., ArgN) - function call  
-import(Module, [f/Arity]). - import function, so that doesn't need Module prefix.

code:get_path()	get code search path
code:add_patha(Path)	add new path to the beginning of the list
code:add_pathz(Path)	add new path to the end of the list

Add path when starting shell:

```
erl -pa Path
```

```
erl -pz Path
```

### Block expressions

```
begin
    Expr1,
    ...
    ExprN
end
```

### Conditionals

```
case cond-expr of
    Pattern1 -> expr1, expr2, .. ;
    Pattern2 -> expr1, expr2, .. ;
    ... ;
    Patternn -> expr1, expr2, ..
end
```

```
if
    Guard1 -> expr11, expr12, .. ;
    Guard2 -> expr21, expr22, .. ;
    .. ;
    Guardn -> exprn1, exprn2, ..
end
```

### Functions

Module:Function(Arg1, ..) - function call

apply(Module, Function, ArgList) - executes given function, returns the result

```
f(arg1, .., argn) -> expr0;
f(arg1, .., argn) ->
    expr1,
    expr2,
    ..
    exprn.
```

f(arg1, .., argn) when Guards -> expr.

Guards may be separated by comma (conjunction, logical AND) or semicolon (disjunction, logical OR).

### Pattern matching

A = {square, 3}	Binds tuple to A.
{square, 3} = A	True.
{square, 0} = A	Error. 0 doesn't match 2 <sup>nd</sup> argument of right hand side tuple.
{square, W} = A	ok. 1 <sup>st</sup> argument matches. W is unbound, so binds 3 to W.
B = {rect, 5, 5}	ok. Binds tuple to B.
{rect, X, X} = B	ok. 1 <sup>st</sup> argument matches. X is unbound, so binds 5 to X. 3 <sup>rd</sup> argument is already bound, so it is compared with 3 <sup>rd</sup> argument of the right hand side tuple (true).
C = {3, 4}	Binds tuple to C.

{Y, Y} = C	Error. Y is unbound, so binds 3 to Y. 2 <sup>nd</sup> argument is already bound but doesn't match 2 <sup>nd</sup> argument of right hand side tuple.
[H   T] = [1,2,3]	Binds head to H and tail to T (H is 1, T is [2, 3]).

```
f("prefix" ++ Str) -> ...
```

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

### Binary data

To create or pattern-match:

Bin = <<E1, E2, .., En>>

<<E1, E2, .., En>> = Bin

where each expression is Expr:Size/Type. Size and type are optional. Size is specified in bits. Type is a list of type specifiers, separated by hyphens:

type	integer, float, binary, byte, bits, bitstring
sign	signed, unsigned (the default). 1 <sup>st</sup> bit determined the sign.
endianess	big (default), little, native
unit:val	define number of bits used by the entry as val*Size

Bin = <<16:8, 1:8, 0:3>> % Bin is a list of three integers, encoded on 8/8/3 bits respectively.

<<0:1/integer-unit:8, W:1/integer-unit:8, Rest/bitstring>> = Bin % Bind integers of 8\*1 bits to Q and W, capture remainder in Rest.

<<X:8/bitstring,Y:1/bitstring, Rest2/bitstring>> = Bin % Bind 8 bit values to Q and W, capture remainder in Rest.

### Tail Recursion

```
sum(List) -> sum_acc(List, 0).
sum_acc([], Sum) -> Sum;
sum_acc([Head|Tail], Sum) -> sum_acc(Tail, Head+Sum).
```

### Exception Handling

Using try .. catch:

```
try Exprs of
    Pattern1 [when Guard1] ->
        ExprBody1;
    Pattern2 [when Guard2] ->
        ExprBody2
catch
    [Class1:]ExceptionPattern1
        [when ExceptionGuardSeq1] ->
```

<pre>        ExceptionBody1; [ClassN:]ExceptionPatternN     [when ExceptionGuardSeqN] -&gt;         ExceptionBodyN end</pre>		<table><tr><td>list_to_integer/1</td><td rowspan="3">round / truncate, return integer</td></tr><tr><td>trunc/1</td></tr><tr><td>round/1</td></tr></table>	list_to_integer/1	round / truncate, return integer	trunc/1	round/1														
list_to_integer/1	round / truncate, return integer																			
trunc/1																				
round/1																				
<p>Error classes are: error - runtime_errors: if_clause, badmatch, badarg, undef, badarith. throw - generated by explicit call to throw, e.g. throw({'EXIT', ..}) exit - raised by calling exit/1.</p> <p>erlang:get_stacktrace() returns stacktrace of the latest thrown exception.</p> <p>Using primitive catch: catch expression If expression evaluates correctly, returns the value of the expressions; returns the tuple {'Exit', Error} if a tuntime error occurs.</p> <p>Raising exceptions: throw(SomeTerm) exit(Reason) erlang:error(Reason) erlang:raise(Class, Reason, Stacktrace) % rethrow exception after catching it</p>		<h3>Creating Processes</h3> <table><tr><td>spawn(Module, Function, ArgsList) -&gt; Pid</td><td>Creates new process by application of named function.</td></tr><tr><td>spawn(Fun) -&gt; Pid</td><td>Creates new process by application of Fun.</td></tr><tr><td>self()-&gt; Pid</td><td>Returns the pid of the calling process.</td></tr><tr><td>register(Alias, Pid)</td><td>Give process an alias which is available for all other processes.</td></tr><tr><td>unregister(Pid)</td><td>Removes registered process name.</td></tr><tr><td>registered()</td><td>Returns list of registered aliases.</td></tr><tr><td>whereis(Alias)</td><td>Returns associated pid.</td></tr></table>	spawn(Module, Function, ArgsList) -> Pid	Creates new process by application of named function.	spawn(Fun) -> Pid	Creates new process by application of Fun.	self()-> Pid	Returns the pid of the calling process.	register(Alias, Pid)	Give process an alias which is available for all other processes.	unregister(Pid)	Removes registered process name.	registered()	Returns list of registered aliases.	whereis(Alias)	Returns associated pid.				
spawn(Module, Function, ArgsList) -> Pid	Creates new process by application of named function.																			
spawn(Fun) -> Pid	Creates new process by application of Fun.																			
self()-> Pid	Returns the pid of the calling process.																			
register(Alias, Pid)	Give process an alias which is available for all other processes.																			
unregister(Pid)	Removes registered process name.																			
registered()	Returns list of registered aliases.																			
whereis(Alias)	Returns associated pid.																			
<h3>Lists, Tuples</h3> <p>hd/1 - returns head tl/1 - returns tail length/1 - length of the list tuple_size/1 - number of elements of a tuple element/1 - returns nth element of a tuple setelement/3 - replaces an element in a tuple, returns new tuple, e.g. setelement(2, MyTuple, monday).</p> <p>Expr1 ++ Expr2 list concatenation operator Expr1 - Expr2 list subtraction operator</p>		<h3>Error Trapping</h3> <p>Process is terminated if it receives nonnormal exit signal and is not trapping signals.</p> <p>process_flag(trap_exit, true) - enable trapping. The caller will receive exit signals {'EXIT', Pid, Reason}.kill cannot be intercepted.</p> <p>Pid = spawn_link(Module, Function, ArgsList) - spawn process and link atomically link(Pid) unlink(Pid)</p> <p>exit(Reason) - will terminate and send {'EXIT', Pid, Reason} exit(Pid, Reason) - send exit signal to other process, Reason can be `normal`, `kill` or Other.</p> <p>Reference = erlang:monitor(process, Pid) - create unidirectional monitor towards Pid erlang:demonitor(Reference, [flush]) will receive {'DOWN',Reference,process,Pid,Reason}</p> <p>spawn_monitor(Module, Function, ArgsList)</p>																		
<table><tr><td>is_binary</td><td rowspan="4">type checks</td></tr><tr><td>is_atom</td></tr><tr><td>is_boolean</td></tr><tr><td>is_tuple</td></tr><tr><td>atom_to_list/1</td><td rowspan="3">convert atoms to strings and back</td></tr><tr><td>list_to_atom/1</td></tr><tr><td>list_to_existing_atom/1</td></tr><tr><td>list_to_tuple/1</td><td rowspan="2">convert between the two data types</td></tr><tr><td>tuple_to_list/1</td></tr><tr><td>float/1</td><td>create a float from integer</td></tr><tr><td>list_to_float/1</td><td>create float from string</td></tr><tr><td>float_to_list/1 integer_to_list/1</td><td>convert float or integer to string and back</td></tr></table>		is_binary	type checks	is_atom	is_boolean	is_tuple	atom_to_list/1	convert atoms to strings and back	list_to_atom/1	list_to_existing_atom/1	list_to_tuple/1	convert between the two data types	tuple_to_list/1	float/1	create a float from integer	list_to_float/1	create float from string	float_to_list/1 integer_to_list/1	convert float or integer to string and back	<h3>Message Passing</h3> <pre>Pid ! Message Pid1 ! Pid2 ! Pid3 ! Message</pre> <h3>Input/Output</h3> <p>io:get_line(Prompt) - read line from stdin io:get_chars(Prompt, NumOfChars) - read a specified number of characters from stdin io:read(Prompt) - read Erlang term (e.g. atom) from stdin io:write(Term) - write Erlang term io:format(FormatString, ArgList) - formatted output Full control sequence is: ~F.P.PadC, where F is the field width, P is precision, Pad is the padding character, C is the control character. Control characters: ~c an ASCII character code</p>
is_binary	type checks																			
is_atom																				
is_boolean																				
is_tuple																				
atom_to_list/1	convert atoms to strings and back																			
list_to_atom/1																				
list_to_existing_atom/1																				
list_to_tuple/1	convert between the two data types																			
tuple_to_list/1																				
float/1	create a float from integer																			
list_to_float/1	create float from string																			
float_to_list/1 integer_to_list/1	convert float or integer to string and back																			

~f float with six decimal places  
~e float to be printed in scientific notation, six digits  
~w any term printed in standard syntax  
~p same as ~w, but in pretty printing mode  
~W, ~P same as above, but eliding structure at a depth of 3. Takes an extra argument in the ArgList indicating max depth  
~B print integer to base 10

## Receiving Messages

```
receive
  Pattern1 when Guard1 -> exp11, ..., exp1n;
  Pattern2 when Guard2 -> exp21, ..., exp2n;
  ...
  Other                -> expn1, ..., expnn
after
  Timeout -> exp1, ..., expn
end
```

The “after” clause and guards are optional. Timeout is in milliseconds or infinity atom. If using “after”, need to take care of synchronisation (flushing).  
The return value of the receive clause is the return value of the last evaluated expression in the body executed (expin).

```
f(Pid) ->
  receive {Pid, {a, b}} -> g(b)
end.
```

## Benchmarking

timer:tc(Module, Function, Arguments) - returns {Microseconds, Status} tuple.

## Records

Defining new record type:

```
-record(name, {field1, [=default], field2 [=default], ...}).
```

Creating a record:

```
Var = #recordname{attr1=..., attr2=..., ...}
```

undefined atom is used if no value nor default is provided.

Access field of a record

```
RecordVar#recordname.fieldname
```

Applying pattern matching:

```
fun(#person{age=Age} = P) -> P#person{age=Age+1}.
```

record\_info(fields, recType) - return list of field names  
record\_info(size, recType) - return the size of the record  
#recType.fieldName - return position of field in the record (tuple).  
is\_record(Term, RecordTag) - return true if Term is a record tuple.

## Preprocessor directives and Macros

-define(Name, Replacement).	define macro
-define(Name(Var1, Var2, ..., VarnN), Replacement)	define macro with parameters. Use ??Varn to get macro's argument as a string.

-undef(Name)	remove macro
-ifdef(Flag).	Conditional compilation
-ifndef(Flag).	
-else.	
-endif.	
-include("filename.hrl").	Include another (header) file.

?Name - use macro

Predefined macros: ?MODULE, ?MODULE\_STRING, ?FILE, ?LINE, ?MACHINE.

Passing flags to compiler:

c(Module,[{d,debug}])). - set debug flag  
c(Module,[{u,debug}])). - unset debug flag.

## Sockets

Sockets may be opened in active or passive mode. In active mode regular erlang messages in the format of {tcp, Socket, Packet} or {udp, Socket, IP, Port, Packet} will be received. In passive mode you need to call recv functions.

gen\_udp module:

open(Port) -> {ok, Socket}   {error, Reason}	
send(Socket, Address, Port, Packet) -> ok   {error, Reason}	
recv(Socket, Length) -> {ok, {Address, Port, Packet}}   {error, Reason}	
recv(Socket, Length) -> {ok, {Address, Port, Packet}}   {error, Reason} -> {ok, {Address, Port, Packet}}   {error, Reason}	
recv(Socket, Length, Timeout) -> {ok, {Address, Port, Packet}}   {error, Reason}	
close(Socket) -> ok   {error, Reason}	
controlling_process(Socket, Pid) -> ok	Assigns a new controlling process Pid to Socket.

Sender:

```
{ok, Socket} = gen_udp:open(1000).
gen_udp:send(Socket, {127,0,0,1}, 1500, "Hello").
gen_udp:close(Socket).
```

Receiver:

```
{ok, Socket} = gen_udp:open(1111),
  receive
    {udp, Socket, _, 2222, Msg} -> io:format("Got: ~p~n", [Msg])
  end,
  gen_udp:close(Socket).
```

gen\_tcp module:

connect(Address, Port, Options) -> {ok, Socket}   {error, Reason}	
---	--

connect(Address, Port, Options, Timeout) -> {ok, Socket}   {error, Reason}	
listen(Port, Options) -> {ok, ListenSocket}   {error, Reason}	
accept(ListenSocket) -> {ok, Socket}   {error, Reason}	
accept(ListenSocket, Timeout) -> {ok, Socket}   {error, Reason}	
send(Socket, Packet) -> ok   {error, Reason}	
recv(Socket, Length) -> {ok, Packet}   {error, Reason}	
recv(Socket, Length, Timeout) -> {ok, Packet}   {error, Reason}	
close(Socket) -> ok   {error, Reason}	

## Debugging

debugger:start() - start the debugger from erl shell  
c(Module, [debug\_info]) or erlc +debug\_info Module.erl - compile for debugging

## Funs and Higher-Order Functions

Name = fun(Args) -> ... end. - binds function to a named variable

Functions as arguments:

```
foreach(F, []) -> ok;  
foreach(F, [X|Xs]) -> F(X), foreach(F, Xs).
```

Functions as results:

```
times(X) -> fun(Y) -> X*Y end.
```

Higher-Order functions in lists module:

all(Pred, List)	
any(Pred, List)	
dropwhile(Pred, List)	
filter(Pred, List)	
foldl(Fun, Acc0, List) -> Acc1	
foldr(Fun, Acc0, List) -> Acc1	
map(Fun, List1) -> List2	
partition(Pred, List) -> {Satisfying, NotSatisfying}	

## List Comprehensions

```
[ Expression || Generators, Guards, Generators, ... ]
```

Generators has the form Pattern <- List.  
Guards should give true or false.  
Expression specifies what the elements of the result will look like.

Example:

```
[X+1 || X <- [1,2,3], X rem 2 == 0].           % gives [3]  
A=[{X,Y} || X <- [2,3,4], Y <- [1, 2, 3], X /= Y]. % gives [{2,1},{2,3},{3,1},{3,2},{4,1},{4,2},{4,3}]
```

## Process dictionary

Please note that using process dictionary is highly discouraged.

put(Key, Val) -> OldVal   undefined	Adds new key-value pair to the process dictionary. Replaces and returns old value (if existed).
get(Key) -> Val   undefined	Returns the value Val associated with Key or undefined.
get() -> [{Key, Val}]	Returns the process dictionary as a list of {Key, Val} tuples.
get_keys(Val) -> [Key]	Returns a list of keys which are associated with the value Val.
erase() -> [{Key, Val}]	Returns the process dictionary and deletes it.
erase(Key) -> Val   undefined	Returns the value Val associated with Key and deletes it.

## ETS Tables

Erlang Term Storage is a built-in term storage that provides constant or logarithmic access time.

set	each key can occur only once
ordered set	same as set, elements can be traversed following the lexicographical order on the keys
bag	allows multiple entries for the same key
duplicate bag	allows duplicated elements

Functions in the ets module:

new(Name, OptsList) -> tid()   atom()	create a new table, returns table id
delete(Tab) -> true	delete table
insert(Tab, ObjectOrObjects) -> true	insert elements, overwrites existing one
lookup(Tab, Key) -> [Object]	return a list of all objects with the key Key
first(Tab) -> Key   '\$end_of_table'	return the first key or '\$end_of_table'
next(TabId, Key1) -> Key2   '\$end_of_table'	return the next key or '\$end_of_table'
last(Tab, Key)	return the last key or '\$end_of_table'
safe_fixtable(Tab, true false)	fix the table for safe traversal
match(Tab, Pattern) -> [Match]	match the objects against pattern. Pattern may contain erlang terms, '.' which matches any term and \$N variables (N=0,1,...), e.g. ets:match(countries, {'\$1', ireland, '_'}) returns [[sean], [chris]]
select(Tab, MatchSpec) -> [Match]	match the objects using a MatchSpec. MatchSpec is a list of one or more tuples of arity 3. The first element is a pattern such as {'\$1','\$2','\$3'}. The 2nd element should be a list of 0 or more guards tests 3rd element should be a list containing a description of the value to actually return. Example: ets:select(countries, [{{'\$1','\$2','\$3'}, [{'/=';\$3',cook}],[['\$2','\$1']}]}) returns [[ireland,sean], [ireland,chris]].

<code>select(Tab, MatchSpec, Limit) -&gt; {[Match], Continuation}   '\$end_of_table'</code>	limit number of matching objects. Returns {[Match], Continuation} or '\$end_of_table'. Continuation term can be used in subsequent call to ets:select/1.
<code>select(Continuation) -&gt; {[Match], Continuation}   '\$end_of_table'</code>	should be used in conjunction with ets:select/3, returns {[Match], Continuation]} or '\$end_of_table'.
<code>fun2ms(LiteralFun) -&gt; MatchSpec</code>	return match specification usable for ets:match Example: M = ets:fun2ms(fun({Name,Country,Job}) when Job /= cook -> [Country,Name] end).
<code>tab2file(Tab, FileName) -&gt; ok   {error, Reason}</code>	dump a table into file
<code>file2tab(FileName) -&gt; {ok, Tab}   {error, Reason}</code>	read a table dump
<code>tab2list(Tab) -&gt; [Object]</code>	returns a list of all elements

OptsList can include:

<code>set   ordered_set   bag   duplicate_bag</code>	ETS table type specifier, set by default.
<code>{keypos, Pos}</code>	Defined key position (by default 1st element of tuple). Use #RecordType.KeyField when creating table for records.
<code>public   protected   private</code>	Defines access rights. Public table is readable/writable by all processes; private is readable/writable only by the process that owns the table. Protected table is readable/writable for the owner, but other processes can only read it (the default).
<code>named_table</code>	Table name is statically registered and can be used to reference the table in ETS operations.

## Dets Tables

Provide file-based, persistent storage. Supported table types are `set`, `bag`, `duplicate_bag`.

`dets:open_file(Name, OptsList)` - opens or creates Dets a table, returns {ok, Name} or {error, Reason}.

OptsList can include:

`{auto_save, Interval}` - specifies flush interval in milliseconds or 'infinity'; 3 min by default.

`{file, FileName}` - overrides the default filename and provides path.

`{repair, Bool}` - will trigger repair automatically if needed; if false will trigger {error, need\_repair}.

`{type, TableType}` - can be set, bag or duplicate\_bag.

`{ram_file, Bool}` - stores elements in RAM and spool them to file on dets:sync(Name) call or when closing the table.

`dets:open_file(FileName)` - opens an existing table, returns {ok, Reference} or {error, Reason}.

`dets:close(Name)` - closes table, only the process that opened a table is allowed to close it.

Examples:

```
dets:open_file(food, [{type, bag}], {file, "/tmp/food"})
```

```
dets:insert(food, {italy, pizza})
```

```
dets:close(food)
```

```
{ok, Ref} = dets:open_file("/tmp/food")
```

```
dets:insert(Ref, {italy, pizza})
```

## Standard modules

application	Generic OTP application functions	gen_event	Generic Event Handling Behaviour
array	Functional, extendible arrays	gen_fsm	Generic Finite State Machine Behaviour
calendar	Local and universal time, day-of-the-week, date and time conversions	gen_server	Generic Server Behaviour
dets	A Disk Based Term Storage	get_tcp	Interface to TCP/IP sockets
dict	Key-Value Dictionary	gen_udp	Interface to UDP sockets
erlang	The Erlang BIFs (built-in functions)	lists	List Processing Functions
ets	Built-In Term Storage	math	Mathematical Functions
file	File Interface Module	orddict	Key-Value Dictionary as Ordered List
filename	Filename Manipulation Functions	queue	Abstract Data Type for FIFO Queues
ftp	A File Transfer Protocol client	random	Pseudo random number generation
gb_trees	General Balanced Trees	ssl	Interface Functions for Secure Socket Layer
gb_sets	An Implementation of ordered sets using General Balanced Trees	string	String Processing Functions
http	An HTTP/1.1 client	supervisor	Generic Supervisor Behaviour
httpd	An implementation of an HTTP 1.1 compliant Web server	timer	Timer Functions
io	Standard IO Server Interface Functions		

## Where to get help

`erl -man Module` - display man page for erlang module

[http://www.erlang.org/doc/reference\\_manual/users\\_guide.html](http://www.erlang.org/doc/reference_manual/users_guide.html)

<http://www.erlang.org/faq/faq.html>