Module

```
-module(foo).
-export([sayhi/0]).
% This is a comment.
sayhi() ->
      io:format(“Hello!~n”),
      io:format("Happy Erlanging!~n").
```

Help

```
erl -man <module_name>
```

Function

Function call:

```
Module:Function(Arg1, ...)
```

Function application to list of arguments:

```
apply(Module, Function, ArgList) -
```

Function definition:

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

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}
```

Tuple

```
T = {1.9, 22, 3.99}
element(2, T) = 22
```

List

```
ListA = [1,2,3,4].
[H|T] = ListA.
```

Record

Define a record

```
-record( person, {name, surname}).
```

Create an instance of a record

```
M=#person{name="Marouan",surname="O"}.
```

Access a single field in a record
```
M#person.name.
```
Functional update
```
M#person{name=”foo”}.
```

Maps
```
F1 = #{a => 1, b => 2}.
F3 = F1#{c => xx}. % can be new key
F4 = F1#{c := 3}. % must be existing key
```

Pattern matching
```
A = {square, 3}.
{square, 3} = A. % true
{square, 0} = A. % error
{square, W} = A.
B = {rect, 5, 5}.
{rect, X, X} = B.
C = {3, 4}.
[H | T] = [1,2,3]
#{foo := Foo, bar := Bar} = #{foo => "foo", bar => "bar"}.
#person{name=A, surname=B } = M.
```

Messages
```
Pid ! Message

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

Case
```
case Expression of
      Pattern1 [when Guard1] -> Expr_seq1;
      Pattern2 [when Guard2] -> Expr_seq2;
      …
      Patternn [when Guardn] -> Expr_seqn
end
```

I/O
```
io:format(“ I am ~s~n”, [String]).
```
~n : new line | ~w : standard output

Processes
```
spawn(Fun) -> pid()    spawn(Node, Fun) -> pid()
spawn(Module, Function, Args) -> pid()
spawn(Node, Module, Function, Args) -> pid()
link(PidOrPort) -> true
monitor(Type, Item) -> MonitorRef
```

Monitor
erl -sname observer -hidden -run observer

Types

```
-spec file:open(FileName, Modes) -> {ok, Handle} | {error, Why} when
      FileName  :: string(),
      Modes     :: [Mode],
      Mode      :: read | write | ...
      Handle    :: file_handle(),
      Why       :: error_term().
```

Predefined

```
-type term() :: any().
-type boolean() :: true | false.
-type byte() :: 0..255.
-type char() :: 0..16#10ffff.
-type number() :: integer() | float().
-type maybe_improper_list() :: maybe_improper_list(any(), any()).
-type maybe_improper_list(T) :: maybe_improper_list(T, any()).
-type string() :: [char()].
-type nonempty_string() :: [char(),...].
-type iolist() :: maybe_improper_list(byte() | binary() | iolist(),
                                        binary() | []).
-type module() :: atom().
-type mfa() :: {atom(), atom(), atom()}.
-type node() :: atom().
-type timeout() :: infinity | non_neg_integer().
-type no_return() :: none().
```

User-defined

```
-type onOff() :: on | off.
-type person() :: #person{}.
-type people() :: [person()].
-type name() :: {firstname(), string()}.
-type age() :: integer().
-type dict(Key, Val) :: [{Key, Val}]
```

Exporting

```
-module(a).
-type rich_text() :: [{font(), char()}].
-type font() :: integer().
-export_type([rich_text/0, font/0]).
```

Opaque

```
-module(a).
-opaque rich_text() :: [{font(), char()}].
-export_type([rich_text/0]).
-export([make_text/1, bounding_box/1]).
-spec make_text(string()) -> rich_text().
-spec bbox(rich_text()) -> {Height::integer(), Width::integer()}.
```

Initialize dialyzer

```
dialyzer --build_plt --apps erts kernel stdlib
```