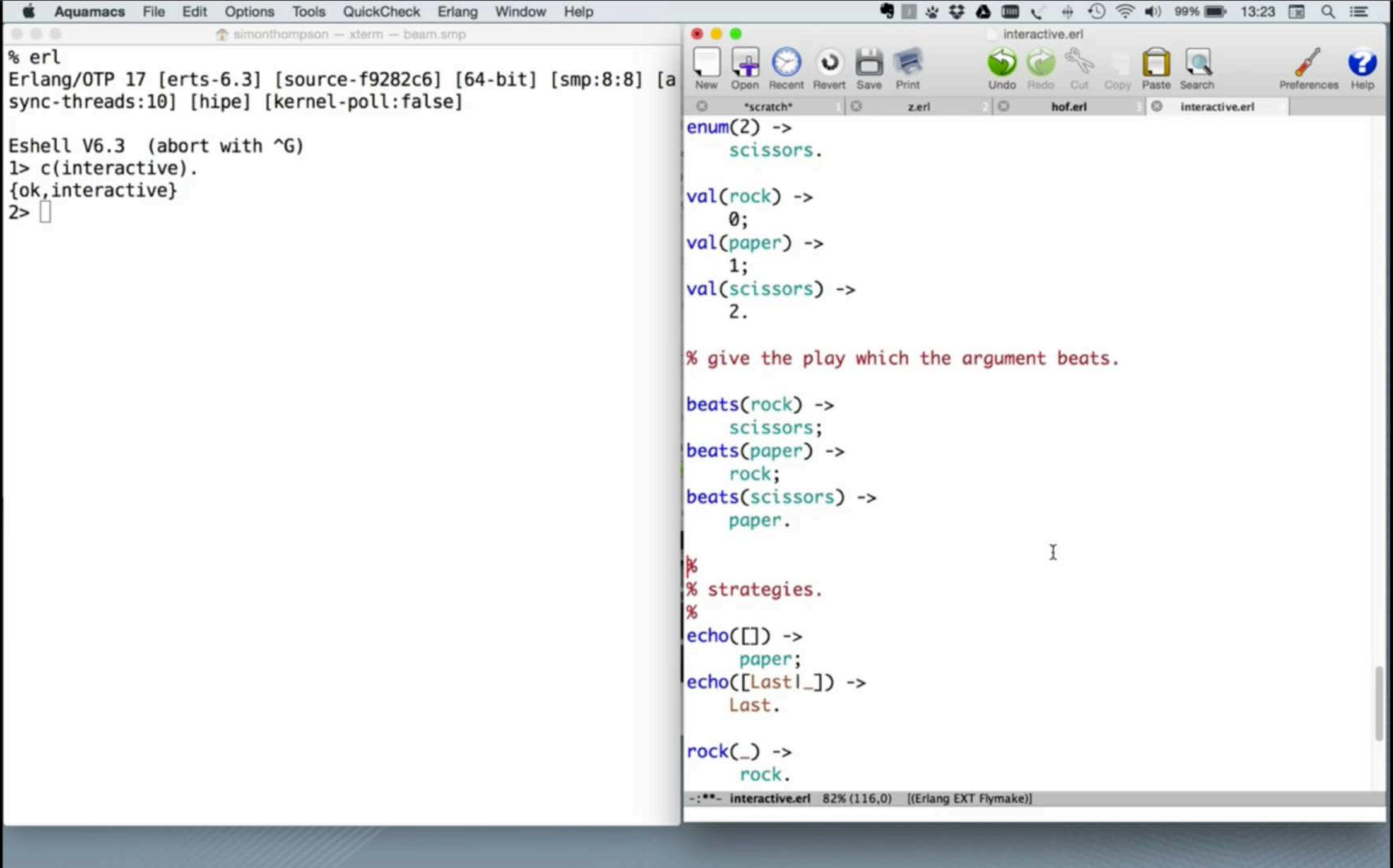
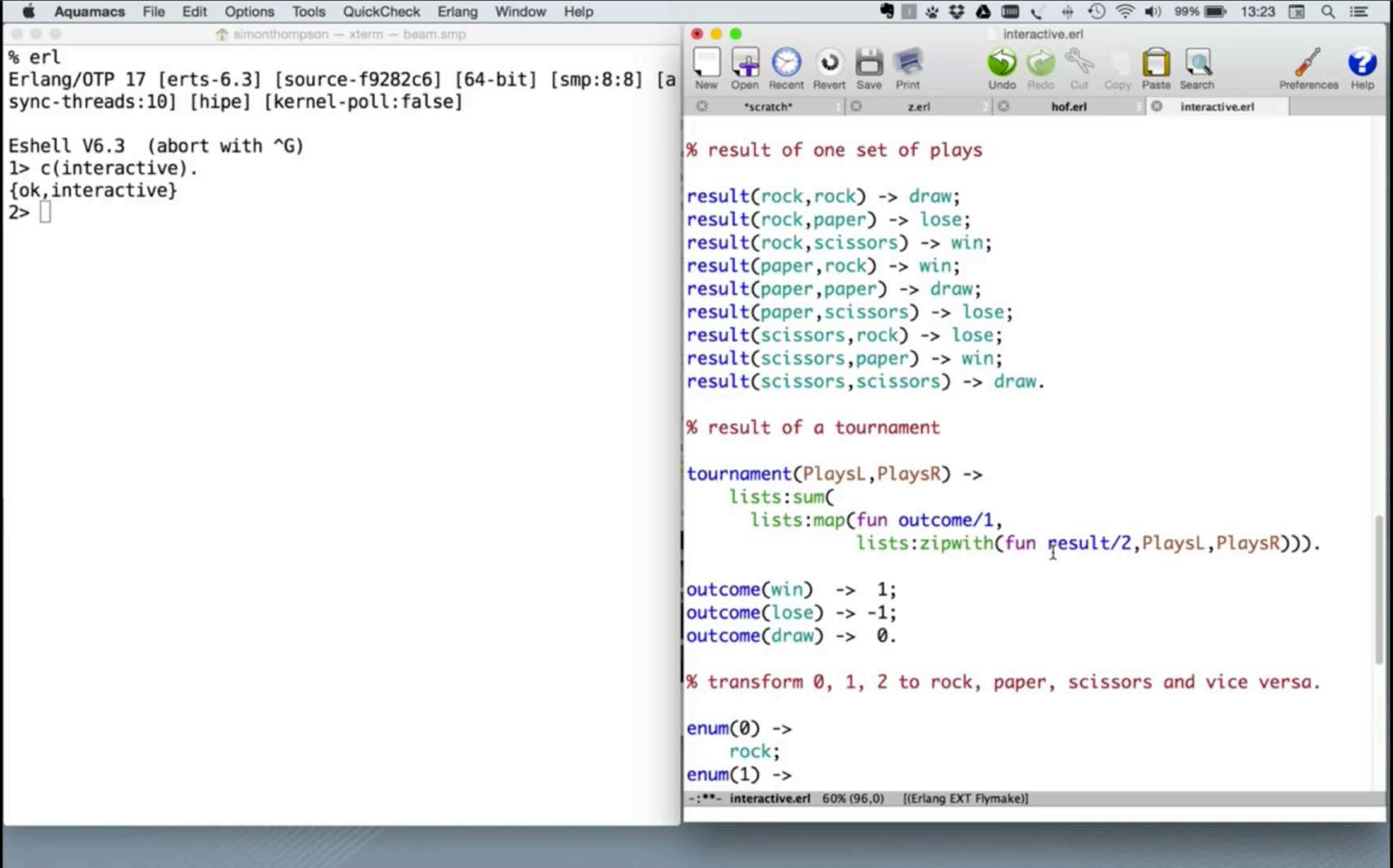
University of Kent



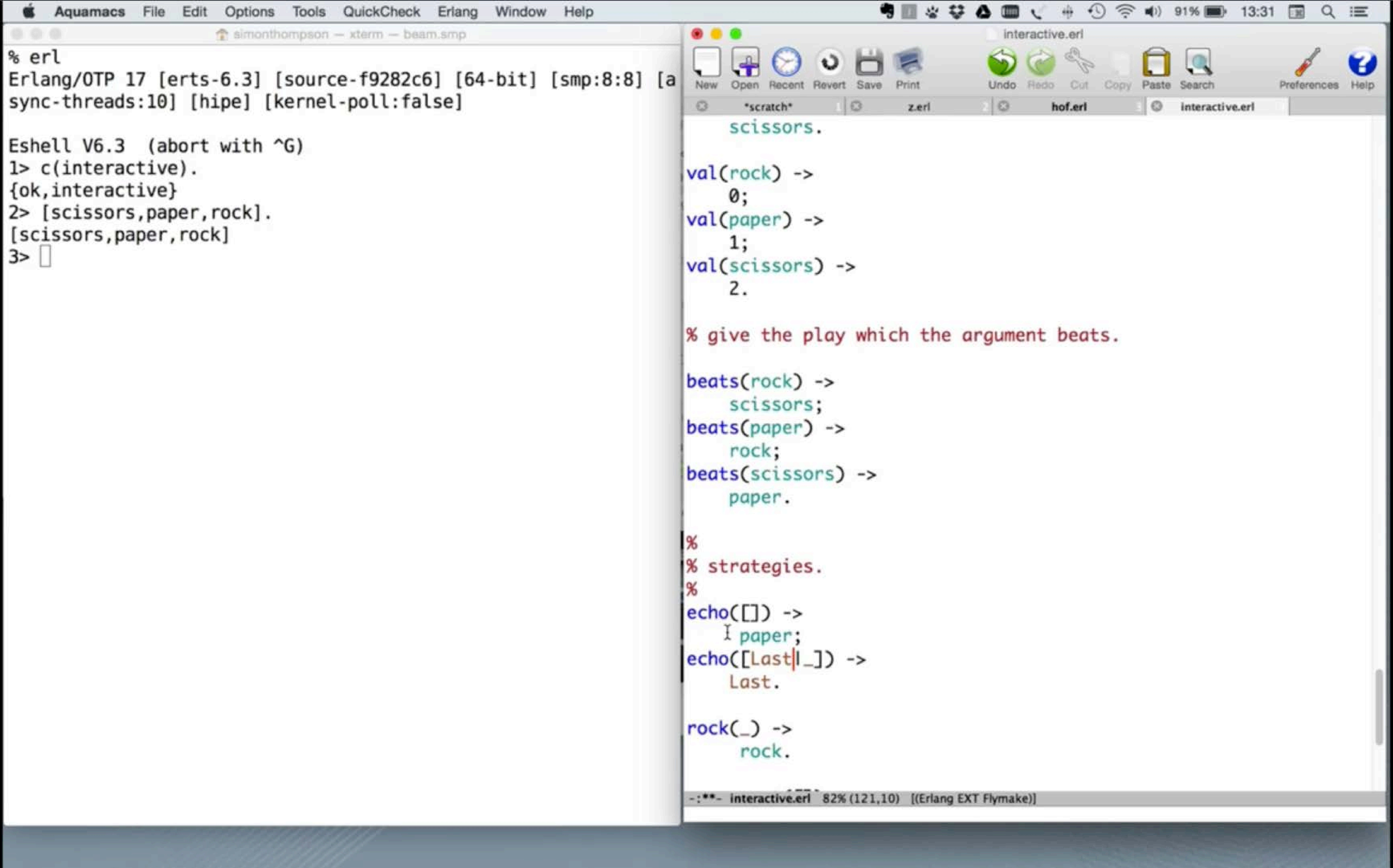
```
% erl
Erlang/OTP 17 [erts-6.3] [source-f9282c6] [64-bit] [smp:8:8] [a
sync-threads:10] [hipe] [kernel-poll:false]

Eshell V6.3  (abort with ^G)
1> c(interactive).
{ok,interactive}
2> 
```

```
enum(2) ->
    scissors.

val(rock) ->
    0;
val(paper) ->
    1;
val(scissors) ->
    2.

% give the play which the argument beats.

beats(rock) ->
    scissors;
beats(paper) ->
    rock;
beats(scissors) ->
    paper.

%
% strategies.
%
echo([]) ->
     paper;
echo([Last|_]) ->
    Last.

rock(_) ->
     rock.
```



```
% erl
Erlang/OTP 17 [erts-6.3] [source-f9282c6] [64-bit] [smp:8:8] [a
sync-threads:10] [hipe] [kernel-poll:false]

Eshell V6.3  (abort with ^G)
1> c(interactive).
{ok,interactive}
2> 
```

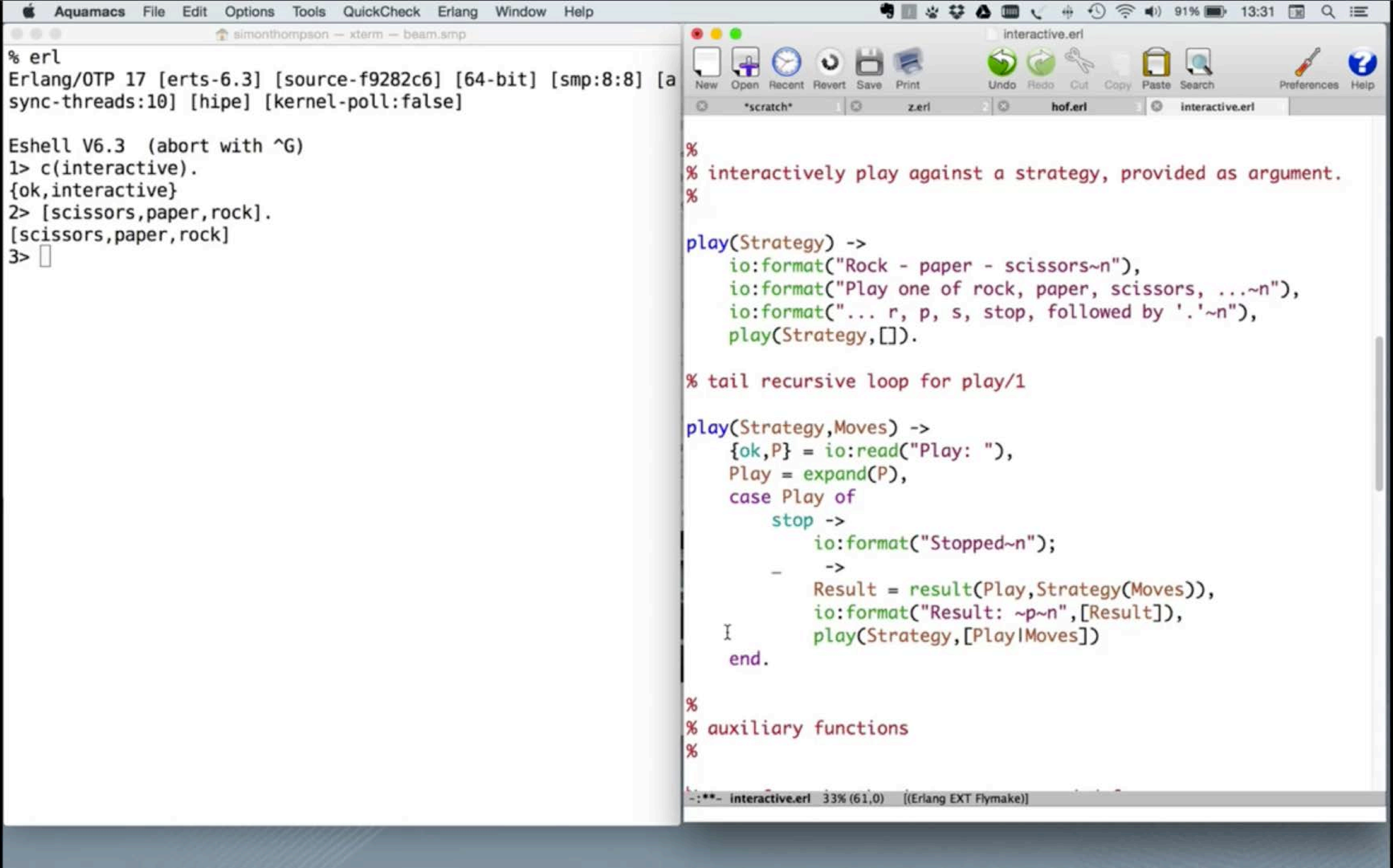
```
% result of one set of plays

result(rock,rock) -> draw;
result(rock,paper) -> lose;
result(rock,scissors) -> win;
result(paper,rock) -> win;
result(paper,paper) -> draw;
result(paper,scissors) -> lose;
result(scissors,rock) -> lose;
result(scissors,paper) -> win;
result(scissors,scissors) -> draw.

% result of a tournament

tournament(PlaysL,PlaysR) ->
    lists:sum(
      lists:map(fun outcome/1,
                lists:zipwith(fun result/2,PlaysL,PlaysR))).

outcome(win)  ->  1;
outcome(lose) -> -1;
outcome(draw) ->  0.

% transform 0, 1, 2 to rock, paper, scissors and vice versa.

enum(0) ->
    rock;
enum(1) ->
```



```
% erl
Erlang/OTP 17 [erts-6.3] [source-f9282c6] [64-bit] [smp:8:8] [a
sync-threads:10] [hipe] [kernel-poll:false]

Eshell V6.3  (abort with ^G)
1> c(interactive).
{ok,interactive}
2> [scissors,paper,rock].
[scissors,paper,rock]
3> 
```

```
    scissors.

val(rock) ->
    0;
val(paper) ->
    1;
val(scissors) ->
    2.

% give the play which the argument beats.

beats(rock) ->
    scissors;
beats(paper) ->
    rock;
beats(scissors) ->
    paper.

%
% strategies.
%
echo([]) ->
     paper;
echo([Last|_]) ->
    Last.

rock(_) ->
     rock.
```

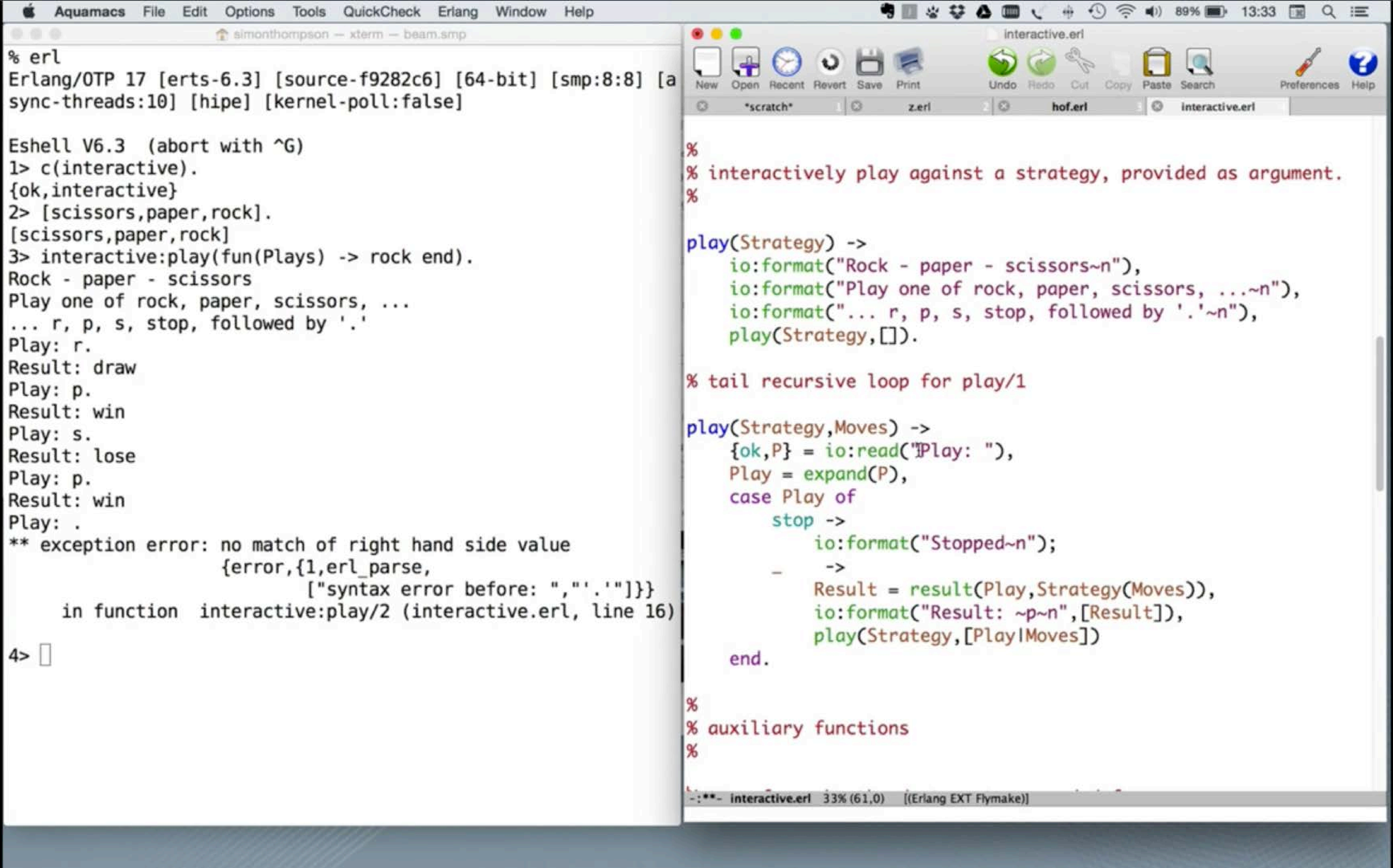
```
-:**-  interactive.erl   82% (121,10)   [(Erlang EXT Flymake)]
```



```
% erl
Erlang/OTP 17 [erts-6.3] [source-f9282c6] [64-bit] [smp:8:8] [a
sync-threads:10] [hipe] [kernel-poll:false]

Eshell V6.3  (abort with ^G)
1> c(interactive).
{ok,interactive}
2> [scissors,paper,rock].
[scissors,paper,rock]
3> 
```

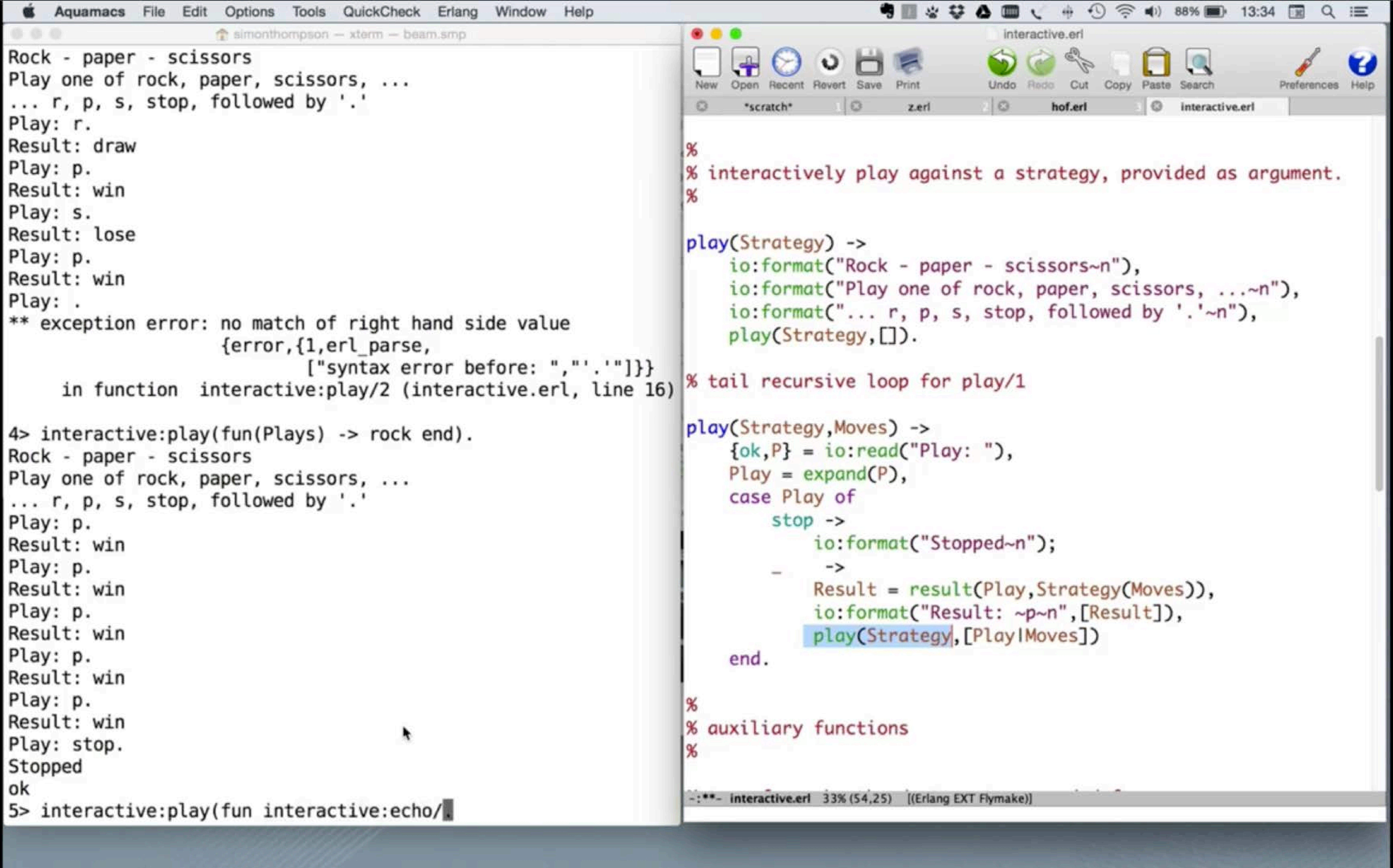
```
%
% interactively play against a strategy, provided as argument.
%

play(Strategy) ->
    io:format("Rock - paper - scissors~n"),
    io:format("Play one of rock, paper, scissors, ...~n"),
    io:format("... r, p, s, stop, followed by '.'~n"),
    play(Strategy,[]).

% tail recursive loop for play/1

play(Strategy,Moves) ->
    {ok,P} = io:read("Play: "),
    Play = expand(P),
    case Play of
        stop ->
            io:format("Stopped~n");
        _    ->
            Result = result(Play,Strategy(Moves)),
            io:format("Result: ~p~n",[Result]),
            play(Strategy,[Play|Moves])
    end.

%
% auxiliary functions
%
```

-:**- interactive.erl   33% (61,0)   [(Erlang EXT Flymake)]



```
% erl
Erlang/OTP 17 [erts-6.3] [source-f9282c6] [64-bit] [smp:8:8] [a
sync-threads:10] [hipe] [kernel-poll:false]

Eshell V6.3  (abort with ^G)
1> c(interactive).
{ok,interactive}
2> [scissors,paper,rock].
[scissors,paper,rock]
3> interactive:play(fun(Plays) -> rock end).
Rock - paper - scissors
Play one of rock, paper, scissors, ...
... r, p, s, stop, followed by '.'
Play: r.
Result: draw
Play: p.
Result: win
Play: s.
Result: lose
Play: p.
Result: win
Play: .
** exception error: no match of right hand side value
                    {error,{1,erl_parse,
                            ["syntax error before: ","'.'"]}}
     in function  interactive:play/2 (interactive.erl, line 16)

4> 
```

```
%
% interactively play against a strategy, provided as argument.
%

play(Strategy) ->
    io:format("Rock - paper - scissors~n"),
    io:format("Play one of rock, paper, scissors, ...~n"),
    io:format("... r, p, s, stop, followed by '.'~n"),
    play(Strategy,[]).

% tail recursive loop for play/1

play(Strategy,Moves) ->
    {ok,P} = io:read("Play: "),
    Play = expand(P),
    case Play of
        stop ->
            io:format("Stopped~n");
        _    ->
            Result = result(Play,Strategy(Moves)),
            io:format("Result: ~p~n",[Result]),
            play(Strategy,[Play|Moves])
    end.

%
% auxiliary functions
%
```



```
Rock - paper - scissors
Play one of rock, paper, scissors, ...
... r, p, s, stop, followed by '.'
Play: r.
Result: draw
Play: p.
Result: win
Play: s.
Result: lose
Play: p.
Result: win
Play: .
** exception error: no match of right hand side value
                    {error,{1,erl_parse,
                            ["syntax error before: ","'.'"]}}
     in function  interactive:play/2 (interactive.erl, line 16)

4> interactive:play(fun(Plays) -> rock end).
Rock - paper - scissors
Play one of rock, paper, scissors, ...
... r, p, s, stop, followed by '.'
Play: p.
Result: win
Play: p.
Result: win
Play: p.
Result: win
Play: p.
Result: win
Play: p.
Result: win
Play: stop.
Stopped
ok
5> interactive:play(fun interactive:echo/.
```

```
%
% interactively play against a strategy, provided as argument.
%

play(Strategy) ->
    io:format("Rock - paper - scissors~n"),
    io:format("Play one of rock, paper, scissors, ...~n"),
    io:format("... r, p, s, stop, followed by '.'~n"),
    play(Strategy,[]).

% tail recursive loop for play/1

play(Strategy,Moves) ->
    {ok,P} = io:read("Play: "),
    Play = expand(P),
    case Play of
        stop ->
            io:format("Stopped~n");
        _    ->
            Result = result(Play,Strategy(Moves)),
            io:format("Result: ~p~n",[Result]),
            play(Strategy,[Play|Moves])
    end.

%
% auxiliary functions
%
```

Press esc to exit full screen

Functions as results

Higher-order functions

Functions as arguments …

… e.g. `map`, `filter`, `zipwith`, `foldr` …

Writing down functions in Erlang

Find the area of all … using **area**

```
all_areas(Xs) -> lists:map(fun area/1,Xs).
```

If **area** is in the module **shape**

```
all_areas(Xs) -> lists:map(fun shape:area/1,Xs).
```

In the shell, with the variable **Area** bound to the **area** function

```
... > lists:map(Area,Xs).
```

As well as being able to use **fun** expressions directly.

"Partially applied" functions

The **+** operator applies to two numbers, to give a number.

What if we "applied it to a single argument" …

```
add(X) ->
    fun(Y) -> X+Y end.
```

… we get a *function*, that adds **X** to its argument.

```
addOneToAll(Xs) ->
    lists:map(add(1),Xs).

addToAll(N,Xs) ->
    lists:map(add(N),Xs).
```

Composing functions

We often use *function composition* to define functions …

… first do this (`F`) and then do this (`G`).

```
compose(F,G) ->
    fun(X) -> G(F(X)) end.
```

The result of `compose` is a *function*, that *"composes"* its arguments.

Higher-order functions

Functions as arguments …

… e.g. **`map`**, **`filter`**, **`zipwith`**, **`foldr`** …

Functions as results …

… e.g. "curried" functions

Functions as arguments *and* results …

… e.g. **`compose`**, **`iterate`**, etc.

University of Kent

University of Kent

Rock-paper-scissors and HoFs

Higher-order functions

Functions as arguments …

… e.g. `map`, `filter`, `zipwith`, `foldr` …

Functions as results …

… e.g. "curried" functions

Functions as arguments *and* results …
… e.g. `compose`, `iterate`, etc.

Strategy vs Strategy

```
play_two(StrategyL,StrategyR,N) ->
    play_two(StrategyL,StrategyR,[],[],N).

play_two(_,_,PlaysL,PlaysR,0) ->
    io:format("Overall result … ");

play_two(StrategyL,StrategyR,PlaysL,PlaysR,N) ->
    PlayL = StrategyL(PlaysR),
    PlayR = StrategyR(PlaysL),
    Result = result(PlayL,PlayR),
    io:format("Result: ~p~n",[Result]),
    play_two(StrategyL,StrategyR,[PlayL|PlaysL],[PlayR|PlaysR],N-1).
```

Strategy vs Strategy

```
play_two(StrategyL,StrategyR,N) ->
    play_two(StrategyL,StrategyR,[],[],N).

play_two(_,_,PlaysL,PlaysR,0) ->
    io:format("Overall result … ");

play_two(StrategyL,StrategyR,PlaysL,PlaysR,N) ->
    PlayL = StrategyL(PlaysR),
    PlayR = StrategyR(PlaysL),
    Result = result(PlayL,PlayR),
    io:format("Result: ~p~n",[Result]),
    play_two(StrategyL,StrategyR,[PlayL|PlaysL],[PlayR|PlaysR],N-1).
```

Iteration

Apply the function **F**, **N** times

```
iterate(0) ->
    fun(_F) ->
      fun id/1 end;

iterate(N) ->
    fun(F) ->
    compose(F,(iterate(N-1))(F)) end.
```

How would you define it using `lists:map`, `lists:foldr` and `compose`?

Higher-order functions

Functions as arguments …

… e.g. `map`, `filter`, `zipwith`, `foldr` …

Functions as results …

… e.g. "curried" functions

Functions as arguments *and* results …
… e.g. `compose`, `iterate`, etc.

University of Kent