Erlang is a concurrent, fault-tolerant, robust, distributed programming language . . .

. . . that is based on the paradigm of functional programming.
FUNCTIONAL
ERLANG
pattern matching

recursion

```latex
isPath(_Maze,[]) ->
    true;

isPath(Maze,[P]) ->
    inGrid(Maze,P);

isPath(Maze,[P1,P2|Ps]) ->
    inGrid(Maze,P1) andalso
    isEmpty(Maze,P1) andalso
    adjacent(P1,P2) andalso
    isPath(Maze,[P2|Ps]).
```
do-it-yourself data types

```plaintext
area({circle,_,R}) ->
  math:pi()*R*R;
area({tri,_,A,B,C}) ->
  S = (A+B+C)/2,
  math:sqrt(S*(S-A)*(S-B)*(S-C)).
```
tail recursion

echo(Pid,N) ->
  receive
    Msg -> Pid!Msg
  end,
  echo(Pid,N+1).

standard HOFs

PossPoints = lists:filter( fun (X) ->
  not lists:member(X,Avoid) end,adjPoints(Maze,P1)),
lists:concat(lists:map(fun (P)->
  [ [P1|Path] || Path <- allPaths(Maze,P,P2,[P1|Avoid]]) ] end, PossPoints))

list comprehensions
numbers
atoms
tuples
lists
functions

2, 2.3, 123456789023456, ...
true, 'not true', symbol, ...
{circle, {2.0, 3.0}, 4.3}, ...
[2, 3, 4, ...], [2, 3 | [4, ...]], ...

fun(F) ->
  fun(Y) -> F(2*Y) - F(Y) end
end
4.13 Influence from functional programming

By now the influence of functional programming on Erlang was clear. What started as the addition of concurrency to a logic language ended with us removing virtually all traces of Prolog from the language and adding many well-known features from functional languages.

Higher-order functions and list comprehensions were added to the language. The only remaining signs of the Prolog heritage lie in the syntax for atoms and variables, the scoping rules for variables and the dynamic type system.

“the influence is clear”
fun
FUNCTIONS AS DATA
“Functions are first-class citizens”

A function actively represents behaviour of some sort, and we deal with it just like any other kind of data.
What is a strategy?

Random
Echo
No repeats
Statistical

...
We choose what to play, depending on your last move, or the history of all your moves.
What is a strategy?

-\texttt{type plays()} :: [\texttt{play()}].

-\texttt{type strategy()} :: fun((\texttt{plays()}) \rightarrow \texttt{play()}).

We choose what to play, depending on your last move, or the history of all your moves.
Random
Echo
No repeats
Statistical

random(_) ->
    random_play();

echo([]) ->
    random_play();
echo([X|_Xs]) ->
    X.

beat([]) ->
    random_play();
beat([X|_]) ->
    case X of
        rock -> scissors;
        paper -> rock;
        scissors -> paper
    end.
interact(Strategy) ->
    interact(Strategy, []).

% The second argument here is the accumulated input from the player
% Note that this function doesn't cheat: the Response is chosen
% before the Play from the player.

interact(Strategy, Xs) ->
    Response = Strategy(Xs),
    {ok, [Play|_]} = io:fread('play one of rock, paper, scissors, or stop: ',';"~a"),
    case Play of
        stop -> ok;
        _ ->
            Result = result({Play, Response}),
            io:format("Machine has played ~p, result is ~p~n", [Response, Result]),
            interact(Strategy, [Play|Xs])
    end.
What is a strategy combinator?

Choose randomly between these strategies.

Apply them all and choose most popular result.

Replay each of these strategies on the history so far and apply the one that’s been best so far.
What is a strategy combinator?

apply them all and choose most popular result.

replay each of these strategies on the history so far and apply the one that’s been best so far.
Take home

Toy example

Generality: not just a finite set . . .
Up a level: combining strategies
Worldrps.com has a new look

Say goodbye to the old cluttered look of the World RPS Society site.

The IT Brigade told us it would take them four weeks to re-do the worldrps.com web site. So after consuming four years, 4 palettes of Mellow Yellow, dozens of crates of Pringles, and surviving a few health scares, the team has done it.
https://github.com/simonjohnthompsonStreams
PARSER
COMBINATORS
text ⟷ [square] ⟷ parse tree

remaining text
-type parser() :: fun((string()) -> {ast(), string()}).

-spec sequence(parser(), parser()) -> parser().
-type parser() :: fun(string() -> [{ast(), string()}]).

-spec sequence(parser(), parser()) -> parser().
Take home

Real example

Haskell, Scala, OCaml, Elixir, . . .

Hints at a design pattern
but 

If all we want is **one** parse, then we should only evaluate the list of possible results on demand.
EVALUATION ON DEMAND
function evaluation in Erlang
function evaluation in Erlang

evaluate the arguments before the body

switch(N,Pos,Neg) ->
  case N>0 of
    true -> Pos;
    _ -> Neg
  end.
function evaluation in Erlang

evaluate the arguments before the body

fully evaluate the argument

```erlang
switch(N,Pos,Neg) ->
    case N>0 of
        true -> Pos;
        _     -> Neg
    end.
```

```erlang
sum_first_two([A,B|_Rest]) -> A+B.
```
but if an argument is a function then it’s passed unevaluated.
but if an argument is a function then it’s passed unevaluated.

fun () -> Stuff end
but if an argument is a function then it’s passed unevaluated.

fun () → Stuff end

fun () → Stuff end ()
STREAMS
build streams
deconstrict

```haskell
cons(X,Xs) ->
  fun() -> \{X,Xs\} end.

head(L) ->
  case (L()) of
  | \{H,_\} -> H
  end.

tail(L) ->
  case (L()) of
  | \{_,T\} -> T
  end.
```
define(cons(X,Xs),
    fun() -> {X,Xs} end).

head(L) ->
    case L(L()) of
    {H, _} -> H
    end.

tail(L) ->
    case L(L()) of
    {_, T} -> T
    end.
\[ \text{ones()} \rightarrow \text{cons}(1, \text{ones}()) \]

\[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, \ldots \]
ones() ->
cons(1, ones()).
\[\text{ns}(N) \rightarrow \text{cons}(N, \text{ns}(N+1)).\]

42, 43, 44, 45, 46, 47, 48, 49, 50, ...
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, …

```plaintext
primes() -> sieve(ns(2)).

sieve(Ns) ->
    H = head(Ns),
    ?cons(H, sieve(cut(H, tail(Ns))))).

cut(N, Ns) ->
    H = head(Ns),
    case H rem N of
    0 -> cut(N, tail(Ns));
    _ -> ?cons(H, cut(N, tail(Ns)))
    end.
```
fibs() →
  ?cons(0,
    ?cons(1,
      addZip(fibs(), tail(fibs()))).

addZip(Xs, Ys) →
  ?cons(head(Xs)+head(Ys), addZip(tail(Xs), tail(Ys))).

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
fibs() ->
  ?cons(0,
  ?cons(1,
    addZip(fibs(), tail(fibs()))).

addZip(Xs, Ys) ->
  ?cons(head(Xs) + head(Ys), addZip(tail(Xs), tail(Ys))).
Take home

“infinite” streams
apparently circular
repeated re-computation
LAZY EVALUATION
ensure that each argument is evaluated at most once
ensure that each argument is evaluated at most once

we must ensure that results are memoised in some way
but isn't that a job for the compiler?
key idea

we explicitly manage how results are stored once evaluated
use an ETS table to keep track of evaluated results, or . . .

. . . model the store functionally, thread it through the calculations
USING ETS TABLES
store either the head and tail, or a “thunk” to be evaluated
-define(cons(X,Xs), begin ets:insert(tab, [0,next_ref()] + 1)) ,  
    ets:insert(tab, [next_ref()], [thunk, fun () -> [X,Xs] end]),  
    io:format("done cons insert~n"),  
    [ref,next_ref()] end).

\[
\begin{array}{cccc}
  0 & 1 & 2 & 3 \\
  3 &   &   &   \\
\end{array}
\]

\{H,T\} \quad \{\text{thunk,F}\}
head([ref,Ref]) ->
  case ets:lookup(tab, Ref) of
    [{Ref, {thunk, F}}] -> Val = F(),
    ets:insert(tab, {Ref, Val}),
    {H, _} = Val,
    H;
  [{Ref, {H, _}}] -> H
end.
ones() ->
  cons(1, ones()).
onesC() ->
  This = next_ref() + 1,
  ?cons(1, {ref, This}).
fibs() ->
   \( ?\text{cons}(0, \)
   \[ \begin{align*}
   & ?\text{cons}(1, \\
   & \text{addZip}(\text{fibs}(), \text{tail}(\text{fibs}()))).
   \end{align*} \]
addZip(Xs, Ys) ->
   \( ?\text{cons}(\text{head}(Xs)+\text{head}(Ys), \text{addZip}(\text{tail}(Xs), \text{tail}(Ys))). \)
fibsC() ->
  This = next_ref()+1,
  Next = This+1,
  ?cons(0,
    ?cons(1,
      addZip({ref,This},{ref,Next})))).
Explicitly managed refs

Simulates full lazy implementation

Uses impure features . . .

. . . but a smooth transition
AN EXPLICIT STORE
store before

store after

input

result
Printing out the first $N$ values

```prolog
ps(Xs, N, Tab) ->
    io:format("\~w\~n", [head(Xs, Tab)]),
    {T, Tab1} = tail(Xs, Tab),
    ps(T, N-1, Tab1).
```
Node to \( \{\text{Head, \{thunk, Tail\}}\} \)

Thunk takes \texttt{state} as argument . . .

. . . so that the suspended computation can be evaluated in the context of the current state.
MEMOISATION
use ETS for general memoisation

\begin{verbatim}
fib(0) -> 0;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
\end{verbatim}
use ETS for general memoisation

```erlang
def(fibM, 0, 0).
def(fibM, 1, 1).
def(fibM, N, [\
  case ets:lookup(tab, N) of
  [[], V = fibM(N-1) + fibM(N-2)],
  ets:insert(tab, [N, V]),
  V,
  [{N, V}] -> V
  end.
```
-type vector(T) :: \{integer(), list(T)\}.

-define(mkV(Xs),\{length(Xs), Xs\}).

-define(length(V),element(1,V)).
-type vector(T) :: {integer(), list(T)}.

-define(mkV(Xs), {length(Xs), Xs}).

-define(length(V), element(1, V)).

-spec joinV(T, vector(T)) -> vector(T).

joinV(Sep, [M, Xs]) -> [2*M-1, lists:join(Sep, Xs)].

-define(join(Sep, V), element(2, joinV(Sep, V))).
TO CONCLUDE
functions are flexible and powerful modelling tool

strategies

collectors

parsers

simulation
pure modelling of effects is not straightforward

monads, monad transformers, effects, . . . provide some useful patterns
reify?

can model DSLs of strategies, parsers, and write interpreters for these DSLs into the functions we’ve seen here
data and types

all the data we used here was well understood 30 years ago

it is just that the types have changed
functions are flexible and powerful modelling tool

strategies
parsers
simulation
and I didn’t say anything directly about dependent types ;-( 