MAKING IT LAZY: NEVER EVALUATE ANYTHING MORE THAN ONCE
fun
https://github.com/simonjohnthompson/streams
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

...
What is a strategy?

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

- type play() :: rock | paper | scissors.
- type strategy() :: fun(([play()])) -> play()).

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

...
% 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.

-spec interact(strategy(),[play()]) -> ok.

interact(Strategy,Xs) ->
    Response = Strategy(Xs),
    {ok,[Play|_]} = io:fread('play rock, paper, scissors, stop: ',"~a"),
    case Play of
        stop -> ok;
        _ ->
            Result = result({Play,Response}),
            io:format("Machine plays ~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.
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.
EVALUATION ON DEMAND
function evaluation in Erlang
function evaluation in Erlang

evaluate the arguments before the body

```erlang
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

\[
\text{switch}(N, \text{Pos}, \text{Neg}) \rightarrow \\
\begin{align*}
\text{case } N > 0 \text{ of} & \\
\text{true} & \rightarrow \text{Pos}; \\
_ & \rightarrow \text{Neg}
\end{align*}
\]

\[
\text{sum\_first\_two}([A,B|\_\text{Rest}]) \rightarrow A+B.
\]
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.

```plaintext
fun () -> Stuff end

fun () -> Stuff end()
```
DELAY!
a lazy switch

\[-\text{spec } \text{lswitch(number(),fun(() \rightarrow T),fun(() \rightarrow T))} \rightarrow T.\]

\[\text{lswitch}(N, \text{Pos}, \text{Neg}) \rightarrow\]
\[\text{case } N>0 \text{ of}\]
\[\quad \text{true} \rightarrow \text{Pos}();\]
\[\quad \_ \rightarrow \text{Neg}()\]
\[\text{end}.\]
a lazy switch

\[-\text{spec } \text{lswitch}(\text{number()}, \text{fun}(() \to T), \text{fun}(() \to T)) \to T.\]

\[\text{lswitch}(N, \text{Pos}, \text{Neg}) \to\]
\[\begin{cases} 
\text{case } N > 0 \ & \text{of} \\
\text{true} \ & \to \text{Pos}(); \\
\_ & \to \text{Neg}() \\
\end{cases}\]
\[\text{end}.\]

\[\text{lex1()} \to \text{lswitch}(1, \text{fun}() \to 3+4 \text{ end}, \text{fun}() \to 1/0 \text{ end}).\]
-spec lswitch(number(), fun(() -> T), fun(() -> T)) -> T.

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

-define(switch(N,Pos,Neg),
  lswitch(N, fun() -> Pos end, fun() -> Neg end)).
-spec lswitch(number(),fun(() ->T),fun(() ->T)) -> T.

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

-define(switch(N,Pos,Neg),
    lswitch(N,fun() -> Pos end,fun() -> Neg end)).

lex2() -> ?switch(1,3+4,1/0).
STREAMS
build streams

\[
\text{cons}(X, Xs) \rightarrow \\
\text{fun}() \rightarrow \{X, Xs\} \text{ end.}
\]
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()) of
  {H,_} -> H
end.
tail(L) ->
case (L()) of
  {_T} -> T
end.
ones() →
  cons(1, ones()).

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
ones() -> ?cons(1, ones()).
\texttt{ns(N) \rightarrow \texttt{\texttt{?\texttt{cons(N,ns(N+1))}}}}.

\texttt{42, 43, 44, 45, 46, 47, 48, 49, 50, \ldots}
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)))).
fibs() ->
  cons(0,
    cons(1,
      addZip(fibs(), tail(fibs()))).

addZip(Xs, Ys) ->
  cons(head(Xs) + head(Ys), addZip(tail(Xs), tail(Ys))).
demo
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
MEMOISATION
use ETS for general memoisation
use ETS for general memoisation

fib(0) -> 0;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
fib(0) -> 0;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).

fibM(0) -> 0;
fibM(1) -> 1;
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.
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).
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() \rightarrow
This = next\_ref()+1,
?cons(1,\{ref,This\}).
fibs() ->
  cons(0,
    cons(1,
      addZip(fibs(), tail(fibs())))).

addZip(Xs, Ys) ->
  cons(head(Xs) + head(Ys), addZip(tail(Xs), tail(Ys))).
fibs() ->
  ?cons(0,
      ?cons(1,
        addZip(fibs(), tail(fibs())))
fib\$CVar() \rightarrow 
   \text{This} = \text{next_ref()} + 1,
   \text{cons}(0, 
   \text{cons}(1, 
   \text{addZip}({\text{ref, This}}, \text{tail}({\text{ref, This}}))))).
Explicitly managed refs

Simulates full lazy implementation

Uses impure features . . .

. . . but a smooth transition
AN EXPLICIT STORE
Printing out the first N values

\[ \text{ss}(_Xs, 0, _T) \rightarrow \text{io:format}("\sim n"); \]

\[ \text{ss}(Xs, N, Sto) \rightarrow \text{io:format}("\sim w, \text{[head}(Xs, Sto)\text{])}, \{T, Sto1\} = \text{tail}(Xs, Sto), \text{ss}(T, N-1, Sto1). \]
Node to \{\text{Head}, \{\text{thunk}, \text{Tail}\}\}

Thunk takes \text{state} as argument \ldots \ldots so that the suspended computation can be evaluated in the context of the current state.
Construct a list

-define(cons(X,F,Sto),
  begin  {{ref,next_ref(Sto)},
       Sto#{next_ref(Sto) => {X, {thunk, F}}},
       0 => next_ref(Sto)+1}
  end).
\text{tail}\{\text{ref,Ref}\},\text{Sto}\) \rightarrow \\
\text{case maps: get(Ref,Sto) of} \\
\{\text{ref},\text{R}\} \rightarrow \\
\text{Hd = head}\{\text{ref},\text{R}\},\text{Sto}\), \\
\{\text{Tl},\text{Sto1}\} =\text{tail}\{\text{ref},\text{R}\},\text{Sto}\), \\
\text{Sto2} = \text{Sto1}\#\{\text{Ref} \Rightarrow \{\text{Hd},\text{Tl}\}\}, \\
\{\text{Tl},\text{Sto2}\}; \\
\{\text{Hd},\{\text{thunk},\text{F}\}\} \rightarrow \\
\{\text{Tl},\text{StoC}\} = \text{F}(\text{Sto}), \\
\text{Sto1} = \text{StoC}\#\{\text{Ref} \Rightarrow \{\text{Hd},\text{Tl}\}\}, \\
\{\text{Tl},\text{Sto1}\}; \\
\{\_,\text{T}\} \rightarrow \\
\{\text{T},\text{Sto}\} \\
\text{end}.
Fibonacci numbers

```haskell
fibsC(Sto) ->
  This = next_ref(Sto),
  ?cons(0, fun(T)) ->
    ?cons(1, fun(S)) ->
      begin
        {Tl,S1} = tail({ref,This},S),
        addZip({ref,This},Tl,S1)
      end,
    end,
  end,
end, Sto).
```
TO CONCLUDE
functions are flexible and powerful modelling tool

strategies
simulations
suspensions
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
fun