15-150

Spring 2012

Streams

```
cat "../../asgn/hw/09/code/ratplane.sml"
|> grep (linesMatching (String "dist"))
|> nl
|> truncate 3 |> flatten
```

Streams

functional interface to an impure world

benign effects: use mutation under the hood to achieve a functional interface

Benign Effects

```
fun reachable g x y =
 let
   val visited = ref Visited.empty
   fun dfs (x : Node.t) : bool =
       case Node.compare (x, y) of
           EQUAL => true
         | _ => let val (ref curVisited) = visited in
                    case Visited.member curVisited x of
                        true => false
                      | false => let val () = visited := (Visited.insert curVisited x) in
                                     (NodeSet.exists dfs (G.successors g x)) end
                end
in
    dfs x
end
```

Benign Effects

imperative implementation that
looks functional to clients

sometimes effects are faster or
nicer (implicit communication)

but have to think about parallelism…

Memoizer

```
fun memo (f : ((D.Key.t -> 'a) -> (D.Key.t -> 'a)))
    : (D.Key.t -> 'a) =
    let
      val hist : 'a D.dict ref = ref D.empty

      fun wrapper x =
        case D.lookup (!hist) x
         of SOME v => v
          | NONE => let val res = f wrapper x
                    in
                        hist := D.insert (!hist) (x, res);
                        res
                    end
  in
      wrapper
  end
```

```
fun fib _ (0 : IntInf.int) = (0 : IntInf.int)
  | fib _ 1 = 1
  | fib f n = f (n - 1) + f (n - 2)
```

Ephemeral Data Structures

```
signature EPH_GAME =
sig
    type state
    type move

    val make_move : (state * move) -> unit

    ...
end
```

Ephemeral Data Structures

admit more implementations

but complicate backtracking/parallelism

Using Effects?

	persistent	ephemeral
parallel	FP benign effects	concurrency
sequential		mutation OK

Mutation

```
fun update (f : 'a -> 'a) (r : 'a ref) : unit =
    let val (ref cur) = r in r := f cur end
fun deposit  n a = update (fn x => x + n) a
fun withdraw n a = update (fn x => x - n) a

val account = ref 100
val () = deposit 100 account
val () = withdraw 50 account
val _ = Seq.tabulate (fn 0 => deposit 100 account
                       | 1 => withdraw 50 account) 2
```

Mutation

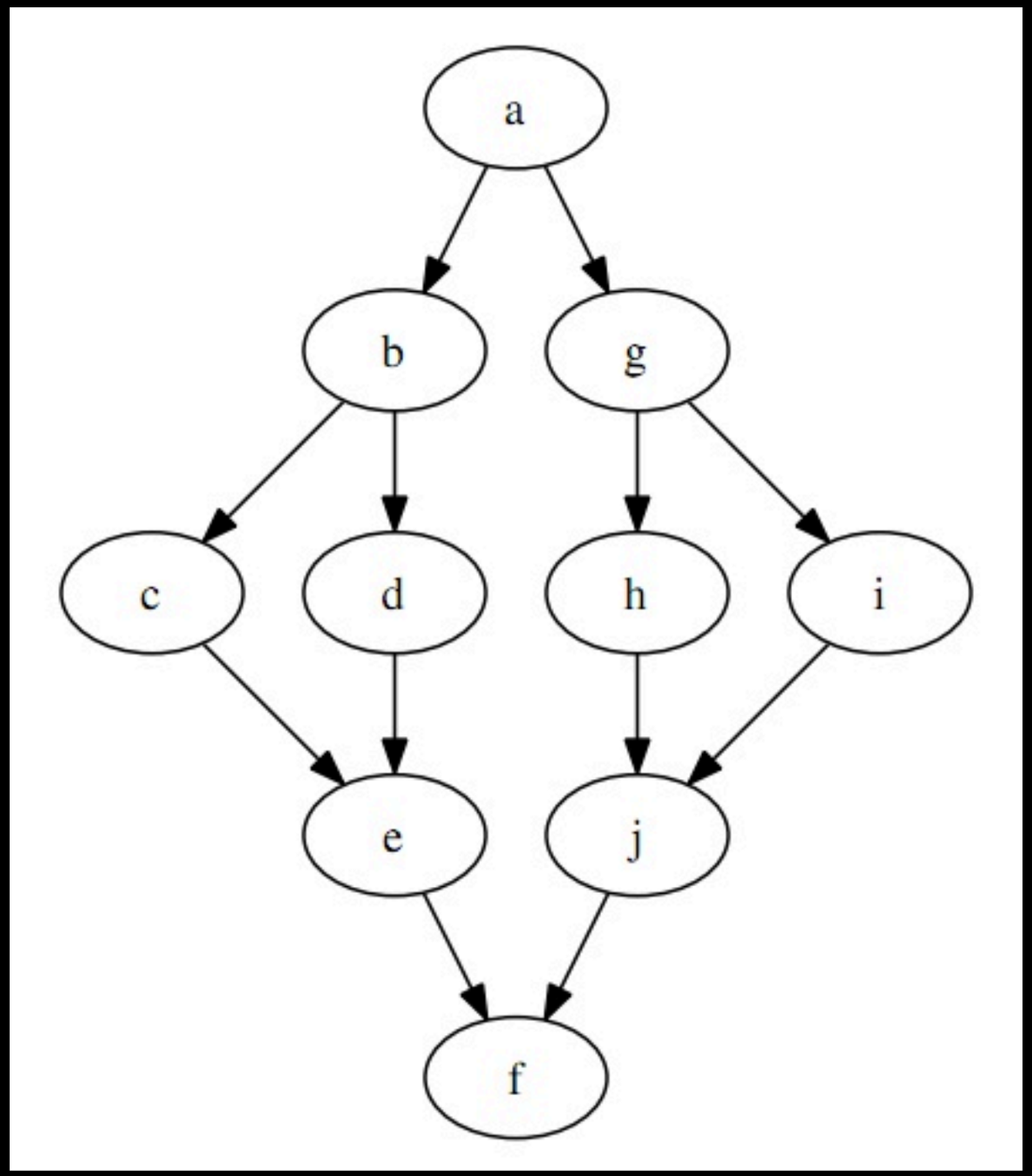
can lead to race conditions

Parallelism and Effects

Determinism?

• Pure FP	Always OK
• Non-termination	Always OK
• Exceptions	Always OK
• I/O	Have to think about it
• Mutation	Have to think about it

Cost Graphs



Cost Graphs

separate the generation of work from scheduling it onto processors (according to Brent's theorem)

Functors

```
signature TWO_PLAYERS =
sig
    structure Maxie  : PLAYER
    structure Minnie : PLAYER
    sharing type Maxie.Game.state = Minnie.Game.state
    sharing type Maxie.Game.move = Minnie.Game.move
end

functor Referee (P : TWO_PLAYERS) : sig val go : unit -> unit end
```

Functors

allow code reuse via
abstraction over both types and values

Persistent Data Structures

```
val choose : Game.player -> edge Seq.seq -> edge =
    fn Game.Maxie => SeqUtils.reduce1 EdgeUtils.max
     | Game.Minnie => SeqUtils.reduce1 EdgeUtils.min

fun search (depth : int) (s : Game.state) : edge =
    choose (Game.player s)
           (Seq.map
            (fn m => (m , evaluate (depth - 1) (Game.make_move (s,m))))
            (Game.moves s))

and evaluate (depth : int) (s : Game.state) : Game.est =
    case Game.status s of
        Game.Over v => Game.Definitely v
      | Game.In_play =>
           (case depth of
                0 => Game.estimate s
              | _ => edgeval(search depth s))
```

Persistent Data Structures

are good for backtracking

Rep. Invariants

```
functor RBTDict(Key : ORDERED) : DICT =
struct

  datatype 'v tree =
      Empty
    | Node of 'v tree * (color * (Key.t * 'v)) * 'v tree

  (* representation invariant: is a RBT *)
  type 'v dict = 'v tree

  ...

end
```

Rep. Invariants

abstract types localize reasoning about invariants to the implementation of the abstraction

Rep. Invariants

```
fun insert d (k, v) =
    let
        (* Root is Red,   both RBT --> ARBT
           Root is Black, at most one ARBT, and the other(s) RBT --> RBT
        *)
        fun balance p =
            case p of
                (Node(Node (a , (Red, x) , b) , (Red , y) , c) , (Black , z) , d) =>
                    Node (Node (a , (Black , x) , b) , (Red , y), Node (c , (Black , z) , d))
              | (Node(a , (Red , x) , Node (b , (Red , y) , c)) , (Black , z) , d) =>
                    Node (Node (a , (Black , x) , b) , (Red , y), Node (c , (Black , z) , d))
              | (a , (Black , x) , Node(Node (b , (Red, y) , c) , (Red , z) , d)) =>
                    Node (Node (a , (Black , x) , b) , (Red , y), Node (c , (Black , z) , d))
              | (a , (Black , x) , Node(b , (Red , y) , Node (c , (Red , z) , d))) =>
                    Node (Node (a , (Black , x) , b) , (Red , y), Node (c , (Black , z) , d))
              | _ => Node p
        (* if t is an ARBT then blackenRoot t is a RBT *)
        fun blackenRoot t = case t of Leaf => Leaf
           | Node (l , (_ , x) , r) => Node (l , (Black , x) , r)
        (* if d is an RBT[Red]   then ins d is an ARBT
           if d is an RBT[Black] then ins d is an RBT
           *)
        fun ins d =
            case d of
                Leaf => Node (empty, (Red, (k, v)), empty)
              | Node (l, (c , (k', v')), r) =>
                    case Key.compare (k,k') of
                        EQUAL => Node (l, (c, (k, v)), r)
                      | LESS => balance (ins l, (c , (k', v')), r)
                      | GREATER => balance (l, (c , (k', v')), ins r)
    in blackenRoot (ins d)
    end
```

Type classes

```
signature ORDERED =
sig
    type t
    val compare : t * t -> order
end

structure IntLt : ORDERED =
struct
    type t = int
    val compare = Int.compare
end
```

Type classes

describe a type equipped with a (non-exhaustive) collection of operations

Type-directed programming

```
type grades = (string * (int list * string list)) list

val db : grades =
 [("drl", ([95,99,98], (["y","n","y","y"]))),
  ("iev", ([99,99,99], (["y","y","y","y","y"]))),
  ("nkindber", ([97,99,99], (["y","y","y","y","y"]))),
  ("srikrish", ([100,100,100], (["n","n","n","y","n"]))),
  ("rmemon", ([98,98,98], (["y","y","y","y","y"]))),
  ("rmurcek", ([98,100,98], (["y","y","y","y","y"])))
  ]
structure SG =
SerializeList(
 SerializePair(
  struct
   structure S1 = SerializeString
   structure S2 = SerializePair(
                   struct
                    structure S1 = SerializeList(SerializeInt)
                    structure S2 = SerializeList(SerializeString)
                   end)
  end))
```

Abstract types

```
signature SEQUENCE =
sig
  type 'a seq
  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val reduce : (('a * 'a) -> 'a) -> 'a -> 'a seq -> 'a
  val tabulate : (int -> 'a) -> int -> 'a seq
  val nth     : 'a seq -> int -> 'a
end
```

Abstract types

```
signature SPACE = sig

  (* scalars and operations on them *)
  structure Scalar : SCALAR
  type scalar = Scalar.scalar

  (* points and vectors and operations on them *)
  type point
  type vec

  val vecToString : vec -> string
  val pointToString : point -> string

  val ++ : vec * vec -> vec    (* v1 ++ v2 evaluates to the sum of the vectors *)
  val ** : vec * scalar -> vec (* v ** c evaluates to the scalar product of v with c *)
  val // : vec * scalar -> vec (* v // c evaluates to the scalar product of v with (1/c) *)

  val --> : point * point -> vec (* X --> Y is the vector from X to Y *)

  ...
end
```

Abstract types

allow clients and implementatons
to evolve separately

localize reasoning

let you divide and conquer your problems

Sequences

```
fun accelerations (bodies : body Seq.seq)
    : vec Seq.seq =
    Seq.map
    (fn b1 =>
      sum bodies (fn b2 => accOn (b1 , b2)))
    bodies
```

Sequences

provide parallelism from mathematical transformations on bulk data

Exceptions

```
exception NoSubset
fun subset_sum_exn (l : int list, s : int) : int list =
    case l
      of [] => (case s of
                    0 => []
                  | _ => raise NoSubset)
       | x::xs => (x :: subset_sum_exn (xs, s - x))
                  handle NoSubset => subset_sum_exn (xs , s)
```

Exceptions

are equivalent to options,
and therefore OK in parallel code

are useful for signaling errors

are useful for backtracking

Staging

```
infixr 8 OR
infixr 9 THEN
fun m1 OR m2 = fn cs => fn k => m1 cs k orelse m2 cs k
fun m1 THEN m2 = fn cs => fn k => m1 cs (fn cs' => m2 cs' k)
fun REPEATEDLY m = fn cs => fn k =>
    let fun repeat cs' = k cs' orelse m cs' repeat
    in
        repeat cs
    end

fun match (r : regexp) : matcher =
    case r of
        Zero => FAIL
      | One => NULL
      | Char c => LITERALLY c
      | Plus (r1,r2) => match r1 OR match r2
      | Times (r1,r2) => match r1 THEN match r2
      | Star r => REPEATEDLY (match r)
```

Staging

curried functions can do useful work
before getting all of their arguments

Regexps

```
fun match r cs k =
    case r of
        Zero => false
      | One => k cs
      | Char c => (case cs of
                     []  => false
                   | c' :: cs' => c = c' andalso k cs')
  | Plus (r1,r2) => match r1 cs k orelse match r2 cs k
  | Times (r1,r2) => match r1 cs (fn cs' => match r2 cs' k)
  | Star r =>
        let fun matchstar cs' = k cs' orelse match r cs' matchstar
        in
            matchstar cs
        end
```

Regexps

it takes mathematical sophistication
to get code right

higher-order functions reify control flow as data,
so you can manipulate it

Functions as values

```
(* represent c_0 x^0 + c_1 x + c_2 x^2 + ...
   by the function that maps
   the natural number i to the coefficient c_i
*)
type poly = int -> rat


fun differentiate (p : poly) : poly =
    fn i => ((i + 1) // 1) ** (p (i + 1))
```

Functions as values

```
functor FunDict (K : ORDERED) : DICT =
struct
  structure Key = K

  datatype 'v func = Func of (Key.t -> 'v option)

  type 'v dict = 'v func

  (* Purpose: Returns a dictionary that contains no mappings *)
  val empty = Func (fn _ => NONE)

  (* Purpose: Inserts an element into a dictionary *)
  fun insert (Func f) (k, v) =
    Func
    (fn k' =>
      case Key.compare (k, k') of
        EQUAL => SOME v
      | _ => f k')

  (* Purpose: Finds an element in a dictionary *)
  fun lookup (Func f) k = f k
end
```

Functions as values

some values are (natural numers, lists, trees, …)

some values do (functions, streams, …)

“Do be do be do” -- Sinatra

Functions as args

```
val maxT : int tree -> int = reduce Int.max minint
val maxAll : (int tree) tree -> int = maxT o map maxT

fun withSuffixes (t : int tree) : (int * int tree) tree
    = zip (t, suffixes t)

val bestGain : int tree -> int =
      maxAll
    o (map (fn (buy,sells) => (map (fn sell => sell - buy) sells)))
    o withSuffixes
```

Functions as args

higher-order functions abstract
patterns of computation

express algorithms using function composition

Functions as args

```
fun plus (m1 : matrix, m2 : matrix) : matrix =
    ListPair.map (ListPair.map op++) (m1, m2)

fun summat (ms : matrix list) : matrix =
    case ms of
      nil => zed (0, 0)
    | m::ms => List.foldl plus m ms

fun allpairs (l1 : 'a list, l2 : 'b list) : ('a * 'b) list list =
    List.map (fn a => List.map (fn b => (a, b)) l2) l1

fun outerprod (v1 : rat list, v2 : rat list) : matrix =
      map (map Rational.times) (allpairs (v1, v2))

fun transpose (m : 'a list list) : 'a list list =
    case m of
       [] => []
     | x :: xs => ListPair.map (fn (a, b) => a :: b) (x, transpose xs)

fun times (m1 : matrix, m2 : matrix) : matrix =
    summat (ListPair.map outerprod (transpose m1, m2))
```

Abstract patterns

```
functor GoogleMapReduce (Key : ORDERED) :
sig
    structure D : DICT
    val gmapred : ('a -> (D.Key.t * 'v) Seq.seq)
                  -> ('v * 'v -> 'v)
                  -> 'a Seq.seq
                  -> 'v D.dict
end

val wordCounts : string Seq.seq -> int MR.D.dict =
    MR.gmapred (fn s => Seq.map (fn w => (w, 1))
                                (SeqUtils.words s))
               (op+)
```

HOFs as “iterators”

```
signature MAPABLE =
sig
    type 'a collection
    val mapreduce :  ('a -> 'b)
                  -> ('b * 'b -> 'b)
                  -> 'a collection
                  -> 'b
end
```

Datatypes

```
datatype block = A | B | C

datatype move =
    PickUpFromBlock of block * block
  | PutOnBlock of block * block
  | PickUpFromTable of block
  | PutOnTable of block

datatype fact =
    Free of block
  | On of block * block
  | OnTable of block
  | HandIsEmpty
  | HandHolds of block

type state = fact list
```

Datatypes

represent your problem

make error states unrepresentable

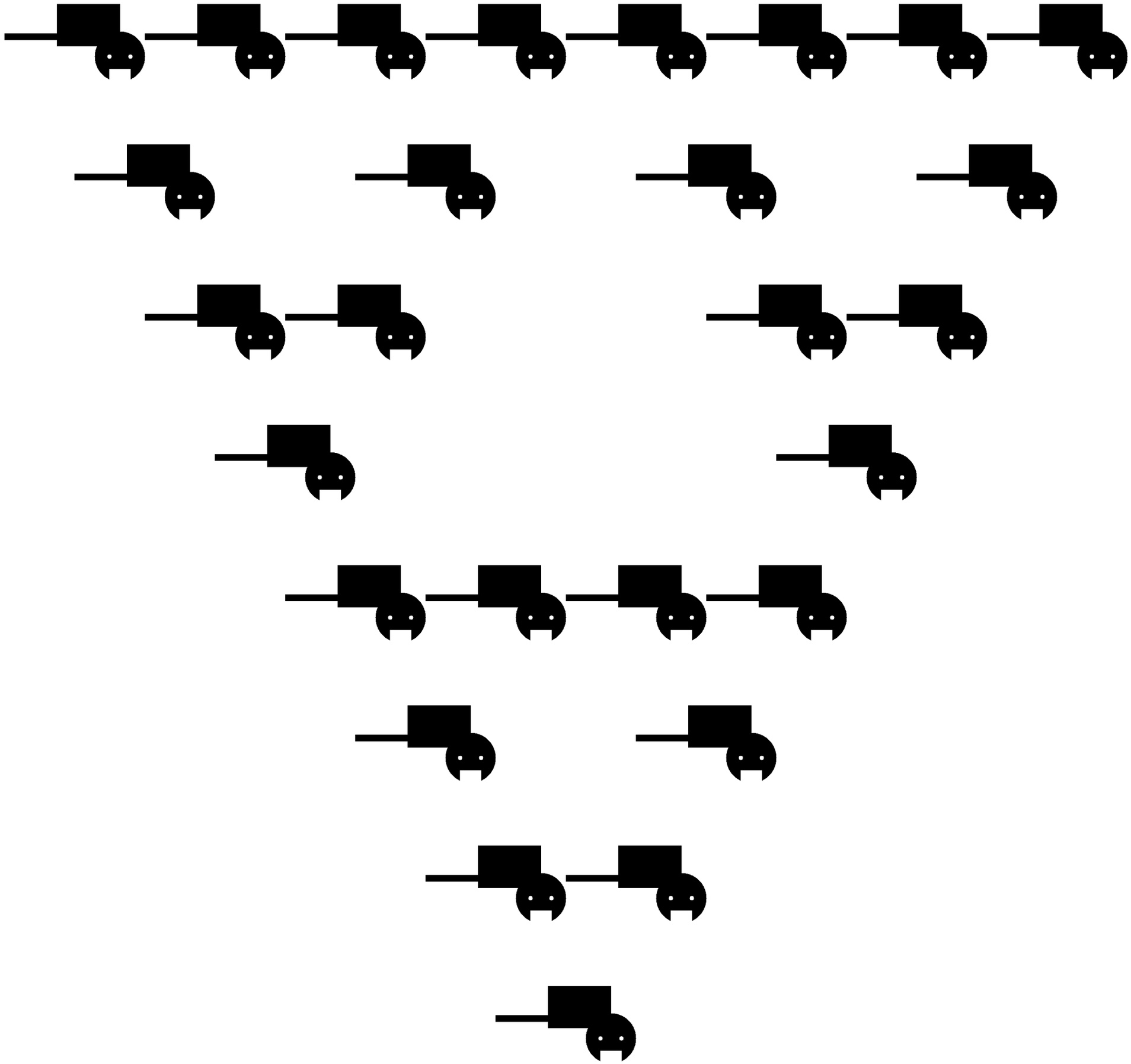
recursive functions come from recursive data

Datatypes

```
datatype shape =
    Rect of point * point (* bottom-left and upper-right *)
  | Disc of point * int (* center and radius *)
  | Intersection of shape * shape
  | Union of shape * shape
  | Without of shape * shape
  | Translate of shape * (int * int)
  | ScaleDown of shape * (int * int) (* x factor, y factor *)
  | ScaleUp of shape * (int * int) (* x factor, y factor *)
```

Datatypes

```
(* Purpose: contains(s,p) == true if p is in the shape,
   or false otherwise *)
fun contains (s : shape, p as (x,y) : point) : bool =
    case s of
        Rect ((xmin,ymin),(xmax,ymax)) =>
        xmin <= x andalso x <= xmax andalso
        ymin <= y andalso y <= ymax
  | Disc ((cx,cy),r) =>
        square(x - cx) + square(y - cy) <= square r
  | Intersection(s1,s2) => contains(s1,p) andalso contains(s2,p)
  | Union(s1,s2) => contains(s1,p) orelse contains(s2,p)
  | Without (s1,s2) => contains (s1,p) andalso (not (contains (s2,p)))
  | Translate (s,(tx,ty)) => contains(s , (x - tx, y - ty))
  | ScaleDown (s,(xf,yf)) => contains (s , (xf * x, yf * y))
  | ScaleUp (s,(xf,yf)) => contains (s , (x div xf, y div yf))
```



Work/Span

```
datatype tree =
    Empty
  | Node of tree * int * tree

fun splitAt (t : tree , bound : int) : tree * tree =
 case t of
     Empty => (Empty , Empty)
   | Node (l , x , r) =>
      (case bound < x of
           true => let val (ll , lr) = splitAt (l , bound)
                   in (ll , Node (lr , x , r))
                   end
         | false => let val (rl , rr) = splitAt (r , bound)
                    in (Node (l , x , rl) , rr)
                    end)

fun merge (t1 : tree , t2 : tree) : tree =
 case t1 of
     Empty => t2
   | Node (l1 , x , r1) =>
      let val (l2 , r2) = splitAt (t2 , x)
      in
          Node (merge (l1 , l2) ,
                x,
                merge (r1 , r2))
      end

fun mergesort (t : tree) : tree =
    case t of
        Empty => Empty
      | Node (l , x , r) =>
          merge(merge (mergesort l , mergesort r),
                Node(Empty,x,Empty))
```

Work/Span

trees are better than lists for parallelism

you can reason abstractly about
both sequential and parallel complexity

Pairs

“harder” problems can be easier to solve

```
(* Purpose: reverse l in linear time *)
fun revTwoPiles (l : int list, r : int list) : int list =
    case l of
        [] => r
      | (x :: xs) => revTwoPiles(xs , x :: r)

fun fastReverse (l : int list) : int list = revTwoPiles(l , [])
```

Lists

```
(* Purpose: sum the numbers in the list *)
fun sum (l : int list) : int =
    case l of
         [] => 0
       | x :: xs => x + sum xs
val 15 = sum [1,2,3,4,5]
```

Lists

(once upon a time,
you didn’t know how to write
simple recursive functions)

Recursion

```
(* Purpose: double the number n
   Examples:
   double 0 ==> 0
   double 3 ==> 6
*)
fun double (n : int) : int =
    case n of
        0 => 0
      | _ => 2 + (double (n - 1))

(* Tests *)
val 0 = double 0
val 6 = double 3
```

Recursion

(once upon a time,
you didn’t know how to write
simple recursive functions)

Typing and Eval.

```
2 : int
1 + 1 : int
(1 + 2) * (3 + 4) : int
"I am" : string
"I am" ^ " the walrus"   : string
intToString 5 : string
"the walrus" + 1      is ill-typed
```

Typing and Eval.

(or what the basic ingredients of a program are)

Deterministic Parallelism

```
fun sum (r : int seq) : int =
    reduce (fn (x,y) => x + y) 0 r

fun count (s : int seq seq) : int =
    sum (map sum s)
```

Deterministic Parallelism

In the first lecture, this code was mysterious!

Now, it's (hopefully) the easiest thing in the world.

Course Objectives

programming: datatypes, functions, exceptions, sequences, references, streams, ...

verification: proofs by induction, using the language-based evaluation model

analysis: recurrences for work and span; big-O

structuring large programs: abstract types, functors

Practice Exam: will be posted soon

Office hours: Next Tuesday and Thursday

Exam review: Wednesday, May 9
4-5:30pm
here

Exam: Friday, May 11 8:30 - 11:30 a.m.
DH 2210 & DH 2315(here)
cumulative, but biased towards 2nd half
1 sheet of notes

Quo vadimus

You might also like

15-210: Parallel data structures and algorithms

15-312: Principles of Programming Languages

15-317: Constructive Logic

80-413: Category Theory

Big uses of FP

Traditional:

- Compilers for functional languages:
  most FP compilers are implemented
  mostly in themselves
- Theorem provers, hardware/software verification
  Twelf,Agda,Coq,Isabelle,ACL2,…

New: Finance
Jane Street Capital, Credit Suisse,
Standard Charter, …

Tools you might like

- MLton: **fast** sequential execution of SML
- Manticore (experimental): parallelism

Better libraries/tools/ecosystem:

- OCaml: very similar to SML
- Haskell: call-by-name (with memoization)
  monadic effects
  otherwise pretty similar to ML
  good implementation of parallelism
- Scheme (Racket): untyped

It’s not all-or-nothing any more:

FP in “other” languages

(Other = not SML, OCaml, Haskell,
Lisp, Scheme, Erlang, …)

```
datatype exp =
    Int of int
  | Plus of exp * exp

fun eval e =
    case e of
        Int i => i
      | Plus (e1 , e2) => eval e1 + eval e2
```

```
class tm:
    pass
class Int(tm):
  def __init__(self, x):
      self.x = x
class Plus(tm):
  def __init__(self, e1, e2):
    self.e1 = e1
    self.e2 = e2
```

```
def eval(tm):
  try: raise tm
  except Int: return tm.x
  except Plus: return eval(tm.e1) + eval(tm.e2)
```

(Simmons, Beckman, Murphy VII, SIGBOVIK 2010)

Python doesn’t have pattern matching, you say?

Any value can be raised as an exception,
and you can case on the class of the exception

It’s not all-or-nothing any more:

FP in “other” languages (seriously)

FP in other languages

- F# (ML + objects) in **Microsoft Visual Studio**
- Scala (“object-oriented ML”) on JVM

- Garbage collection
  (automatic memory management)
- Functions as values in
  C# (“delegates”)
  Python, Ruby, JavaScript, …
- Polymorphism
  “Generics” in Java and C#

Course Objectives

programming: datatypes, functions, exceptions, sequences, references, streams, ...

verification: proofs by induction, using the language-based evaluation model

analysis: recurrences for work and span; big-O

structuring large programs: abstract types, functors

Take-away Skills

programming:

think about mathematical transformations on data

control your use of effects

verification: reason inductively about invariants

analysis: use big-O to guide your coding

structuring large programs: hide information

code is math

code is art

code is math

parallelism: mathematical transformations on data

verification: code is subject to mathematical analysis

code is art

code can be beautiful

code is for people: a good program explains an idea

code can change the way you think

code is math

code is art