

15-150

Spring 2025

unwind

Theorems About Code

Theorem

$$\text{map } f (X @ Y) \cong (\text{map } f X) @ (\text{map } f Y)$$

for any pure total function f and list values X and Y with correct types.

In this course:

We would prove the theorem by structural induction on X , and by looking at the specific implementations of map and $@$.

Theorems For Free

Theorem

$$\text{map } f (X \oplus Y) \cong (\text{map } f X) \oplus (\text{map } f Y)$$

for any pure total function f and list values X and Y with correct types.

One can do better (using Reynolds' *Abstraction Theorem*):

A proof does *not need* to know the code for $@$.

Theorem actually *holds for any* pure total operator $\oplus$ with

$(\text{op } \oplus) : 'a \text{ list} * 'a \text{ list} \rightarrow 'a \text{ list} .$

Uncomputability

Diagonalization

There is no algorithm **H** to decide whether $(f \ x)$ will return a value when evaluated.

```
fun diag x = if H(diag, x) then loop() else x
```

Reduction

There is no algorithm **E** to decide whether $f \cong g$.

```
fun H(f, x) = E(fn y => (f x; y), fn y => y)
```

Uncomputability

important to understand (one's) limits

Parsing

```
(* Grammar  E --> lambda X.E  | (E E) | X
              X --> <any alphanumeric string> *)

datatype token = LAMBDA | LPAREN | RPAREN | ID of string | DOT

datatype exp =    Fun of string * exp  | App of exp * exp
                 | Var of string

(* parseExp : token list -> (exp * token list -> 'a) -> 'a  *)

fun parseExp ((ID x)::ts) k = k(Var x, ts)
  | parseExp (LPAREN::ts) k =
      parseExp ts
      (fn (e1, t1) =>
          parseExp t1
          (fn (e2, RPAREN::t2) => k(App(e1,e2), t2)
            | _ => raise ParseError))

  | parseExp (LAMBDA::(ID x)::DOT::ts) k =
      parseExp ts (fn (e, ts') => k(Fun(x,e), ts'))

  | parseExp _ _ = raise ParseError
```

Parsing

context-free grammars as tools for
automatically generating parsers

Streams

```
signature STREAM =
sig
  type 'a stream      (* abstract *)
  datatype 'a front = Empty | Cons of 'a * 'a stream (* concrete *)

  val delay : (unit -> 'a front) -> 'a stream
  val expose : 'a stream -> 'a front
  val empty : 'a stream
  ...
end
```

Streams

```
signature STREAM =  
sig  
  type 'a stream      (* abstract *)  
  datatype 'a front = Empty | Cons of 'a * 'a stream (* concrete *)  
  
  val delay : (unit -> 'a front) -> 'a stream  
  val expose : 'a stream -> 'a front  
  val empty : 'a stream  
  ...  
end
```

Transformations of Infinite Data:

```
fun sieve s = S.delay (fn () => sieve' (S.expose s))  
and sieve' S.Empty = S.Empty  
  | sieve' (S.Cons(p, s)) =  
    S.Cons(p, sieve (S.filter (notDivides p) s))  
  
(* All the primes as a stream: *)  
val primes = sieve (natsFrom 2)
```

Stream Implementation

```
signature STREAM =
sig
  type 'a stream      (* abstract *)
  datatype 'a front = Empty | Cons of 'a * 'a stream (* concrete *)

  val delay : (unit -> 'a front) -> 'a stream
  val expose : 'a stream -> 'a front
  val empty : 'a stream
  ...
end

structure S :> STREAM =
struct
  datatype 'a stream = Stream of unit -> 'a front
  and      'a front = Empty | Cons of 'a * 'a stream

  fun delay (d) = Stream (d)
  fun expose (Stream (d)) = d()
  val empty = Stream (fn () => Empty)
  ...
end
```

Memoizing Stream

Can do so hidden from the user, in `delay`:

```
fun delay (d : unit -> 'a front) : 'a stream =  
  let  
    val memoCell = ref d      (* temporarily *)  
    fun memoFun () =          (* called at most once *)  
      let  
        val r = d ()  
      in  
        (memoCell := (fn () => r); r)  
      end  
    val _ = memoCell := memoFun  
  in  
    Stream (fn () => !memoCell())  
  end  
fun expose (Stream d) = d()  
...
```

Benign Effects

```
type graph = int -> int list

(* reachable : graph -> int * int -> bool
   reachable g (x,y) ==> true if y is reachable from x,
                           false otherwise.
*)
fun reachable (g:graph) (x:int, y:int) : bool =
  let
    val visited = ref [] (* more abstractly, "empty" *)
    fun dfs n = (n=y) orelse
      (not (member n (!visited))
       andalso
       (visited := n::(!visited);
        List.exists dfs (g n)))
  in
    dfs x
  end
```

Benign Effects

imperative implementation
that looks functional to clients

sometimes effects are faster or nicer

need to think about parallelism

Using Effects

	persistent	ephemeral
sequential	FP	OK
parallel	benign effects	concurrency

Mutation

```
fun update (f: 'a -> 'a) (r: 'a ref): unit =  
    r := f(!r)
```

```
fun deposit (n: int) (a: int ref): unit =  
    update (fn x => x + n) a
```

```
fun withdraw (n: int) (a: int ref): unit =  
    update (fn x => x - n) a
```

```
val account = ref 100
```

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

Mutation

can lead to race conditions

Sequences

signature SEQUENCE =

sig

type 'a seq (* abstract *)

val empty : unit -> 'a seq  **note**

val singleton : 'a -> 'a seq

val tabulate : (int -> 'a) -> int -> 'a seq

val nth : 'a seq -> int -> 'a

val length : 'a seq -> int

val drop : 'a seq -> int -> 'a seq

val take : 'a seq -> int -> 'a seq

val map : ('a -> 'b) -> 'a seq -> 'b seq

val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a

val filter : ('a -> bool) -> 'a seq -> 'a seq

val append : 'a seq * 'a seq -> 'a seq

val flatten : 'a seq seq -> 'a seq

If argument functions are **O(1)**:

... (* O(1) span: empty, singleton, tabulate, nth, length,
drop, take, map, append, ...

end

O(log n) span: reduce, filter, flatten, ... *)

Sequences

parallel-friendly ordered-collections ¹

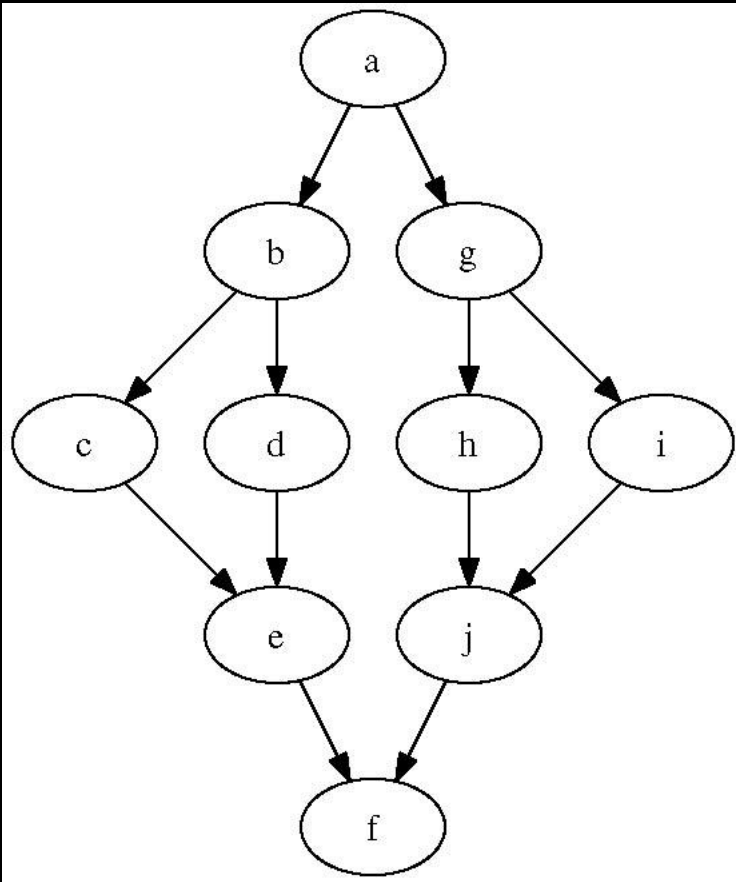
provide parallelism for mathematical
transformations on bulk data

¹ “ordered” doesn’t mean “sorted”, just that there is a first element, second element, etc.

Cost Graphs

$$(1 + 2) * (3 + 4)$$

$$W = 10$$
$$S = 5$$



Pebbling

CPUs

time ↓	1	a	
	2	b	g
	3	c	d
	4	h	i
	5	e	j
	6	f	

Cost Graphs

separate generation of work from scheduling

Brent's Theorem

An expression with work W and span S can be evaluated on a p -processor machine in time

$$\Omega(\max(W / p, S)).$$

Game Playing

```
signature PLAYER =
sig
    structure Game : GAME      (* parameter *)
    val next_move : Game.state -> Game.move
end

functor MiniMax (Settings : SETTINGS) : PLAYER = ...

signature TWO_PLAYERS =
sig
    structure Maxie : PLAYER  (* parameter *)
    structure Minnie : PLAYER (* parameter *)
    sharing Maxie.Game = Minnie.Game
end

functor Referee (P : TWO_PLAYERS) : GO = ...
```

Functors

```
signature DICT =
sig
  structure Key : ORDERED          (* parameter *)
  type 'a entry = Key.t * 'a      (* concrete *)
  type 'a dict          (* abstract *)
  val lookup : 'a dict -> Key.t -> 'a option
  ...
end

functor TreeDict (K : ORDERED) : DICT =
struct
  structure Key = K
  type 'a entry = Key.t * 'a
  datatype 'a dict = Empty | Node of 'a dict * 'a entry * 'a dict

  fun lookup tree key = ... Key.compare ...
  ...
end
```

Functors

```
signature DICT =
sig
  structure Key : ORDERED          (* parameter *)
  type 'a entry = Key.t * 'a      (* concrete *)
  type 'a dict          (* abstract *)
  val lookup : 'a dict -> Key.t -> 'a option
  ...
end

functor TreeDict (K : ORDERED) :> DICT where type Key.t = K.t =
struct
  structure Key = K
  type 'a entry = Key.t * 'a
  datatype 'a dict = Empty | Node of 'a dict * 'a entry * 'a dict

  fun lookup tree key = ... Key.compare ...
  ...
end
```

Functors

allow code reuse
by abstracting
over both types and values

Type Classes

```
signature ORDERED =  
sig  
  type t (* parameter *)  
  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 (not usually exhaustive)
collection of operations

Representation Invariants

```
fun insert (dict, entry as (key, datum)) =
  let
    fun ins Empty = Red(Empty, entry, Empty)
      | ins (Red(left, entry1 as (key1, _), right)) =
          (case String.compare (key, key1)
           of EQUAL => Red(left, entry, right)
            | LESS  => Red(ins left, entry1, right)
            | GREATER => Red(left, entry1, ins right))
      | ins (Black(left, entry1 as (key1, _), right)) =
          (case String.compare (key, key1)
           of EQUAL => Black(left, entry, right)
            | LESS  => restoreLeft(Black(ins left, entry1, right))
            | GREATER => restoreRight(Black(left, entry1, ins right)))
  in
    (case ins dict
     of Red (t as (Red _, _, _)) => Black t      (* re-color *)
      | Red (t as (_, _, Red _)) => Black t      (* re-color *)
      | dict => dict)
  end
```

Representation Invariants

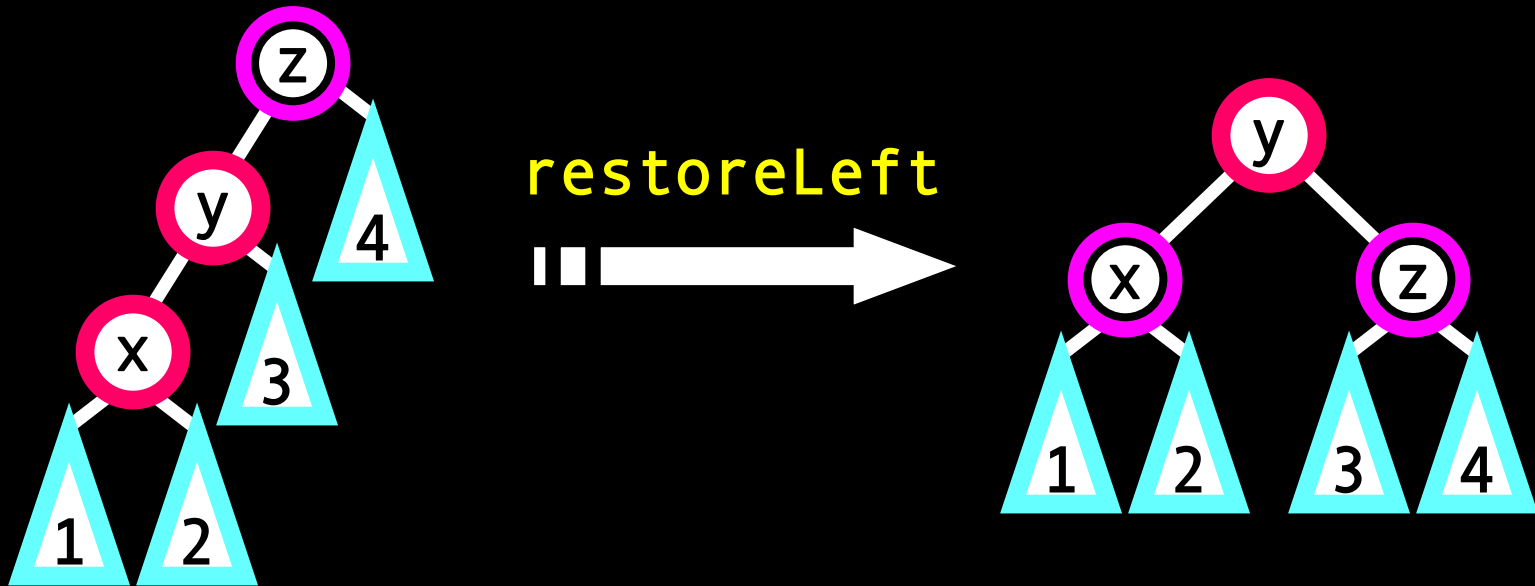
Red Black Tree

- 1) Tree is a binary search tree.
- 2) Children of a Red node are Black.
- 3) Every path from root to leaf has the same number of Black nodes.

Almost Red Black Tree

- 1) As before.
- 2) As above, except: Red root may have a Red child.
- 3) As before.

Patterns: Programming by Picture



```
fun restoreLeft (Black(Red(Red(d1, x, d2), y, d3), z, d4)) =  
    Red(Black(d1, x, d2), y, Black(d3, z, d4))  
| restoreLeft (Black(Red(d1, x, Red(d2, y, d3)), z, d4)) =  
    Red(Black(d1, x, d2), y, Black(d3, z, d4))  
| restoreLeft dict = dict
```

Alternate implementations

```
structure Q2 : QUEUE =
struct
  type 'a queue = 'a list * 'a list
  (* Abstraction Function for (f,b):
     f @ (rev b) represents queue elements
     in arrival order. *)

  val empty = (nil, nil)
  fun enq ((f,b), x) = (f, x::b)
  fun null (nil, nil) = true
    | _ = false
  exception Empty
  fun deq (nil, nil) = raise Empty
    | deq (x::f, b) = (x, (f,b))
    | deq (nil, b) = deq (rev b, nil)
end
```

Structures as concrete implementations of abstractions

```
structure Q1 : QUEUE =  
struct  
  type 'a queue = 'a list  
  (* Abstraction Function: list represents  
     queue elements in arrival order. *)  
  
  val empty = nil  
  fun enq (q,x) = q @ [x]  
  val null = List.null  
  exception Empty  
  fun deq nil = raise Empty  
    | deq (x::xs) = (x, xs)  
end
```

Signatures as interfaces for abstract datatypes

```
signature QUEUE =  
sig  
  type 'a queue          (* abstract type *)  
  
  val empty : 'a queue  
  val enq   : 'a queue * 'a -> 'a queue  
  val null  : 'a queue -> bool  
  
  exception Empty  
  
  (* deq (q) raises Empty if q is empty *)  
  val deq   : 'a queue -> 'a * 'a queue  
end
```

Modules for Queues

Staged Combinator-Based Regular Expression Matcher

```
fun match (Char(a)) = CHECK_FOR a
| match  One      = ACCEPT
| match  Zero     = REJECT
| match (Times(r1,r2)) = (match r1) THEN (match r2)
| match (Plus(r1,r2))  = (match r1) ORELSE (match r2)
| match (Star(r))      = REPEAT (match r)
```

Staged RegExp

```
infixr 8 ORELSE
infixr 9 THEN
fun m1 ORELSE 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 REPEAT m = fn cs => fn k =>
    let fun mstar cs' =
            k cs' orelse m cs' (fn cs'' => not (cs' = cs'')
                                andalso mstar cs'')
        in
            mstar cs
        end
end
```

```
fun match ((Char a) : regexp) : matcher = CHECK_FOR a
| match One = ACCEPT
| match Zero = REJECT
| match (Times (r1, r2)) = (match r1) THEN (match r2)
| match (Plus (r1, r2)) = (match r1) ORELSE (match r2)
| match (Star r) = REPEAT (match r)
```

Combinators

(* define combinators by pointwise principle *)

infixr ++

fun (f ++ g) (x : 'a) : int = f(x) + g(x)

fun MIN(f,g) (x : 'a) : int = Int.min(f x, g x)

fun square (x:int):int = x*x

fun double (x:int):int = 2*x

val quadratic = square ++ double

val lowest = MIN(square, double)

Combinators

(* define combinators by pointwise principle *)

infixr ++

fun (f ++ g) (x : 'a) : int = f(x) + g(x)

fun MIN(f,g) (x : 'a) : int = Int.min(f x, g x)

fun square (x:int):int = x*x

fun double (x:int):int = 2*x

val quadratic = square ++ double

val lowest = MIN(square, double)

**Functions
are
values!**

Staging

```
fun f (x : int) (y : int) : int =  
  let val z = horriblecomputation(x)  
  in z + y end
```

```
val partial : int -> int = f 10      (* FAST *)  
val res5 : int = partial 5           (* slow *)  
val res2 : int = partial 2           (* slow *)
```

```
fun f (x : int) : int -> int =  
  let val z = horriblecomputation(x)  
  in (fn (y:int) => z + y) end
```

```
val partial : int -> int = f 10      (* slow *)  
val res5 : int = partial 5           (* FAST *)  
val res2 : int = partial 2           (* FAST *)
```

Staging

```
fun f (x : int) (y : int) : int =  
  let val z = horriblecomputation(x)  
  in z + y end
```

```
val partial : int -> int = f 10      (* FAST *)  
val res5 : int = partial 5           (* slow *)  
val res2 : int = partial 2           (* slow *)
```

```
fun f (x : int) : int -> int =  
  let val z = horriblecomputation(x)  
  in (fn (y:int) => z + y) end
```

```
val partial : int -> int = f 10      (* slow *)  
val res5 : int = partial 5           (* FAST *)  
val res2 : int = partial 2           (* FAST *)
```

Staging

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

Regular Expressions

```
fun match (Char(a)) cs k =  
  (case cs of  
    nil => false  
    | c::cs' => a=c andalso k cs')  
| match One cs k = k cs  
| match Zero _ _ = false  
| match (Times(r1,r2)) cs k =  
  match r1 cs (fn cs' => match r2 cs' k)  
| match (Plus(r1,r2)) cs k =  
  match r1 cs k or else match r2 cs k  
| match (rs as Star(r)) cs k =  
  k cs or else  
  match r cs (fn cs' => not (cs = cs')  
    andalso match rs cs' k)
```

Regular Expressions

it takes mathematical sophistication
to get code right

higher-order functions encapsulate control flow
as data, so one can manipulate it

n-Queens with Exceptions

```
(*  
  addqueen:int * int * (int * int) list -> (int * int) list  
  try : int -> (int * int) list  
*)
```

```
exception Conflict
```

```
fun addqueen (i, n, Q) =  
  let fun try j =  
    (if conflict (i,j) Q then raise Conflict  
     else if i=n then (i,j)::Q  
     else addqueen (i+1, n, (i,j)::Q))  
      handle Conflict =>  
        (if j=n then raise Conflict  
         else try (j+1))  
    in  
      try 1  
    end
```

Exceptions

are useful for signaling errors

are useful for backtracking

n-Queens with Continuations

```
(*  
  addqueen : int * int * (int * int) list ->  
             ((int * int) list -> 'a) -> (unit -> 'a) -> 'a  
  try : int -> 'a  
*)
```

```
fun addqueen (i, n, Q) sc fc =  
  let fun try j =  
        let fun fc' () = if j=n then fc() else try (j+1)  
        in if (conflict (i,j) Q) then fc'()  
          else if i=n then sc((i,j)::Q)  
            else addqueen (i+1, n, (i,j)::Q) sc fc'  
        end  
      in  
        try 1  
      end
```

n-Queens with Options

```
(*  
  addqueen : int * int * (int * int) list ->  
              (int * int) list option  
  try : int -> (int * int) list option  
*)
```

```
fun addqueen (i, n, Q) =  
  let  
    fun try j =  
      (case (if conflict (i,j) Q then NONE  
             else if i=n then SOME((i,j)::Q)  
             else addqueen (i+1, n, (i,j)::Q))  
       of NONE => if j=n then NONE else try (j+1)  
        | result => result)  
    in  
      try 1  
    end
```

CPS

```
(* sum : int list -> int
   ENSURES:  sum L adds all the integers in L.
*)
```

```
fun sum [] = 0
  | sum (x::xs) = x + sum(xs)
```

```
(* ksum : int list -> (int -> 'a) -> 'a
   ENSURES:  ksum L k == k(sum L)
*)
```

```
fun ksum [] k = k(0)
  | ksum (x::xs) = ksum xs (fn s => k(x + s))
```

CPS

```
(* sum : int list -> int
   ENSURES:  sum L adds all the integers in L.
*)
```

```
fun sum [] = 0
  | sum (x::xs) = x + sum(xs)
```

```
(* ksum : int list -> (int -> 'a) -> 'a
   ENSURES:  ksum L k == k(sum L)
*)
```

```
fun ksum [] k = k(0)
  | ksum (x::xs) = ksum xs (fn s => k(x + s))
```

Continuations

(* match inorder traversal of tree against list.
Stop as soon as there is a mismatch.

```
prefix : int tree -> int list -> (int list -> bool) -> bool)  
*)
```

```
fun prefix Empty L k = k(L)  
  | prefix (Node(t1, x, t2)) L k =  
    prefix t1 L  
      (fn nil => false  
        | y::L' => (x=y) andalso (prefix t2 L' k))
```

```
(* treematch : int tree -> int list -> bool *)
```

```
fun treematch T L = prefix T L List.null
```

Continuations

higher-order functions encapsulate control flow
as data, so one can manipulate it

Functions as Values

some values *are* (numbers, lists, trees, ...)

some values *do* (functions, streams, ...)

Higher-Order List Functions

```
(* map : ('a -> 'b) -> 'a list -> 'b list *)
```

```
fun map (f:'a -> 'b) ([]:'a list) : 'b list = []  
  | map f (x::xs) = (f x)::(map f xs)
```

```
(* foldr and foldl both have type
```

```
    ('a * 'b -> 'b) -> 'b -> 'a list -> 'b  *)
```

```
fun foldr f z [] = z
```

```
  | foldr f z (x::xs) = f(x, foldr f z xs)
```

```
fun foldl f z [] = z
```

```
  | foldl f z (x::xs) = foldl f (f(x,z)) xs
```

Currying

```
fun add (x : int, y : int) : int = x + y
```

add is bound to

```
fn (x:int, y:int) => x + y
```

```
(* Test *)
```

```
val 13 = add(6, 7)
```

```
fun addcur (x : int) (y : int) : int = x + y
```

addcur is bound to

```
fn (x:int) => fn (y:int) => x + y
```

```
(* Test *)
```

```
val 13 = addcur 6 7
```

Functions as Values

(* dictionaries represented as functions *)

```
datatype 'v dict = Func of string -> 'v option
```

```
val empty = Func (fn _ => NONE)
```

```
fun insert (Func f) (k, v) =  
  Func  
    (fn k' => (case String.compare(k, k') of  
                 EQUAL => SOME v  
                 | _    => f k'))
```

```
fun lookup (Func f) key = f key
```

Functions as Values

```
(* represent the polynomial  
     $c_0 + c_1x + c_2x^2 + c_3x^3 + \dots$   
    by the function that maps  
    natural number  $i$  to the coefficient  $c_i$   
    *)
```

```
type poly = int -> rat    (more general numbers)
```

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

Functions as Arguments

```
fun square (x : int) : int = x * x
```

```
(* sqrf : (int -> int) * int -> int
```

```
    REQUIRES: true
```

```
    ENSURES:  sqrf (f, x) returns f(x)*f(x)
```

```
*)
```

```
fun sqrf (f : int -> int, x : int) : int =
```

```
    square (f x)
```

```
(* Test *)
```

```
val 81 = sqrf (fn n => n + 2, 7)
```

Anonymous Functions

(fn (x : int) => x + x) : int -> int

“lambda”

(fn x => x) : 'a -> 'a

Polymorphism (abstract patterns)

```
datatype 'a option = NONE | SOME of 'a

fun lookup (eq : 'a * 'a -> bool,
           x : 'a,
           L : ('a * 'b) list) : 'b option =
  (case L of
    [] => NONE
  | ((a,b)::rest) =>
    if eq(a,x) then SOME(b)
    else lookup(eq,x,rest))
```

Polymorphism (abstract patterns)

```
datatype 'a list = nil | :: of 'a * 'a list
infixr ::
```

```
datatype 'a tree =      Empty
                      | Node of 'a tree * 'a * 'a tree
```

```
(* trav : 'a tree -> 'a list
   REQUIRES: true
   ENSURES: trav(T) ==> elements in inorder traversal
*)
```

```
fun trav (Empty : 'a tree) : 'a list = nil
  | trav (Node(t1,x,t2)) = trav(t1) @ (x :: trav(t2))
```

Transforming Data

```
(* flatten: tree -> int list
   REQUIRES: true
   ENSURES: flatten(T) ==> elements in inorder traversal
*)
```

```
fun flatten(Leaf(x) : tree) : int list = [x]
  | flatten(Node(t1, t2)) = flatten(t1) @ flatten(t2)
```

```
(* flatten2 : tree * int list -> int list
   REQUIRES: true
   ENSURES: flatten2(T, acc) == flatten(T) @ acc
*)
```

```
fun flatten2(Leaf(x):tree, acc:int list):int list = x::acc
  | flatten2(Node(t1,t2), acc) =
    flatten2(t1, flatten2(t2, acc))
```

Datatypes

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

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

Datatypes

recursive functions
come from
recursive data

motivated by recursive transformations

Datatypes

represent recursive data

provide principles of structural induction

Datatypes

represent the problem

make error states impossible at runtime

Work and Span

```
fun SplitAt (x : int, Empty : tree) : tree * tree = (Empty, Empty)
| SplitAt (x, Node(left, y, right)) =
  (case compare(x, y) of
    LESS => let val (t1, t2) = SplitAt(x, left)
             in (t1, Node(t2, y, right))
             end
    | _   => let val (t1, t2) = SplitAt(x, right)
             in (Node(left, y, t1), t2)
             end)
```

```
fun Merge (Empty : tree, t2 : tree) : tree = t2
| Merge (Node(l1,x,r1), t2) =
  let val (l2, r2) = SplitAt(x, t2)
  in
    Node(Merge(l1, l2), x, Merge(r1, r2))
  end
```

```
fun Msort (Empty : tree) : tree = Empty
| Msort (Node(left, x, right)) =
  Ins (x, Merge(Msort left, Msort right))
```

```

fun Ins (x : int, Empty : tree) : tree = Node(Empty, x, Empty)
  | Ins (x, Node(t1, y, t2)) = (case compare(x,y) of
                                GREATER => Node(t1, y, Ins(x, t2))
                                | _      => Node(Ins(x, t1), y, t2))

```

```

fun SplitAt (x : int, Empty : tree) : tree * tree = (Empty, Empty)
  | SplitAt (x, Node(left, y, right)) =
    (case compare(x, y) of
      LESS => let val (t1, t2) = SplitAt(x, left)
               in (t1, Node(t2, y, right))
               end
      | _   => let val (t1, t2) = SplitAt(x, right)
               in (Node(left, y, t1), t2)
               end)

```

```

fun Merge (Empty : tree, t2 : tree) : tree = t2
  | Merge (Node(l1,x,r1), t2) =
    let val (l2, r2) = SplitAt(x, t2)
    in
      Node(Merge(l1, l2), x, Merge(r1, r2))
    end

```

$$W = O(n \log n)$$

$$S = O((\log n)^3)$$

(with rebalancing)

```

fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
    Ins(x, Merge(Msort left, Msort right))

```

parallel-friendly

Binary Search Trees

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

fun Ins (x : int, Empty : tree) : tree =
    Node(Empty, x, Empty)
  | Ins (x, Node(t1, y, t2)) =
    (case compare(x,y) of
      GREATER => Node(t1, y, Ins(x, t2))
    | _       => Node(Ins(x, t1), y, t2))
```

Work and Span

can reason abstractly about both
sequential and parallel complexity

trees are more parallel-friendly than lists

“harder” problems can be faster

```
(*  rev : int list -> int list
    rev (L) reverses L          *)
fun rev ([]:int list):int list = []
  | rev (x::xs) = rev(xs) @ [x]
```

```
(*  trev : int list * int list -> int list
    trev (L, acc) == rev(L) @ acc      *)
fun trev ([]:int list, acc:int list):int list = acc
  | trev (x::xs, acc) = trev(xs, x::acc)
```

```
(*  fastrev : int list -> int list  *)
fun fastrev (L:int list):int list = trev(L, [])
```

Tail-Recursion

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) ==> length of list L
*)
fun length(nil : int list) : int = 0
  | length(_::L : int list) : int = 1 + length(L)

(* length2 : int list * int -> int
   REQUIRES: true
   ENSURES: length2(L, acc) == length(L) + acc
*)
fun length2(nil : int list, acc : int) : int = acc
  | length2(_::L : int list, acc : int) : int =
    length2(L, 1 + acc)
```

Tail-Recursion

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) ==> length of list L
*)
fun length(nil : int list) : int = 0
  | length(_::L : int list) : int = 1 + length(L)

(* length2 : int list * int -> int
   REQUIRES: true
   ENSURES:
       length2(L, acc) == length(L) + acc
*)
fun length2(nil : int list, acc : int) : int = acc
  | length2(_::L : int list, acc : int) : int =
    length2(L, 1 + acc)
```

Tail-Recursion

(once upon a time,
you didn't know what
extensional equivalence meant

==

≈

)

Lists

```
(*  sum : int list -> int
    REQUIRES:  true
    ENSURES:  sum(L) computes the
               sum of the elements in L.
*)
```

```
fun sum ([]:int list):int = 0
  | sum (x::xs) = x + sum(xs)
```

```
val 10 = sum [0,1,2,3,4]
```

Lists

A list of integers (an `int list`) is one of:

- `[]` (aka `nil`)
- `x :: xs` with `x : int`
and `xs : int list`

(and that's all)

Lists

(once upon a time,
you didn't know what
recursively defined lists were)

“harder” problems can be faster

```
(*  fib : int -> int
    REQUIRES:  n >= 0
    ENSURES:  fib(n) computes nth Fibonacci number
*)
```

```
fun fib (0:int):int = 1
  | fib (1:int):int = 1
  | fib (n:int):int = fib(n-1) + fib(n-2)
```

```
(*  fib2 : int -> int * int
    REQUIRES:  n >= 0
    ENSURES:  fib2(n) == (fib(n), fib(n-1))
*)
```

```
fun fib2 (0:int):int*int = (1, 0)
  | fib2 (n) = let val (f1, f2) = fib2(n-1)
               in (f1 + f2, f1) end
```

Recursion

```
(*  fact : int -> int
    REQUIRES:  n >= 0
    ENSURES:  fact(n) ==> n!
*)

fun fact(0:int):int = 1
  | fact(n:int):int = n * fact(n-1)

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

Recursion

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

Typing and Evaluation

`~4 : int`

`3.14 : real`

`true : bool`

`(1, "ab") : int * string`

`(fn (r:real) => 1 + round r) :
real -> int`

Typing and Evaluation

(the basic ingredients of a program)

Parallelism

```
type row = int Seq.seq
```

```
type room = row Seq.seq
```

```
fun sum (s : row) : int =  
    Seq.reduce (op +) 0 s
```

```
fun count (class : room) : int =  
    sum (Seq.map sum class)
```

Parallelism

In the first lecture, this code may have been mysterious. Now it is (we hope) clear.

In the first lecture, we used this code to count in $O(n)$ rather than $O(n^2)$ parallel time. Some of you suggested $O(\log n)$ might be possible. Indeed, that is the span.

Programming as Explanation

High expectation to explain precisely and concisely:

Types

REQUIRES & ENSURES

Proofs of correctness

Code

Analyze, Decompose & Fit, Prove

Course Objectives

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

verification: proofs by induction, stepping by evaluation, extensional equivalence

analysis: recurrences for work and span, big-O

structuring large programs: abstract types, functors

Skills

programming: think about mathematical transformations on data; control use of effects

verification: reason inductively about invariants

analysis: use big-O to guide your work

structuring large programs: hide information to give more robust guarantees

Where to next?

15-210: Parallel data structures and algorithms

15-312: Principles of Programming Languages

15-317: Constructive Logic

15-411: Compiler Design

15-451: Algorithms

15-814: Types and Programming Languages

80-413: Category Theory

Types, Arrows, Topology

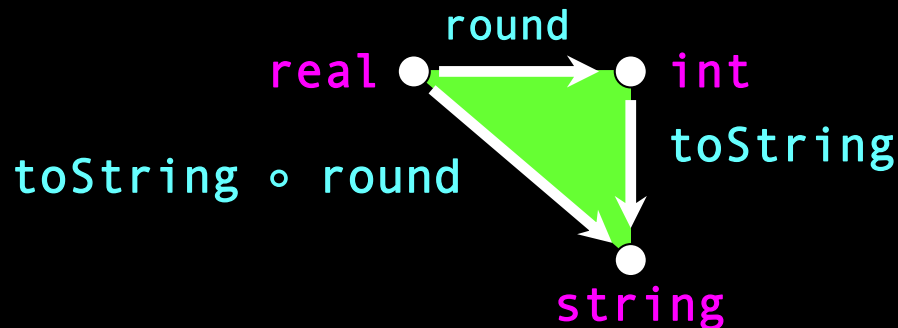
A type is a point:



A function is an edge:



Two composable functions define a triangle:



Three composable functions form a tetrahedron ...

Acting under Uncertainty

Connections between types, topology,
planning, games, uncertainty.

Real World

Air Traffic Control: Proving Correctness of TCAS
(Traffic Collision Avoidance System)

Finance: Jane Street Capital

Computer Industry: Microsoft (ML tools in MSR, F*)

Robotics: Planning with Uncertainty

code is math

transformations on data

subject to analysis

code is art

code can be beautiful

code can explain an idea

code can change the way
you think

(not so legible)

```
fun step x = . . .
```

```
fun iterate x 0 = SOME x  
  | iterate x n =  
    if (not ((step x) = NONE))  
    then let val SOME(y) = step x  
          in if y=1 then SOME(1)  
              else iterate y (n-1)  
          end  
    else NONE
```

```
val SOME 1 = iterate 10000000 101
```

(more elegant)

```
fun step x = . . .
```

```
fun iterate x 0 = SOME x
```

```
  | iterate x n =
```

```
    (case (step x) of
```

```
      SOME 1 => SOME 1
```

```
    | SOME y => iterate y (n-1)
```

```
    | NONE   => NONE)
```

```
val SOME 1 = iterate 1000000 101
```

(transformationally informative)

```
(fn (_, SOME 1) => SOME 1
  | (_, SOME y) => step y
  | _           => NONE)
```

function of iteration index and current value
(iteration index happens to be irrelevant)

(transformationally informative)

```
fun iterate x n =  
  foldl (fn (_, SOME 1) => SOME 1  
        | (_, SOME y) => step y  
        | _ => NONE)  
    (step x)  
    (time n)
```

think functionally

code well

Advice for the Final

Review

Lectures, Code & Notes (web), Labs, Homeworks

Potential Topics (in alphabetical order)

Abstraction, Analysis, Balance, Binding, Closure, Combinator, Context-Free Grammar, Continuation, Datatype, Decision Procedure, Effect, Environment, Equivalence, Evaluation, Exception, Function, Functor, Higher-Order Function, Induction, List, Parallelism, Pattern, Polymorphism, Recurrence, Recursion, Reference Cell, Regular Expression, Representation, Scope, Sequence, Signature, Sorting, Span, Staging, Streams & Laziness, Structural Induction, Structure, Tail Recursion, Totality, Tree, Type, Unit, Value, Work.

More Advice for the Final

Read

Spend time reading problems.

The code you write is short.

Sleep

Be well

!