Advice for the final

Review

Labs, homeworks, lectures

Practice

Types, recursion, work and span, induction, equivalence, totality, effects, signatures, structures, functors, streams, sequences, higher-order functions and cps

Sleep
If languages were cars ...

C was the great all-rounder: compact, powerful, goes everywhere, and reliable in situations where your life depends on it.

http://crashworks.org/if_programming_languages_were_vehicles/
If languages were cars ...

C++ is the new C — twice the power, twice the size, works in hostile environments, and if you try to use it without care and special training you will probably crash.

http://crashworks.org/if_programming_languages_were_vehicles/
If languages were cars ...

Java is another attempt to improve on C. It sort of gets the job done, but it's way slower, bulkier, spews pollution everywhere, ...

http://crashworks.org/if_programming_languages_were_vehicles/
If languages were cars ...

Python is great for everyday tasks: easy to drive, versatile, comes with all the conveniences built in. It isn't fast or sexy, but neither are your errands.

http://crashworks.org/if_programming_languages_were_vehicles/
If languages were cars ... 

OCaml is this funny shaped thing that Europeans like for some reason.

http://crashworks.org/if_programming_languages_were_vehicles/
What is SML?

• A functional programming language
  Computation = evaluation

• A typed language
  Only well-typed expressions are evaluated

• A polymorphic typed language
  well-typed expressions have a most general type

• A call-by-value language
  Function calls evaluate their arguments first
Benefits

• **referential transparency**
  • equivalent code is interchangeable, in all contexts
  • Simple compositional reasoning

• **mathematical foundations**
  • can use math and logic to prove correctness
  • use induction to analyze recursive code and data

• **functions are values**
  • can be used as data in lists, tuples, …
  • and argument or result of other functions
Principles

• Expressions must be well-typed.
  Well-typed expressions don't go wrong.

• Every function needs a specification.
  Well-specified programs are easy to understand.

• Every specification needs a proof.
  Well-proven programs do the right thing.

• Large programs should be designed as modules.
  Well-interfaced code is easier to maintain.
Principles

• **Data structures algorithms.**
  Good choice of data structure leads to better code.

• **Exploit parallelism.**
  Parallel code may run faster.

• **Strive for simplicity.**
  Programs should be as simple as possible, but no simpler.
Typing and Evaluation

\[ \sim 4 : \text{int} \]
\[ 3.14 : \text{real} \]
\[ \text{true} : \text{bool} \]
\[ (1, \text{“ab”}) : \text{int} \times \text{string} \]
\[ (\text{fn } (r:\text{real}) \rightarrow 1 + \text{round } r) : \text{real} \rightarrow \text{int} \]

The basic ingredients of a program
Once upon a time you did not even know how to write recursive programs
Recursion

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

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

(* Tests *)
val 0 = fact 0
val 6 = fact 3
"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
Once upon a time you did not know what recursively defined lists were
Lists

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

- `[]` (aka nil)

- \( x :: xs \) with \( x : \text{int} \) and \( xs : \text{int list} \)

(and that's all)
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]
Once upon a time you did not know what 
extensional equivalence meant
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)
“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, [])
Work and Span

can reason abstractly about both sequential and parallel complexity

Trees are more parallel-friendly than lists
**Binary Search Trees**

```plaintext
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)
```
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 \( \text{Ins} \ (x : \text{int}, \ \text{Empty} : \text{tree}) : \text{tree} = \text{Node}(\text{Empty}, x, \text{Empty}) \)

| \( \text{Ins} \ (x, \ \text{Node}(t1, y, t2)) = \) case compare\((x, y)\) of
| \( \text{GREATER} \Rightarrow \text{Node}(t1, y, \text{Ins}(x, t2)) \)
| \( \_ \Rightarrow \text{Node}(\text{Ins}(x, t1), y, t2) \)

fun \( \text{SplitAt} \ (x : \text{int}, \ \text{Empty} : \text{tree}) : \text{tree} \times \text{tree} = (\text{Empty}, \text{Empty}) \)

| \( \text{SplitAt} \ (x, \ \text{Node}(\text{left}, y, \text{right})) = \)
| case compare\((x, y)\) of
| \( \text{LESS} \Rightarrow \) let \( \text{val} \ (t1, t2) = \text{SplitAt}(x, \text{left}) \)
| \( \text{in} \ (t1, \text{Node}(t2, y, \text{right})) \)
| \( \_ \Rightarrow \) let \( \text{val} \ (t1, t2) = \text{SplitAt}(x, \text{right}) \)
| \( \text{in} \ (\text{Node}(\text{left}, y, t1), t2) \)
| \( \_ \)

fun \( \text{Merge} \ (\text{Empty} : \text{tree}, t2 : \text{tree}) : \text{tree} = t2 \)

| \( \text{Merge} \ (\text{Node}(l1, x, r1), t2) = \)
| let \( \text{val} \ (l2, r2) = \text{SplitAt}(x, t2) \)
| \( \text{in} \ \text{Node}(\text{Merge}(l1, l2), x, \text{Merge}(r1, r2)) \)
| \( \_ \)

fun \( \text{Msort} \ (\text{Empty} : \text{tree}) : \text{tree} = \text{Empty} \)

| \( \text{Msort} \ (\text{Node}(\text{left}, x, \text{right})) = \)
| \( \text{Ins}(x, \text{Merge}(\text{Msort left, Msort right})) \)

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

\( S = O((\log n)^3) \) (with rebalancing)

parallel-friendly
Recursive functions come from recursive data motivated by recursive transformations
Datatypes

represent the problem
make error states impossible
Datatypes

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

datatype tree =
  Leaf of int
  | Node of tree * tree
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))
Polymorphism
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))
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))
Functions as Values

some values are
(numbers, lists, trees, …)

some values do
(functions, streams, …)
Functions as Values

(* represent the polynomial
  \[ c_0 + c_1x + c_2x^2 + c_3x^3 + \ldots \]
  by the function that maps
  natural number \( i \) to the coefficient \( c_1 \)
*)

type poly = int -> rat

fun differentiate (p : poly) : poly =
  fn i => ((i + 1) // 1) ** (p (i + 1))
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) k = f k
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
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 [] = []
  | foldr f z (x:::xs) = f(x, foldr f z xs)

fun foldl f z [] = []
  | foldl f z (x:::xs) = foldl f (f(x,z)) xs
Continuations

Higher-order functions encapsulate control flow as data, so one can manipulate it
(* 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
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
Exceptions

useful for

signaling errors
backtracking
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
Regular Expressions

it takes mathematical sophistication to get code right

higher-order functions encapsulate control flow as data, so one can manipulate it
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 orelse match r2 cs k
| match (rs as Star(r)) cs k =
  k cs orelse
  match r cs (fn cs' => not (cs = cs')
                andalso match rs cs' k)
Staging

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

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)
Modules
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
Structures as concrete implementations of abstractions

```plaintext
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
```
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
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.
fun restoreLeft (Black(\_d1, x, \_d2), y, \_d3), z, \_d4)) =
    Red(Black(d1, x, d2), y, Black(d3, z, d4))
  || restoreLeft (Black(\_d1, x, Red(d2, y, d3)), z, \_d4)) =
    Red(Black(d1, x, d2), y, Black(d3, z, d4))
  || restoreLeft dict = dict
Representation Invariants

```haskell
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
```
Type Classes

Describe a type equipped with a (not usually exhaustive) collection of operations
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
Allow code re-use by abstracting over types and values
Brent’s Theorem and Parallelism

An expression with work $W$ and span $S$ can be evaluated on a $p$-processor machine in time $O(\max(W/p, S))$. 
Cost Graphs

\[(1 + 2) \times (3 + 4)\]

\[W = 10\]
\[S = 4\]

Pebbling

<table>
<thead>
<tr>
<th>CPU</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td>d</td>
<td>3</td>
</tr>
<tr>
<td>h</td>
<td>4</td>
</tr>
<tr>
<td>i</td>
<td>4</td>
</tr>
<tr>
<td>e</td>
<td>5</td>
</tr>
<tr>
<td>j</td>
<td>5</td>
</tr>
<tr>
<td>f</td>
<td>6</td>
</tr>
</tbody>
</table>
Sequences

parallel-friendly ordered-collections

provide parallelism for mathematical transformations on bulk data

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

signature SEQUENCE =

sig
  type 'a seq
  val empty : unit -> 'a seq
  val singleton : 'a -> 'a seq
  val tabulate : (int -> 'a) -> int -> 'a seq
  val nth : 'a seq -> int -> 'a
  val length : 'a seq -> int
  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 toList : 'a seq -> 'a list

... assuming argument functions are O(1).
end

(* O(1) span: empty, singleton, tabulate, nth, length, map.
  O(log n) span: reduce, mapreduce, filter.
  O(n) span: toList, fromList. *)
Game Playing

signature PLAYER =
  sig
    structure Game : GAME
    val next_move : Game.state -> Game.move
  end

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

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) : GO = ...
Mutation

can lead to race conditions
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)

2
Effects are benign if the implementation looks functional to clients.

<table>
<thead>
<tr>
<th></th>
<th>persistent</th>
<th>ephemeral</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel</td>
<td>FP</td>
<td>concurrency</td>
</tr>
<tr>
<td>sequential</td>
<td>benign effects</td>
<td>OK</td>
</tr>
</tbody>
</table>
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
signature STREAM =
sig
  type 'a stream
  datatype 'a front = Empty | Cons of 'a * 'a stream

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

signature STREAM =
sig
type 'a stream
datatype 'a front = Empty | Cons of 'a * 'a stream
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)
...

signature STREAM =
sig
  type 'a stream
  datatype 'a front = Empty | Cons of 'a * 'a stream

  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)
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()...

Memoizing Stream
Parsing
context-free grammars as tools for automatically generating parser
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
Uncomputability
important to understand
one’s limit
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)
We did not tell you that ...

Complexity of type inference is doubly exponential in the worst-case
Try the following for your amusement:

```
fun pair x y = fn z => z x y
val f1 = fn y => pair y y
val f2 = fn y => f1 (f1 y)
val f3 = fn y => f2 (f2 y)
val f4 = fn y => f3 (f3 y)
```
Functional Programming in Practice

• Theorem provers, hardware/software verification

• Companies in finance and telecommunications

• Compilers for most functional languages are implemented in themselves.
You might also like

• 15-210: Parallel Data Structures and Algorithms
• 15-312: Principles of Programming Languages
• 15-317: Constructive Logic
• 15-411: Compiler Design
• 15-451: Algorithms
• 15-453: Formal Languages, Automata and Computability
• 80-413: Category Theory
Two Sources of Beauty In Programs

• **Structure:** code as an expression of an idea

• **Efficiency:** code as instructions for a computer
Thanks to

Mike Erdmann
Jacob Neumann
Pankaj Bhojvani Ariel Davis Joy Gu Malaika Handa Leo Huang
Maxwell Johnson Thejas Kadur Minji Lee Adam Lerner Jeremy Leung
Wan Shen Lim Crystal Lin Tristan Marino Shashank Ojha
Pavi Pandurangan Cliff Ressel Kylee Santos Brian Scheuermann
Travis Schwartz Mera Tegene Elliot Toy Michael Wagner
Jordan Widjaja Carter Williams Cameron Wong Grace Yu Erin Zhang