Verification and Parallelism in Intro CS

Dan Licata
Wesleyan University

Starting in 2011, Carnegie Mellon revised its intro CS curriculum

- Computational thinking [Wing]
- Specification and verification
- Parallelism

CMU Intro Courses

Fundamentals of Computing

Imperative Computation

Functional Computation

Computer Systems

Data Structures and Algorithms

Software Construction

CMU Intro Courses

Fundamentals of Computing

Imperative Computation

Functional Computation

Computer Systems

Data Structures and Algorithms

Software Construction

Course Design

- **Imperative computation (Fa'11):**
  Frank Pfenning, Tom Cortina, William Lovas
- **Functional computation (Sp'11):**
  me, Bob Harper
- **Data structures and algorithms (Fa'12):**
  Guy Blelloch, Margaret Reid-Miller,
  Kanat Tangwongsan

taught and refined by many people since!

Status

- First group of students just graduated
- Courses are generally well-liked
- Anecdotally, students seemed stronger than before in some upper-level classes
- Preliminary studies by Carol Frieze indicate they preserve the women-CS fit at CMU

My role

- Designed Functional Computation
- Taught Spring’11, Fall’11, Spring’12
- Now at Wesleyan
- Taught Imperative Computation this spring; teaching Imperative next fall and Functional next spring

CMU

- Premier tech school
- Students admitted to CS
- ~600 students per year in intro
- ~150 CS majors per year

Wesleyan

- Liberal arts school, strong in sciences
- Not
- ~100 students per year in intro
- ~30 CS majors per year

CMU

- Imperative has a “basic programming” prereq
- Imperative has a math co-req
- Functional has a math prereq

Wesleyan

- Many students have never programmed before
- No math pre/co-reqs, though many students have it

Hypothesis

CMU courses can work elsewhere,
with some adaptation to context

Wesleyan this spring: 2/3rds of CMU Imperative

- less programming background
- shorter semester
- fewer student hours per week

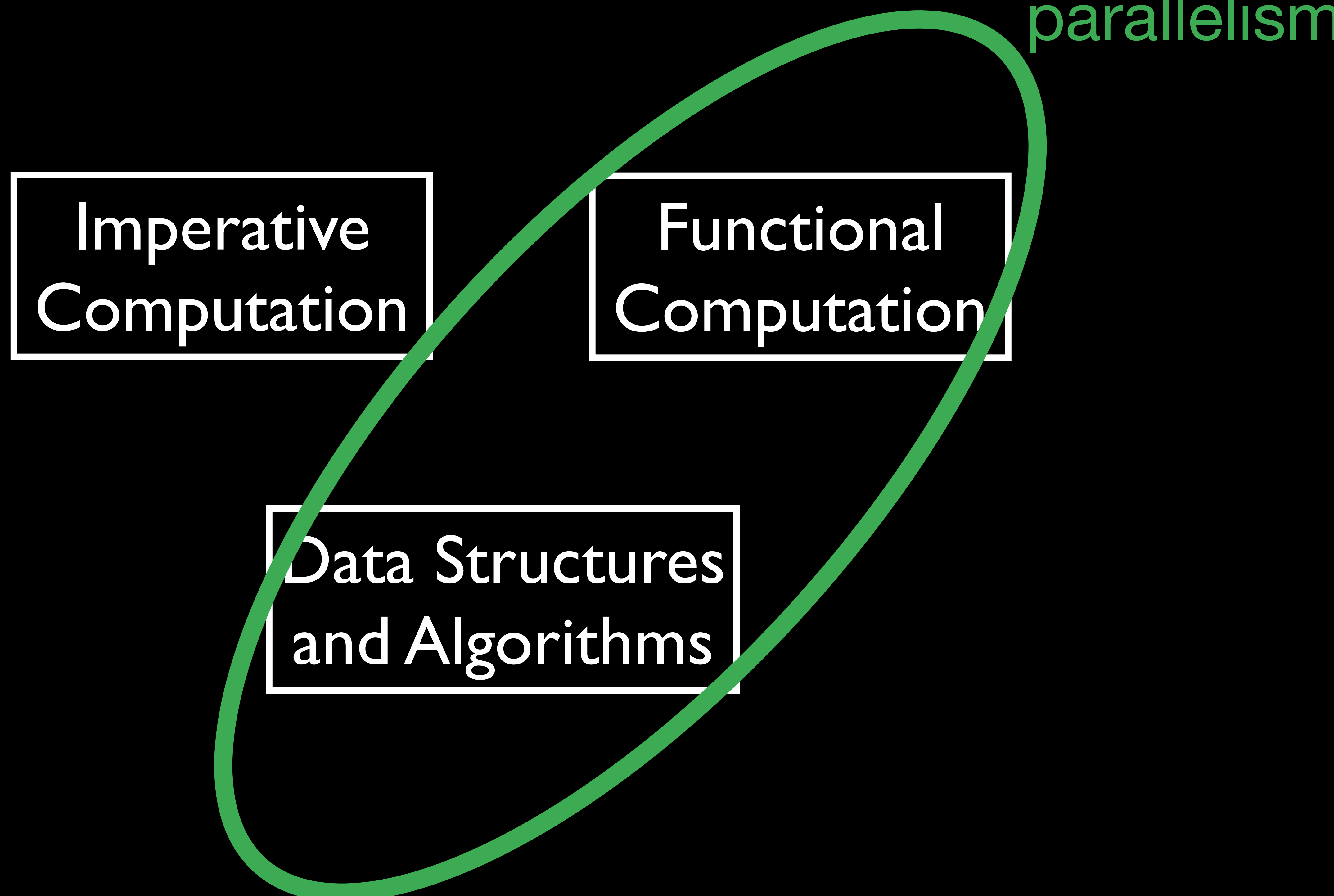
Course Design

Imperative Computation

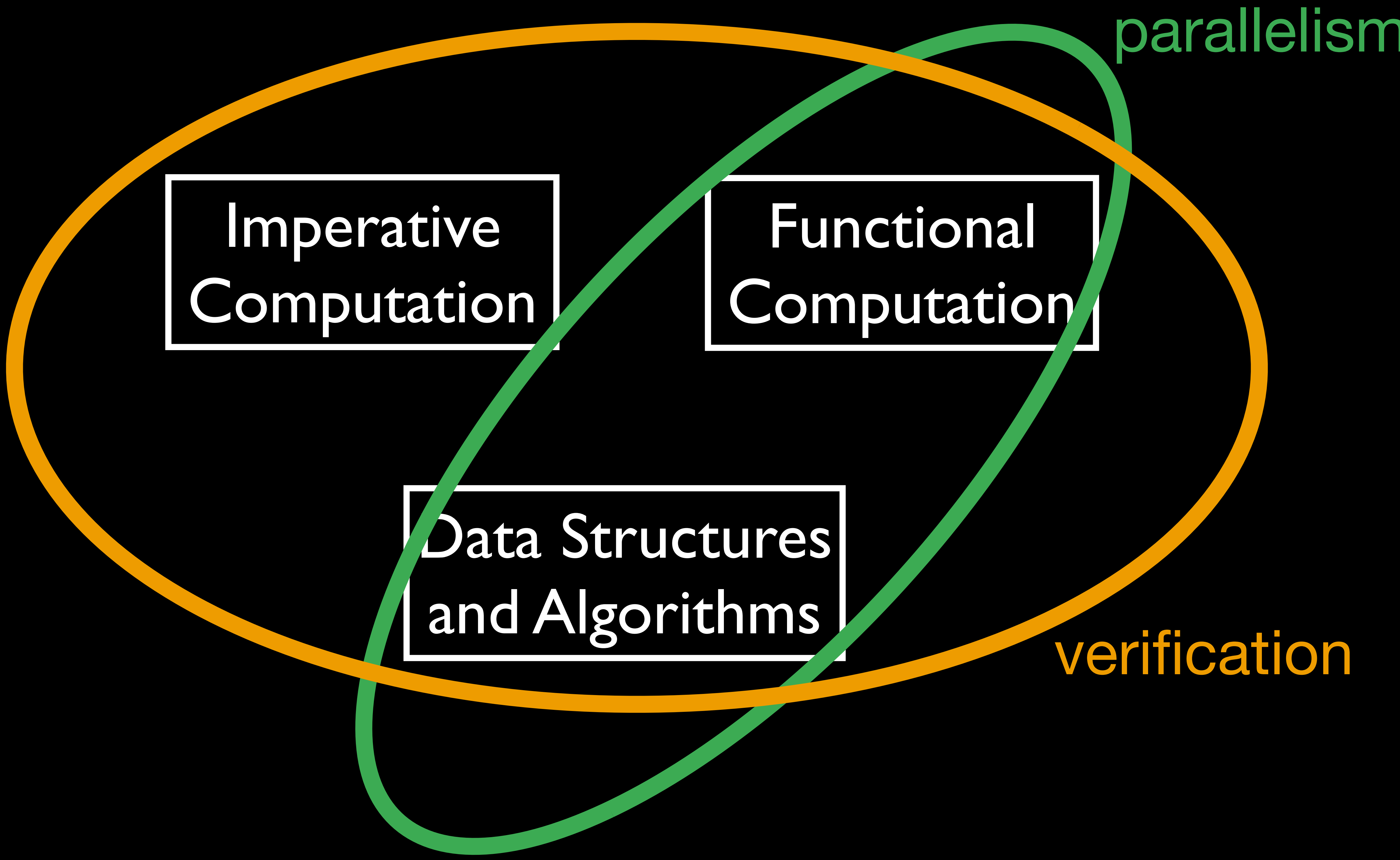
Functional Computation

Data Structures and Algorithms

Course Design



Course Design



Trade-offs

- Breadth vs depth
- Outward vs inward
- Motivation vs skills
- Short- vs long-term
- Systems vs theory
- Jump in vs training wheels

Trade-offs

- Breadth vs depth
- Outward vs inward
- Motivation vs skills
- Short- vs long-term
- Systems vs theory
- Jump in vs training wheels

Objectives

computational thinking

imperative and functional programming

algorithm design and analysis

specification and verification

Computational Thinking

what vs how

correctness and safety

efficiency

parallelism

randomness

self-reference

modularity

functions as data

ephemerality vs persistence

Objectives

computational thinking

imperative and functional programming

algorithm design and analysis

specification and verification

Imperative

C0: teaching language based on C, designed by Frank Pfenning and Rob Arnold

functions, variables, loops, arrays, pointers&structs

rudimentary interfaces

type safe, bounds checked, garbage collected

Imperative

Then transition to C

memory management
void* and casting
function pointers

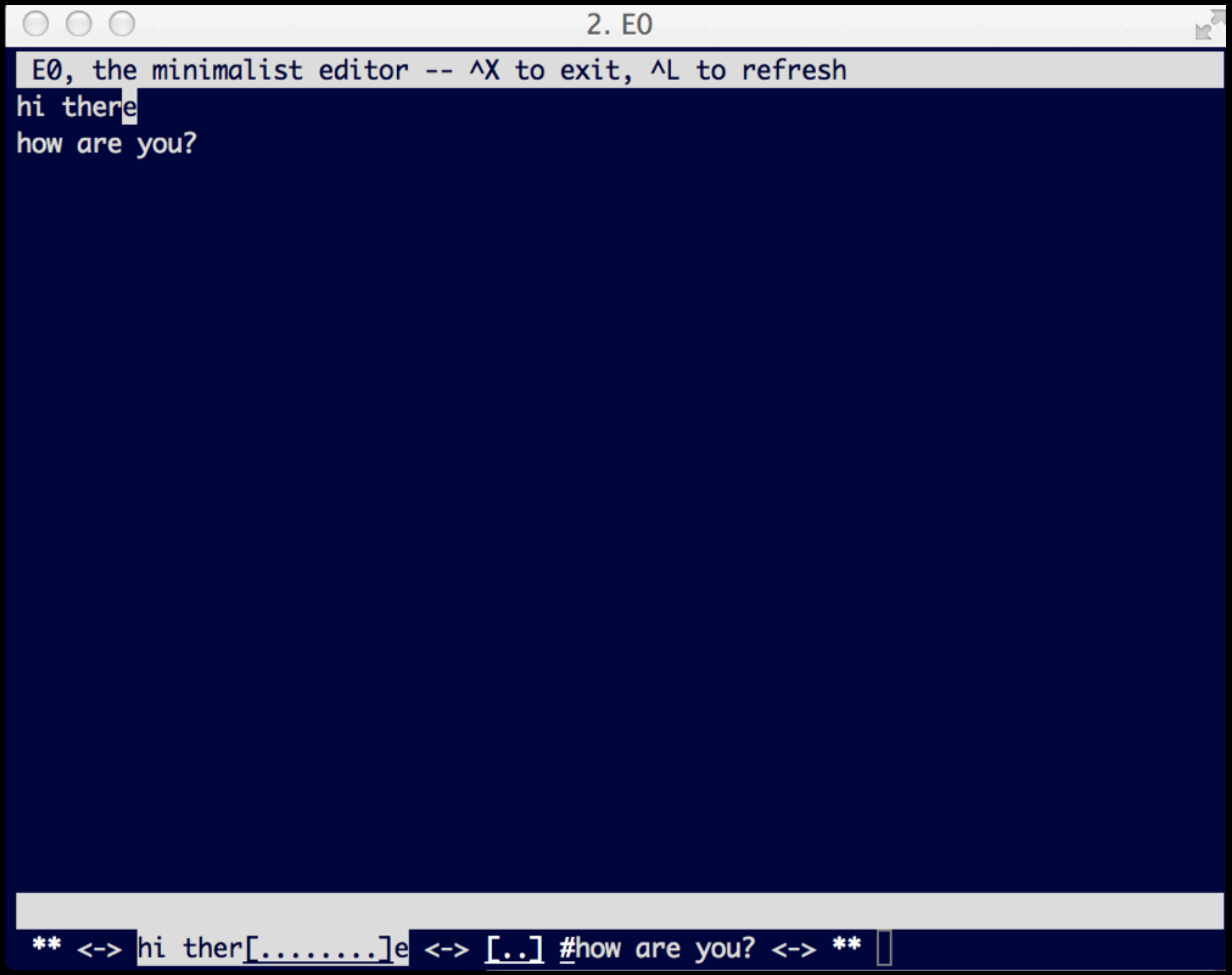
Imperative

Arrays

Search

```
Top 10 most frequent words in texts/twitter_200k.txt
i 83670
to 37445
the 36925
lol 35615
a 30300
u 28033
my 25093
you 24574
me 21326
it 21144
```

Pointers and structs



Data structures

COMP 211 Assignment 8 **Page 11 of 13**

2.7 Summary of data structures

There are a lot of different data structures being used in this homework! Here is a summary to help you keep them straight:

- Building a Huffman tree converts a dictionary (represented as a pair of arrays) to a Huffman tree, using a priority queue (whose elements are Huffman trees).
- To encode, we convert a Huffman tree to a hashtable (mapping characters to bitstrings) using a stack (of pairs of Huffman trees and bitstrings).
- To decode, we just use the Huffman tree.
- The hashtable elements are a struct `char_with_string` that pairs a character with a bitstring.
- The priority queue elements are Huffman tree trees.
- The stack elements are a struct `hufftree_with_path` that pairs a Huffman tree with a bitstring.
- Decode returns a `decode_result`, which is either `NULL` or a pair of a character and a bitstring.

Functional

Standard ML

numbers, pairs, lists, trees, datatypes

functions as arguments and results

signatures, structures, functors

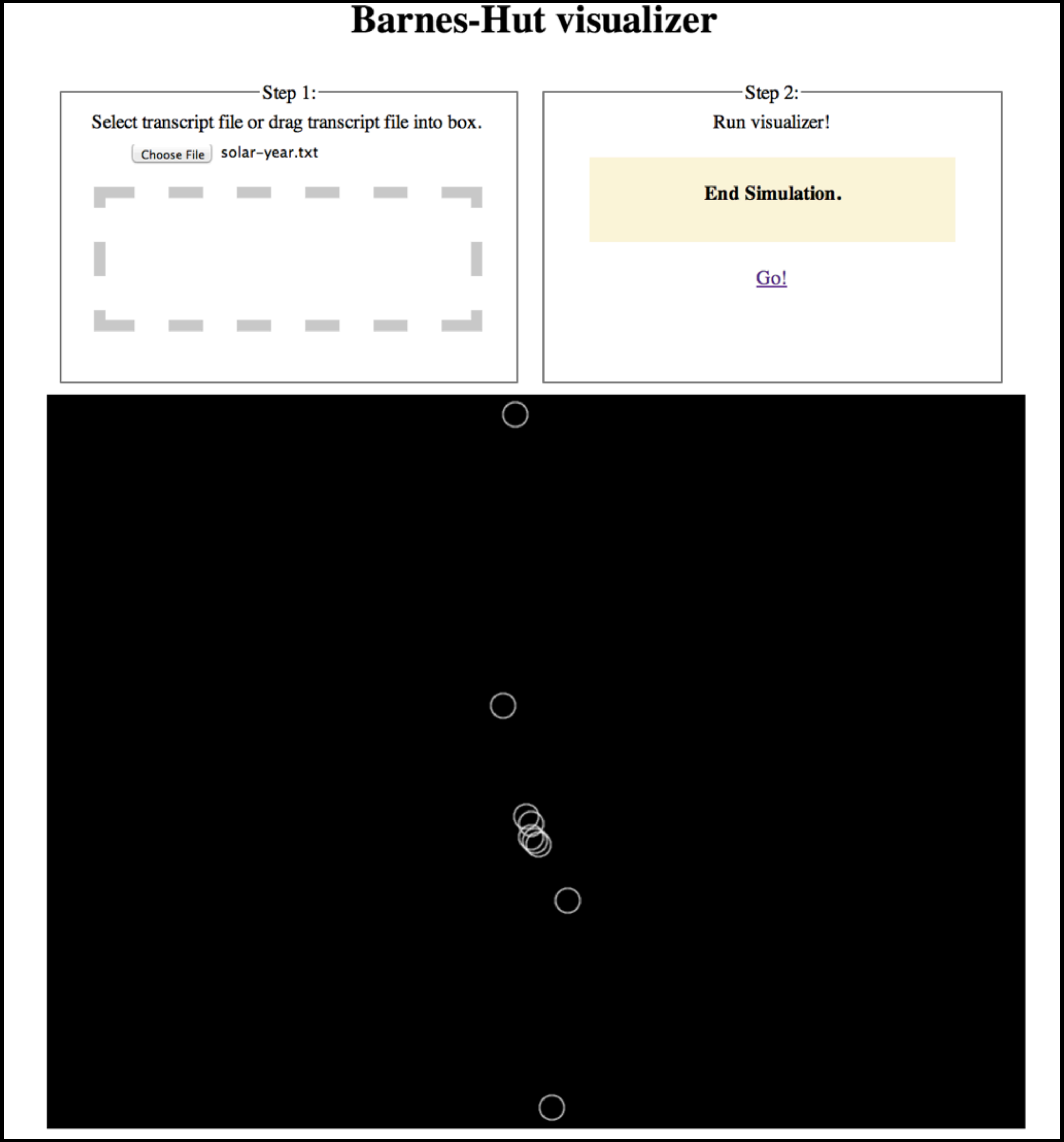
exceptions, mutation, IO

Functional

Why not

- Ocaml: no parallel implementation (Manticore for SML)
- Haskell: laziness complicates cost analysis
- F#: want the ML module system

Functional


Barnes-Hut visualizer
Step 1:
Select transcript file or drag transcript file into box.
Choose File
solar-year.txt
Step 2:
Run visualizer!
End Simulation.
Go!

Functional

```
Maxie, please type your move: 0
Maxie decides to make the move 0 in 14 seconds.
 0 1 2 3 4 5 6
---------------
| | | | | | | |
| | | | | | | |
| | | | | | | |
|X| | | | | | |
|X| | | | | | |
|X| | |O|O| | |


Minnie decides to make the move 0 in 0 seconds.
 0 1 2 3 4 5 6
---------------
| | | | | | | |
| | | | | | | |
|O| | | | | | |
|X| | | | | | |
|X| | | | | | |
|X| | |O|O| | |
```

Objectives

computational thinking

imperative and functional programming

algorithm design and analysis

specification and verification

Imperative

- ephemeral data structures
- arrays
- searching
- in-place sorting
- linked lists
- stacks
- queues
- unbounded arrays
- priority queues
- hash tables
- DFS/BFS
- balanced BSTs
- tries
- spanning trees
- union find

Imperative

ephemeral data structures

arrays

searching

in-place sorting

linked lists

stacks

queues

unbounded arrays

priority queues

hash tables

DFS/BFS

Analysis

big-O

worst case

amortized

expected case

Functional

persistent data structures

lists

trees

sorting

regular expression matching

n-body simulation

balanced BSTs

game tree search

Analysis

recurrence relations

closed forms

log

Data structures and algorithms

divide and conquer
sequences
sets and tables
randomization
dynamic programming

BFS/DFS
shortest paths
treaps
leftist heaps
k-grams

Functional

lists
trees
sorting
n-body simulation
game tree search

DS&A

divide and conquer
sequences
sets and tables
randomization
BFS/DFS
shortest paths
treaps
dynamic programming
leftist heaps

Functional

lists
trees
sorting
n-body simulation
game tree search

DS&A

divide and conquer
sequences
sets and tables
randomization
BFS/DFS
shortest paths
treaps
dynamic programming
leftist heaps

always have parallelism in mind!

Imperative

mutable data structures
arrays
searching
in-place sorting
linked lists
stacks
queues
unbounded arrays
priority queues
hash tables
DFS/BFS
tries
spanning trees
union find

Functional/ DS&A

lists
trees
sorting
n-body simulation
game tree search
divide and conquer
sequences
sets and tables
randomization
BFS/DFS
shortest paths
treaps
dynamic programming
leftist heaps

Imperative	Functional/ DS&A
mutable data structures	lists
arrays	trees
searching	sorting
in-place sorting	n-body simulation
linked lists	game tree search
stacks	divide and conquer
queues	sequences
unbounded arrays	sets and tables
priority queues	randomization
hash tables	BFS/DFS
DFS/BFS	shortest paths
tries	treaps
spanning trees	dynamic programming
union find	leftist heaps

always have verification in mind!

Objectives

computational thinking

imperative and functional programming

parallel algorithm design and analysis

specification and verification

Specification and Verification

Specification and Verification

Imperative: pre/post-conditions and loop invariants, expressed using boolean-valued functions

Functional: mathematical statements about calculational behavior of programs

Slogans

proof-oriented programming

deliberate programming

proof-directed debugging

Imperative

Specifications

safety

behavioral

data structure invariants

Safety

```
// try to prove that all array accesses are in bounds

void process_heartbeat(struct ssl* request)
{

  int payload = request->data[0];

  int[] buffer = alloc_array(int, 1 + payload);

  buffer[0] = payload;

  for (int i = 0; i < payload; i = i + 1)
  {
    buffer[i+1] = request->data[i+1];
  }

  send(buffer,1+payload);

}
```

Behavior

```
void sort(int[] A, int lower, int upper)
//@ requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ ensures is_sorted(A,lower,upper);
{
  for (int i = lower; i < upper ; i = i + 1)
    //@loop_invariant lower <= i && i <= upper;
    //@loop_invariant is_sorted(A,lower,i);
    //@loop_invariant le_segs(A,lower,i,i,upper);
    {
      int smallest_index = get_min(A,i,upper);
      swap(A,i,smallest_index);
    }
}
```

Behavior

```
//return true iff A[lower,upper) is sorted
bool is_sorted(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
{
  for (int i = lower; i + 1 < upper; i = i + 1)
    //@loop_invariant 0 <= i;
    {
      if (A[i] > A[i+1]) { return false; }
    }
  return true;
}
```

Proofs

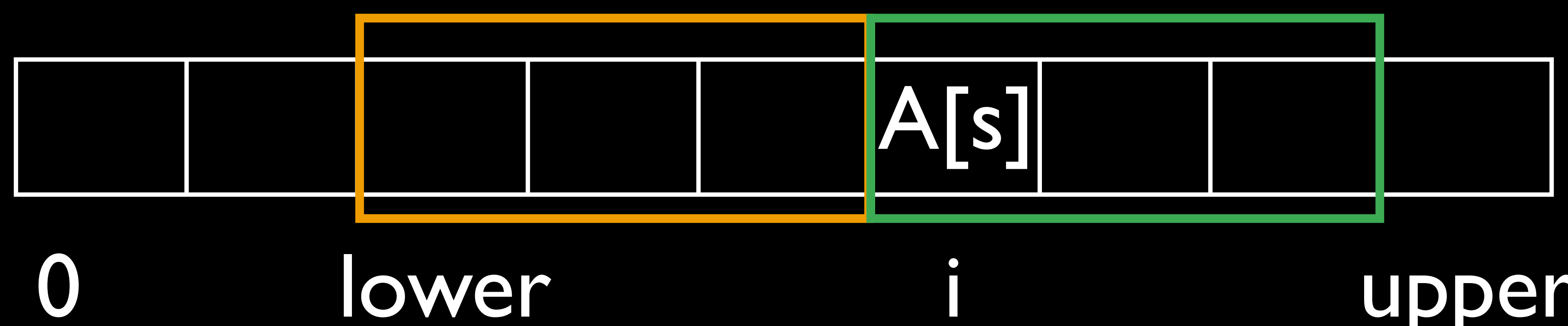
Loop invariants hold initially

Loop invariants preserved by one iteration

Loop invariants imply the postcondition

```
void sort(int[] A, int lower, int upper)
//@ requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ ensures is_sorted(A,lower,upper);
{
  for (int i = lower; i < upper ; i = i + 1)
    //@loop_invariant lower <= i && i <= upper;
    //@loop_invariant is_sorted(A,lower,i);
    //@loop_invariant le_segs(A,lower,i,i,upper);
    {
      int smallest_index = get_min(A,i,upper);
      swap(A,i,smallest_index);
    }
}
```

Preservation of LI



A[lower,i) sorted

A[lower,i) <= A[i,upper)

A[s] <= A[i+1,upper)

A[lower,i+1) sorted

A[lower,i+1) <=
A[i+1,upper)

Binary Search

```
int search(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result == -1 && !is_in(x, A, 0, n))
        || (0 <= \result && \result < n && A[\result] == x); @*/
{
  int lower = 0;
  int upper = n;

  // look in A[lower,upper)

  while (lower < upper)
    //@ loop_invariant 0 <= lower && lower <= upper && upper <= n;
    //@ loop_invariant lower == 0 || (x > A[lower - 1]);
    //@ loop_invariant upper == n || x < A[upper];
    {
      int mid = lower + (upper - lower) / 2;
      //@ assert lower <= mid && mid < upper;
      if (A[mid] == x) { return mid; }
      else if (A[mid] > x) {
        upper = mid;
      }
      else {
        //@ assert A[mid] < x;
        lower = mid+1;
      }
    }

  // @ assert lower == upper;

  return -1;
}
```

Binary Search

```
int search(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result == -1 && !is_in(x, A, 0, n))
        || (0 <= \result && \result < n && A[\result] == x); @*/
{
  int lower = 0;
  int upper = n;

  // look in A[lower,upper)

  while (lower < upper)
    //@ loop_invariant 0 <= lower && lower <= upper && upper <= n;
    //@ loop_invariant lower == 0 || (x > A[lower - 1]);
    //@ loop_invariant upper == n || x < A[upper];
    {
      int mid = lower + (upper - lower) / 2;
      //@ assert lower <= mid && mid < upper;
      if (A[mid] == x) { return mid; }
      else if (A[mid] > x) {
        upper = mid;
      }
      else {
        //@ assert A[mid] < x;
        lower = mid+1;
      }
    }

  // @ assert lower == upper;

  return -1;
}
```

Assignment: binary search for *first* occurrence

Data structure invs

```
struct heap_header {
  int limit;        /* limit = capacity+1 */
  int next;         /* 1 <= next && next <= limit */
  elem[] data;      /* \length(data) == limit */
};
typedef struct heap_header* heap;

/* Just checks the basic invariants described above, none
 * of the ordering invariants. */
bool is_safe_heap(struct heap_header* H) {
  if (H == NULL) return false;
  if (!(1 <= H->next && H->next <= H->limit)) return false;
  //@assert \length(H->data) == H->limit;
  return true;
}
```

Data structure invs

```
bool is_heap(struct heap_header* H)
{

  if (!is_safe_heap(H)) return false;

  for (int i = 2; i < H->next; i = i + 1)
    //@loop_invariant 2 <= i;
    {
      if (!(priority(H,i) >= priority(H,parent(i)))) return false;
    }

  return true;

}
```

Representation invs

```
/*** Client interface ***/

// typedef ______________ pq_elem;

int pq_elem_priority(pq_elem e);

/*** Library interface ***/

// typedef _______________ pq;
typedef struct heap_header* pq;

bool pq_empty(pq P);

pq pq_new(int capacity)
/*@ensures pq_empty(\result); @*/ ;

void pq_insert(pq P, pq_elem e)

pq_elem pq_delmin(pq P)
/*@requires !pq_empty(P); @*/ ;
```

Preservation

```
void pq_insert(heap H, elem e)
//@requires is_heap(H);
//@ensures is_heap(H);
{

  H->data[H->next] = e;
  int except = H->next;
  H->next = H->next + 1;

  // not necessarily is_heap(H)
  //@assert is_heap_except_up(H,except);

  // sift up

  while (except != 1 &&
         priority(H,except) < priority(H,parent(except)))
    //@ loop_invariant is_heap_except_up(H,except);
    {
      swap(H->data,except,parent(except));
      except = parent(except);
    }

  // @assert is_heap(H);

}
```

Data structure invs

```
bool is_heap_except_up(heap H, int except)
// ensures is_heap_except_up(H,1) == is_heap(H)
{
  if (!is_safe_heap(H)) {return false;}

  for (int n = 2; n < H->next ; n = n + 1)
    //@ loop_invariant (2 <= n);
    {

      if (n != except) {
        if (!(priority(H,n) >= priority(H,parent(n)))) {
          return false;
        }
      }

      if ((n == left(except) || n == right(except))
          && exists(parent(except),H->next)) {
        if (!(priority(H,n) >= priority(H,parent(except)))) {
          return false;
        }
      }

    }
  return true;
}
```

Heaps

(f) Draw a picture of what happens when the swap on line `/*2*/` is performed. Explain why the loop invariant is preserved in this case.

Solution:

Data structure invs

Task 1 (6 pts) *A valid text buffer satisfies all the invariants described above: it is a valid doubly-linked list containing valid size-16 gap buffers, it is aligned, and it consists of either one empty gap buffer or one or more non-empty gap buffers. Implement the function*

```
bool is_tbuf(tbuf B)
```

that formalizes the text buffer data structure invariants.

Contracts

```
'w': START <--> abc12345[........] <--> _678#w[..]WXYZdefgh_ <--> [........]ABCDEFGH <--> END
'x': START <--> abc12345[........] <--> _678#wx[.]WXYZdefgh_ <--> [........]ABCDEFGH <--> END
'y': START <--> abc12345[........] <--> _678#wxy□WXYZdefgh_ <--> [........]ABCDEFGH <--> END
'z': START <--> abc12345[........] <--> _678#wxyz[.......]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
'#': START <--> abc12345[........] <--> _678#wxyz#[......]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _678#wxyz[.......]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _678#wxy[........]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _678#wx[.........]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _678#w[..........]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _678#[...........]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _678[............]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _67[.............]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _6[..............]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _[...............]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> _abc1234[.........]_ <--> [...............]W <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
=> : START <--> abc1234[.........] <--> _W[...............]_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: tbuf.c0:197.4-197.23: @ensures annotation failed
```

```
void tbuf_delete(tbuf B)
//@requires is_tbuf(B);
//@ensures is_tbuf(B);
```

Functional

Specifications

Beyond requires/ensures

Specs relating multiple functions

Propositions, not booleans

Computing by calculation

```
    (1 + 2) * (3 + 4)
|-> 3 * (3 + 4)            (because 1 + 2 |-> 3)
|-> 3 * 7                  (because 3 + 4 |-> 7)
|-> 21
```

Contextual equivalence

```
(fn x => e) e' == [e'/x]e  if  e' valuable
```

```
e1 e2 == e1 e2'  if  e2 == e2'
e1 e2 == e1' e2  if  e1 == e1'
```

```
f == g : T2 -> T if
  for all values v : T2, f v == g v : T
```

Task 2.1 (5%). Write the function

```
zip : int list * string list -> (int * string) list
```

Task 2.2 (5%). Write the function

```
unzip : (int * string) list -> int list * string list
```

Task 2.3 (10%). Prove Theorem 1.

Theorem 1. *For all* `l : (int * string) list`, $\texttt{zip(unzip l)} \cong l$.

Task 2.4 (4%). Prove or disprove Theorem 2.

Theorem 2. *For all* `l1 : int list` *and* `l2 : string list`,

$$\texttt{unzip(zip (l1,l2))} \cong \texttt{(l1,l2)}$$

Props, not bools

$s \in L(0)$	iff	$\bot$
$s \in L(1)$	iff	$s = []$
$s \in L(c)$	iff	$s = [c]$
$s \in L(r_1 + r_2)$	iff	$s \in L(r_1) \lor s \in L(r_2)$
$s \in L(r_1 r_2)$	iff	$\exists s_1, s_2.(s = s_1 s_2) \land s_1 \in L(r_1) \land s_2 \in L(r_2)$

Props, not bools

$s \in L(0)$	iff	$\bot$
$s \in L(1)$	iff	$s = []$
$s \in L(c)$	iff	$s = [c]$
$s \in L(r_1 + r_2)$	iff	$s \in L(r_1) \vee s \in L(r_2)$
$s \in L(r_1 r_2)$	iff	$\exists s_1, s_2 . (s = s_1 s_2) \wedge s_1 \in L(r_1) \wedge s_2 \in L(r_2)$

Props, not bools

$s \in L(0)$	iff	$\bot$
$s \in L(1)$	iff	$s = []$
$s \in L(c)$	iff	$s = [c]$
$s \in L(r_1 + r_2)$	iff	$s \in L(r_1) \lor s \in L(r_2)$
$s \in L(r_1 r_2)$	iff	$\exists s_1, s_2. (s = s_1 s_2) \land s_1 \in L(r_1) \land s_2 \in L(r_2)$

want to reason logically about existential,
not about try-all-splits implementation

Props, not bools

$$s \in L(r^*) \quad \text{iff} \quad s \in L^*(r)$$

$$\frac{}{[] \in L^*(r)} \qquad \frac{s = s_1 s_2 \qquad s_1 \in L(r) \qquad s_2 \in L^*(r)}{s \in L^*(r)}$$

inner inductive definition of L*

Props, not bools

```
match : regexp -> char list -> (char list -> bool) -> bool
```

Soundness For all cs, k, if $\texttt{match}\ r\ cs\ k \cong \texttt{true}$
then there exist p, s such that $p@s \cong cs$ and $p \in L(r)$ and $k\ s \cong \texttt{true}$.

Completeness For all cs, k,
if (there exist p, s such that $p@s \cong cs$ and $p \in L(r)$ and $k\ s \cong \texttt{true}$)
then $\texttt{match}\ r\ cs\ k \cong \texttt{true}$.

Regexp

```
fun match (r : regexp) (cs : char list) (k : char list -> bool) : bool =
    case r of
        Zero => false
      | One => k cs
      | Char c => (case cs of
                         [] => false
                       | c' :: cs' => chareq (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
```

Regexp

```
fun match (r : regexp) (cs : char list) (k : char list -> bool) : bool =
    case r of
        Zero => false
      | One => k cs
      | Char c => (case cs of
                       []  => false
                     | c' :: cs' => chareq (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
```

Regexp

```
fun match (r : regexp) (cs : char list) (k : char list -> bool) : bool =
    case r of
        Zero => false
      | One => k cs
      | Char c => (case cs of
                         []  => false
                       | c' :: cs' => chareq (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
```

termination bug

Proof-directed debugging

$$s \in L(r_1 \cap r_2) \quad \text{iff} \quad s \in L(r_1) \wedge s \in L(r_2)$$

Modularity

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

signature DICT =
sig
  structure Key : ORDERED
  type 'v dict

  val empty  : 'v dict
  val insert : 'v dict -> (Key.t * 'v) -> 'v dict
  val lookup : 'v dict -> Key.t -> 'v option
end
```

Representation invs

```
A RBT satisfies
(red) no red node has a red child
(black) all paths from the root to a leaf have the same number of black nodes
```

```
(* If  d is a RBT  then  insert d (k,v)  is a RBT *)
fun insert d (k, v) = ...
```

Representation invs

```
signature QUEUE=
sig
   type queue
   val emp : queue
   val ins : int * queue -> queue
   val rem : queue -> (int * queue) option
end
```

$$\mathcal{R}(\texttt{l:int list}, \texttt{(f,b):int list * int list}) \quad \text{iff} \quad l \cong f@(\texttt{rev}\, b)$$

and $\mathcal{R}$ respects equivalence in that if $l \cong l'$, $(f, b) \cong (f', b')$, and $\mathcal{R}(\texttt{l}, \texttt{(f,b)})$ then $\mathcal{R}(\texttt{l'}, \texttt{(f',b')})$.

Showing that this relation is respected by both implementations for all the values in `QUEUE` amounts to proving the following theorem:

Theorem 1.

(i.) The empty queues are related:

$$\mathcal{R}(\mathit{LQ.emp}, \mathit{LLQ.emp})$$

(ii.) Insertion preserves relatedness:

For all `x:int`, `l:int list`, `f:int list`, `b:int list`

$$\text{If } \mathcal{R}(\texttt{l}, \texttt{(f,b)}), \text{ then } \mathcal{R}(\texttt{LQ.ins(x,l)}, \texttt{LLQ.ins(x,(f,b))})$$

(iii.) On related queues, removal gives equal integers and related queues:

For all `l:int list`, `f:int list`, `b:int list`, *if* $\mathcal{R}(\texttt{l}, \texttt{(f,b)})$ *then one of the following is true:*

(a) `LQ.rem l` $\cong$ `NONE` *and* `LLQ.rem (f,b)` $\cong$ `NONE`

(b) *There exist* `x:int`, `y:int`, `l':int list`, `f':int list`, `b':int list`, *such that*

i. `LQ.rem l` $\cong$ `SOME(x,l')`

ii. `LLQ.rem (f,b)` $\cong$ `SOME(y,(f',b'))`

iii. $x \cong y$

iv. $\mathcal{R}(\texttt{l'}, \texttt{(f',b')})$

Verification

safety

behavioral specifications

representation invariants + modularity

booleans and propositions

contracts

Slogans

proof-oriented programming

deliberate programming

proof-directed debugging

Parallelism

Parallelism

recognize the dependencies

deterministic parallelism

language-based cost model

asymptotic analysis

Parallelism != Concurrency

parallelism: multiple processors/cores.
property of the machine.

concurrency: interleaving of threads.
property of the application.

Parallelism != Concurrency

	sequential	concurrent
serial	traditional algorithms	traditional OS
parallel	**deterministic parallelism**	general parallelism

Parallel Calculation

```
    (1 + 2) * (3 + 4)
|-> 3 * (3 + 4)             (because 1 + 2 |-> 3)
|-> 3 * 7                   (because 3 + 4 |-> 7)
|-> 21
```

```
    (1 + 2) * (3 + 4)
|=> 3 * 7                   (because 1 + 2 |=> 3 AND 3 + 4 |=> 7)
|=> 21
```

Parallel Calculation

```
    (1 + 2) * (3 + 4)
|-> 3 * (3 + 4)             (because 1 + 2 |-> 3)
|-> 3 * 7                   (because 3 + 4 |-> 7)
|-> 21
```

```
    (1 + 2) * (3 + 4)
|=> 3 * 7                   (because 1 + 2 |=> 3 AND 3 + 4 |=> 7)
|=> 21
```

verification is the same as without parallelism

Work and span

work is usual serial time complexity

$$\text{work } \langle e_1, e_2 \rangle = \text{work}(e_1) + \text{work}(e_2)$$

span is parallel time complexity

$$\text{span } \langle e_1, e_2 \rangle = \max(\text{span}(e_1), \text{span}(e_2))$$

Mergesort

```
fun split (l : int list) : int list * int list =
    case l of
        [] => ([] , [])
      | [ x ] => ([ x ] , [])
      | x :: y :: xs => let val (pile1 , pile2) =
                                split xs
                        in (x :: pile1 , y :: pile2)
                        end

fun merge (l1 : int list , l2 : int list) : int list =
    case (l1 , l2) of
        ([] , l2) => l2
      | (l1 , []) => l1
      | (x :: xs , y :: ys) =>
            (case x < y of
                 true => x :: (merge (xs , l2))
               | false => y :: (merge (l1 , ys)))

fun mergesort (l : int list) : int list =
    case l of
        [] => []
      | [x] => [x]
      | _ => let val (pile1,pile2) = split l
             in
                 merge (mergesort pile1, mergesort pile2)
             end
```

Work

[7,1,3,6,8,4,2,5]

Work

[7,1,3,6,8,4,2,5]

[7,1,3,6] [8,4,2,5]

Work

[7,1,3,6,8,4,2,5]

[7,1,3,6] [8,4,2,5]

[7,1] [3,6] [8,4] [2,5]

Work

[7,1,3,6,8,4,2,5]

[7,1,3,6] [8,4,2,5]

[7,1] [3,6] [8,4] [2,5]

[7] [1] [3] [6] [8] [4] [2] [5]

Work

[7,1,3,6,8,4,2,5]

[7,1,3,6]　　　　[8,4,2,5]

[7,1]　　[3,6]　　　　[8,4]　　[2,5]

[7]　[1]　　[3]　[6]　　　　[8]　[4]　　[2]　[5]

$W(n) = n + 2\,W(n/2)$　　is $O(n \log n)$

Span

[7,1,3,6,8,4,2,5]

[7,1,3,6]　　[8,4,2,5]

[7,1]　[3,6]　[8,4]　[2,5]

[7]　[1]　[3]　[6]　[8]　[4]　[2]　[5]

Span

[7,1,3,6,8,4,2,5]

[7,1,3,6] [8,4,2,5]

[7,1] [3,6] [8,4] [2,5]

[7] [1] [3] [6] [8] [4] [2] [5]

Span

[7,1,3,6,8,4,2,5]

[7,1,3,6] [8,4,2,5]

[7,1] [3,6] [8,4] [2,5]

[7] [1] [3] [6] [8] [4] [2] [5]

$S(n) = n + S(n/2)$ is $O(n)$

Mergesort

on lists: $O(n \log n)$ work

$O(n)$ span

on trees: $O(n \log n)$ work

$O((\log n)^3)$ span

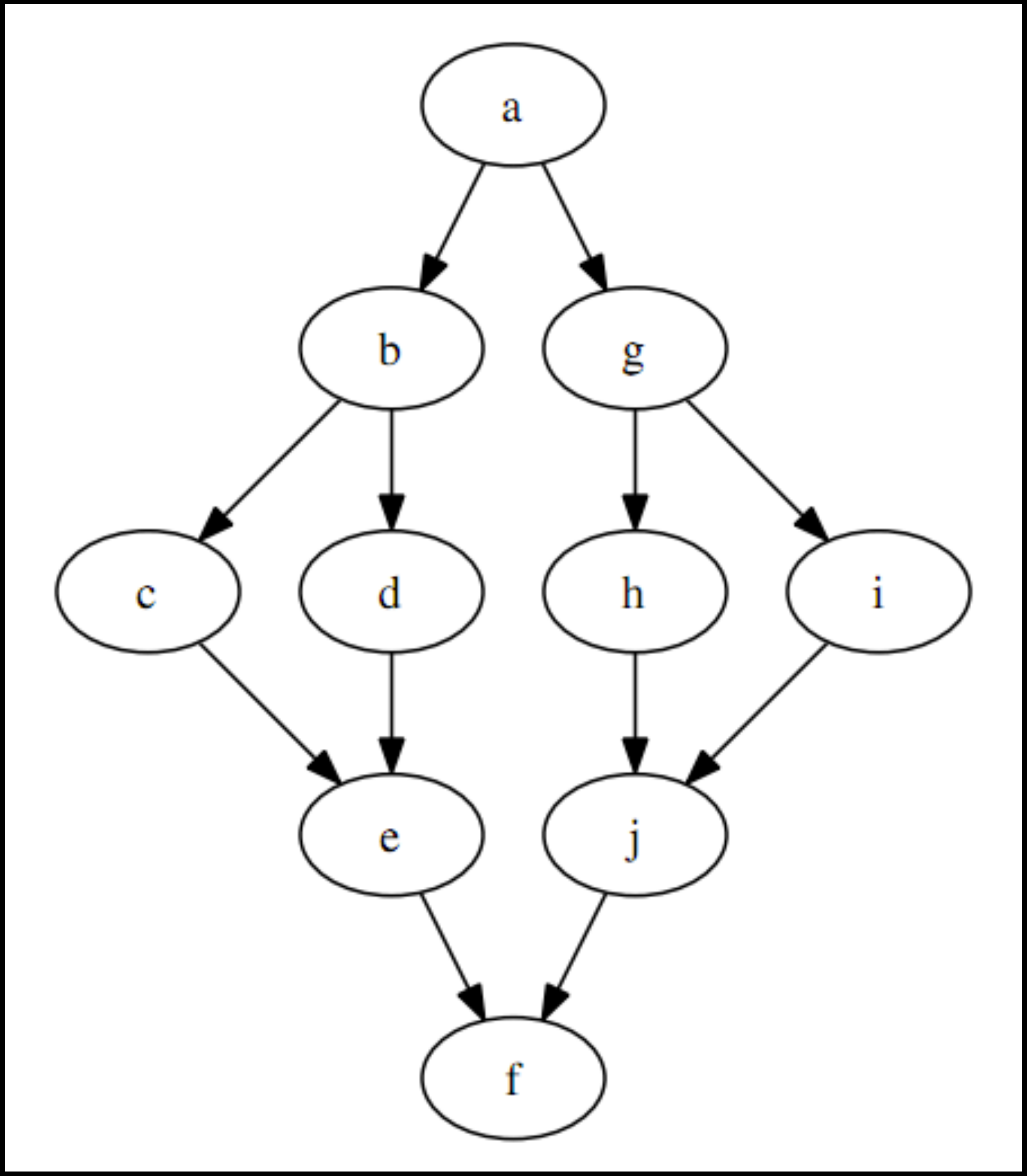
```
fun splitAt (t : tree , bound : int) : tree * tree =
 case t of
     Empty => (Empty , Empty)
   | Node (l , x , r) =>
      (case bound < x of
           true => let val (ll , lr) = splitAt (l , bound)
                   in (ll , Node (lr , x , r))
                   end
         | false => let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

Design principles

keep work low (ideally *work-efficient*)

then minimize span

Brent's principle: time to run on p processors is O(max(work/p,span))



Sequences

```
(* map f <x1, ..., xn> == <f x1, ..., f xn>
   work O(n)
   span O(1)
 *)
val map : ('a -> 'b) -> 'a seq -> 'b seq

(* reduce op b <x1, ..., xn> == x1 op x2 ... op xn
   work O(n)
   span (log n)
 *)
val reduce : (('a * 'a) -> 'a) -> 'a -> 'a seq -> 'a
```

n-body simulation

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

quadratic work

logarithmic span

Barnes-Hut

divide space recursively into quadrants, approximating contribution of bodies that are too far away by their center of mass

O(n log(n)) work

logarithmic span

Game tree search

Minimax: at each node,
work is O(children)
span is O(log children)

Alpha-beta pruning:
work is better
but span is linear

Jamboree: trade work for span

Data structures and algorithms

divide and conquer
sequences
sets and tables
randomization
dynamic programming

BFS/DFS
shortest paths
graph contractability
treaps
leftist heaps

Implementation

Nested parallelism can be realized in

Nesl

Manticore (SML)

Parallel Haskell

Cilk

TPL (C#/F#)

OpenMP

Parallelism

recognize the dependencies

deterministic parallelism

language-based cost model

asymptotic analysis

Objectives

computational thinking

imperative and functional programming

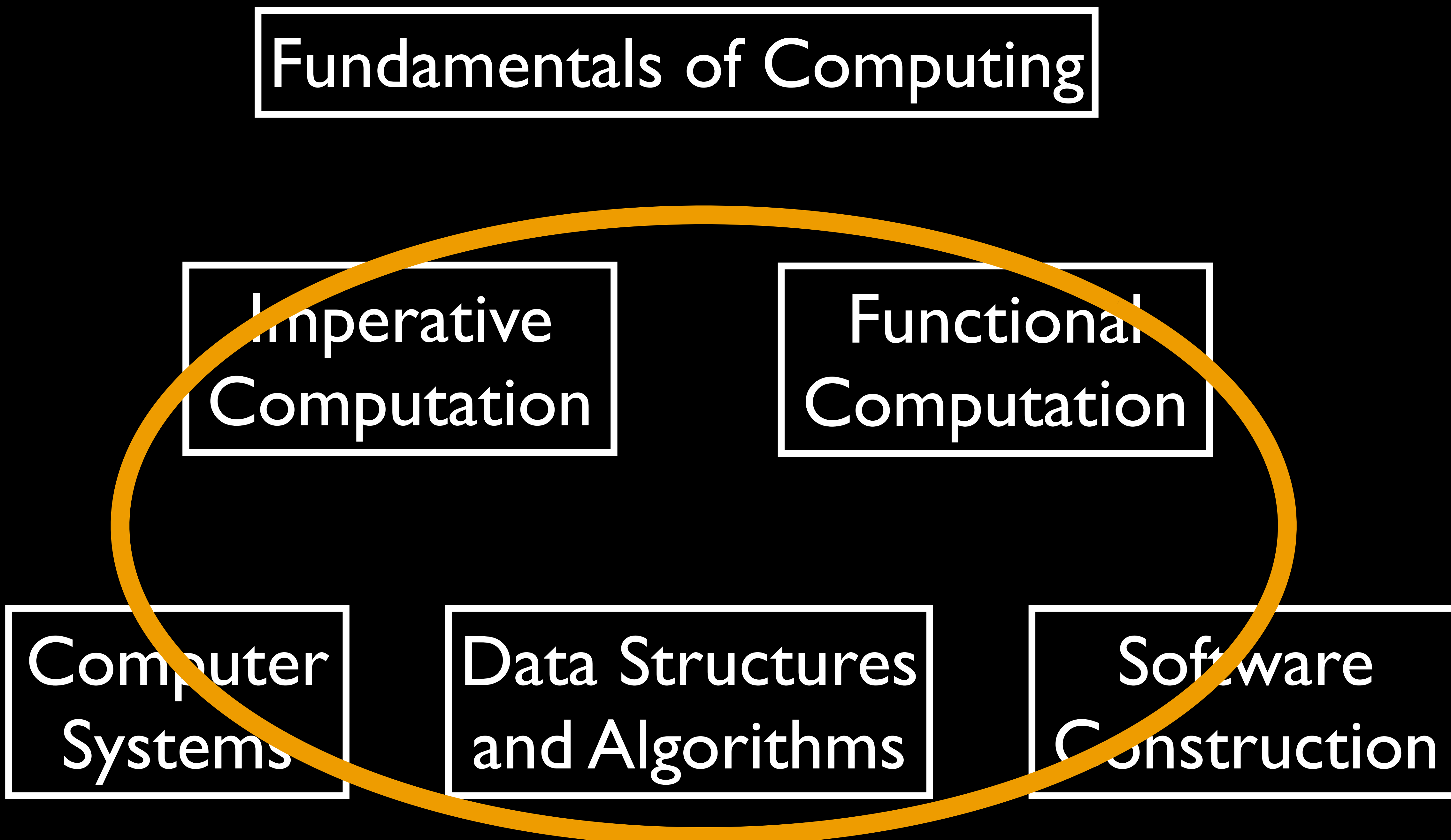
parallel algorithm design and analysis

specification and verification

Trade-offs

- Breadth vs depth
- Outward vs inward
- Motivation vs skills
- Short- vs long-term
- Systems vs theory
- Jump in vs training wheels

CMU Intro Courses



Activities

lecture

lab/recitation

homework

lots of TA homework help time

lots of TA grading time

Experiment

CMU courses can work elsewhere,
with some adaptation to context

Wesleyan this spring: 2/3 of CMU Imperative

Wesleyan next year: 3/4 of CMU Functional?