Announcements
"and"
more trees
Modules – a list-based Queue

(define f (lambda (x) (+ x 1))
  (f (f 3))
Classes Thanksgiving Week

• Yes, there’s class on Monday, Nov. 25.
• Yes, there’s class on Wed, Nov. 27.
• Yes, there might be a quiz on either day.

| 27 Wed Nov until 1 Sun Dec | Thanksgiving recess beginning Wednesday at noon |
“and” keyword: mutual references

Goofy idea: a natural number n is even if n-1 is odd; define evenP recursively:

```ocaml
let rec evenP:int => bool = fun
| 0 => true
| n => oddP(n-1);
```

Problem: oddP is undefined!
“and” keyword: mutual references

let rec evenP:int => bool = fun
| 0 => true
| n => oddP(n-1);

oddP:int => bool = fun
| 0 => false
| n => evenP(n-1);

Problem: oddP is *still* undefined when the def'n of evenP is done!
“and” keyword: mutual references

let rec evenP:int => bool = fun
| 0 => true
| n => oddP(n-1)
and
oddP:int => bool = fun
| 0 => false
| n => evenP(n-1);
“and” keyword: mutual references

"and" also can be used for mutually-referential types:

type f = Foo(g);
type g = Bar(f);

This type constructor's parameter, `g`, can't be found. Is it a typo?
type f = Foo(g)
and
g = Bar(f);
Trees again; tree recursion

```ocaml
type tree('a) = Leaf | Node('a, tree('a), tree('a));

let rec tContains17: tree(int) => bool = fun
| Leaf => false
| Node(v, left, right) => ???
```
Trees again; tree recursion

type tree('a) = Leaf | Node('a, tree('a), tree('a));

let rec tContains17: tree(int) => bool = fun
    | Leaf => false
    | Node(v, left, right) => (v == 17) ||
                              tContains17(left) ||
                              tContains17(right);
Trees again; tree recursion

type tree('a) = Leaf | Node('a, tree('a), tree('a));

let rec tContains17: tree(int) => bool = fun
| Leaf => false
| Node(v, left, right) => (v == 17) ||
    tContains17(left) ||
    tContains17(right);

Analysis: Let $T(n)$ be the max number of elementary op'ns in evaluating $\text{tContains17}$ on an tree of $n$ nodes. {edited from in-class presentation where I had "nodes and leaves"}
$T(0) = A$
$T(n) < B + ...
Trees again; tree recursion

type tree('a) = Leaf | Node('a, tree('a), tree('a));

let rec tContains17: tree(int) => bool = fun
  | Leaf => false
  | Node(v, left, right) => (v == 17) ||
    tContains17(left) ||
    tContains17(right);

Analysis: Let T(n) be the max number of elementary op'ns in evaluating tContains17 on an
tree of n nodes.
T(0) = A
T(n) < B + T(n-1) + T(n-1) [pretty pessimistic, easy-ish to work with]
Leads to T(n) in O(n -> 2^n).
Trees again; tree recursion

type tree('a) = Leaf | Node('a, tree('a), tree('a));

let rec tContains17: tree(int) => bool = fun
| Leaf => false
| Node(v, left, right) => (v == 17) ||
  tContains17(left) ||
  tContains17(right);

Analysis: Let T(n) be the max number of elementary op'ns in evaluating tContains17 on an
tree of n nodes.
T(0) = A
T(n) < B + max T(k) + T(n-k-1) [max over k = 1, 2, ..., n-2] [hard to work with].
We'll return to this and come up with something better ...
Quiz

• Write a function that replaces the data at every node of an int-tree with the number 17:

let rec tImprove: tree(int) => tree(int) = fun
| Leaf => ...
| Node(v, left, right) => ...;
Modules (final bit of ReasonML syntax)

• A module is like a tuple: it contains a bunch of disparate things
• Typically it contains several procedures.

Not so different from

```reasonml
type procs = {
    proc1: int => int,
    proc2: int => int,
    proc3: int => bool
};

let myProcs = {
    proc1: x => x+1,
    proc2: x => x + 2,
    proc3:x => true
};
```

• The procedures have names, and so does the module.
• Every ReasonML file you've written has *implicitly* been a module!
Creating a module

module Queue = {
  type queue = (list(int), list(int));
  let emptyQ = ([], []);
  let enq: (int, queue) => queue = (num, (front, back)) =>
    (front, [num, ... back]);
    ...
};

let s = Queue.emptyQ;
let t = Queue.enq(3, s);
Creating a module

module Queue = {
  type queue = (list(int), list(int));
  let empty = ([], []);
  let enq: (int, queue) => queue = (num, (front, back)) =>
    (front, [num, ... back]);
  ...
};

• This module contains functions (enq), values (empty), and types (queue). So it's a little fancier than what you can do with a tuple.
• To refer to things in the module, you prefix with the module-name
Why modules?

• Modules are a way to "package things together" in a cooperative way
• everything to do with "queues" gets put in one module.
• Everything to do with "lists" gets put in another
• Maybe both have items called "empty", but one is Queue.empty, the other is List.empty, so there's no name-collision!
Using modules

• We often want to use all the parts of the "Queue" module without constantly putting "Queue" in front of things.
• Solution:
  ```
  open Queue;
  ```
• Now we can refer to things in Queue without a prefix!
• You've seen this already:
  ```
  open CS17setup;
  ```
• The "module" here was implicit: the CS17setup.re file doesn't include the word "module" anywhere!
The contents of MyStuff.re

• The contents of file MyStuff.re are implicitly enclosed in module MyStuff {
  ...
};
• If you declared the Queue module inside MyStuff.re, the full name of the empty queue would be MyStuff.Queue.empty
Let's flesh out a list-based queue module

• A queue is like a line of people waiting to get on a bus
• A new person gets put at the back of the line
• The first person removed from the queue (to get on the bus) is at the FRONT of the line
• The queue could be empty.
• A lot like lists, except with lists, we add things at the front ("cons") and remove things from the front (first, rest).
Implementing a queue with lists

- Clever idea: use two lists, "front" and "back".
- Enqueue things on the "back" list
- Dequeue things from the "front" list
- Occasionally, as needed, move things from back to front.

Suppose we have \[ \text{front: [2, 3], back: [4, 7]} \]
- We enqueue "6". \[ \text{front: [2, 3], back: [6, 4, 7]} \]
- We dequeue an element...the first thing. \[ \text{front: [3], back [6, 4, 7]} \]
- We dequeue an element...the first thing. \[ \text{front: [], back [6, 4, 7]} \]
- We want to dequeue an element...it should be "7"...but the front list is empty!
Implementing a queue with lists

• We dequeue an element...the first thing. front: [], back [6, 4, 7]
• We want to dequeue an element...it should be "7"...but the front list is empty!
• When we want to dequeue and the front list is empty...
  • set front = reverse(back), back = [].
  • THEN dequeue
  • Fails if both are empty, though...so check that first!
module Queue = {
    type queue = (list(int), list(int));
    let emptyQ = ([], []);
    let enq: (int, queue) => queue = (num, (front, back)) =>
        (front, [num, ... back]);
    let rec deq: queue => queue = fun
        | ([], []) => failwith("Can't remove an item from an empty queue")
        | ([hd, ...tl], back) => (tl, back)
        | ([], back) => deq (List.rev(back), []);
    let first: queue => int = fun
        | ([], []) => failwith("no first item in an empty queue")
        | ([hd, ...], _) => hd
        | ([], back) => first (List.rev(back), []);
};
More about Queues

• They are an "abstract data type" (ADT), just like "list"
  • Support certain specified operations
  • Not required to do anything else
• Our implementation isn't the only possible one
  • More on this later!
• A full queue implementation in a few days
• You'll be implementing a "dictionary" ADT for HW
• Did a queue really have to contain just integers? Of course not.
Enhancing Queue to handle other data types

BROKEN code: you can't just name a type without saying what it is.

```plaintext
module Queue = {
  type myType;
  type queue = (list(myType), list(myType));
  let emptyQ = ([], []);
  ...
};
```

BROKEN code v 2: modules don't take type-parameters, alas

```plaintext
module Queue('myType) = {
  type queue = (list(myType), list(myType));
  let emptyQ = ([], []);
  ...
};
```
Module type Queue = {
  type myType; /* allowed inside a 'module type' */
  type queue = (list(myType), list(myType));
  let emptyQ : queue;
  ...
};

• Says that a "Queue" must have a type in it called "myType", but doesn't say what it has to be!
Modules have a "type" just like everything else

module type Queue = {
    type myType; /* allowed inside a 'module type' */
    type queue = (list(myType), list(myType));
    let emptyQ : queue;
    ...
};
module IntQueue:Queue = {
    type myType = int; /* myType now made concrete! */
    type queue = (list(myType), list(myType));
    let emptyQ = ([], []); /* said what the value really IS */
    ...
};
Module type subtleties

• There are peculiar things about module types that we'll encounter later

• Essential for the idea of hiding implementation details from users of your module
  • ...which lets you *improve* the implementation without breaking their programs!