Games, etc

Warmup: argmin
Recursion types
Games
ReasonML representation of games
Warmup

• Find largest square of an item in a nonempty int-list
• maxSquare([1, 4, 3]);
  • 16

let rec maxSquare: list(int) => int = fun
| [] => failwith("empty")
| [a] => a*a
| [hd, ... tl] => max (hd * hd, maxSquare(tl));
Warmup 2

• Find largest $f(n)$ (where $f: \text{int} \to \text{int}$) of an item in a nonempty int-list
• $\text{maxFunc}([1, -4, 3], x \Rightarrow x*x)$;
  
  let rec $\text{maxFunc}$: (list(int), int => int)=>int = (aloi, f) =>
  switch(aloi) {
  | [] => failwith("empty")
  | [a] => f(a)
  | [hd, ... tl] => max (f(hd), $\text{maxFunc}$(tl, f))
  };
• Also good:
  let $\text{maxFunc}$ = (aloi, func) => max (List.map(func, aloi));
Warmup 3

• Find the integer in a nonempty list where f takes its max

let rec argmax: (list(int), int => int) => int = (aloi, f) =>
  switch (aloi) {
  | [] => failwith("No argmax for empty list")
  | [a] => a
  | [hd, ...tl] =>
    let x = argmax(tl, f);
    if (f(hd) > f(x)) {
      hd;
    } else {
      x;
    };
  }
How would you write \text{argmin}?

\begin{verbatim}
let rec argmin: (list(int), int => int) => int =
argmax(aloi, n => -func(n));
\end{verbatim}
Warmup 3

• Find the location in a nonempty list where f takes its max
• \texttt{iargmax([1, 4, -5, 2], x => x*x)} is 3, because -5 is the third item.
• Approach: "strengthen the recursion!"
• Compute the location and the value associated to that location
  • See whether the value here is larger than the prior best

\begin{verbatim}
let rec iaHelper:(list(int), int => int) => (int, int) = (aloi, f) =>
  switch (aloi) {
    | [] => failwith("Can't handle empty lists in iaHelper")
    | [a] => (f(a), 1)
    | [hd, ...tl] =>
      let (v, i) = iaHelper(tl, f);
      if (f(hd) > v) {
        (f(hd), 1);
      } else {
        (v, i + 1);
      };
  };
\end{verbatim}
To finish up

...just extract the location from the (locn, value) pair!

```haskell
let rec iargmax:(list(int), int=>int) => int = (aloi, f) => switch(iaHelper(aloi,f)) {
| (_, i) => i
};
```
Non-quiz

• If you are planning to take CS18 next semester, but did not register for it, please raise your hand.
  • Kathi needs accurate headcount to get proper TA staff
Retrospective (and clue for next week's HW)

• Recursion on lists
  • Do something special with empty list
  • Do something with (first alod); cons that onto (myfunc (rest data))
  • Uses *structure* of lists
    • A list is either *empty* or
    • (cons item z), where z is a list

• Recursion on natural numbers (like "factorial")
  • Do something special with 0
  • Do something with n; combine that with (myfunc (pred n))
  • Uses *structure* of natural numbers
    • A natnum is either 0 or
    • (succ k) where k is a natnum
Retrospective (part 2)

• Recursion on trees
  • Do something special with *Leaf*
  • For *Node*, Do something with the value at a node; combine that with myFunc(leftChild), myFunc(rightChild)
  • Uses *structure* of trees
    • A tree is either *a Leaf* or
    • Node(val, leftChild, rightChild), where both children are trees.

• All three of these approaches are called *structural recursion*
Retrospective (part 3)

- **Mergesort** is different
  - Instead of working on the head and tail and combining
    - It splits the list in half
    - Not a "basic" operation on list structures (i.e., not "first" or "rest")
  - This is a "non-structural" recursion
- You'll be doing one of these on homework
  - Computed greatest common divisor of two numbers n, k
  - Computation does *not* involve n-1 or k-1
  - One of the earliest known algorithms
    - Invented long before the word "algorithm"
Time to play a game!

• Yucky Chocolate
• Break "slabs" off a Hershey bar
  • Either remove one or more ROWS
  • or one or more COLUMNS
• Bottom left square is bad
  • Made of soap instead of chocolate
  • Last player has to eat the yucky chocolate
Game characteristics

• Two-player
• Finite
• Sequential
• Alternating turns
• Zero-sum (I win when you lose, and vice-versa)
• Complete information
• Deterministic (no dice, shuffling, etc.)

• A "move" can be represented by an integer
Goal

• Write a program that lets us "play" a game
• We'll need something to represent a player
  • A player must be able, given the current state of the game, to pick a next move from among the legal moves
• We'll need a way to represent the game itself
  • Starting "state"
  • Rules
    • Legal moves at any stage of the game
    • Determine if someone has won/lost?
• We'll have a "referee" who starts the game, and then alternately asks each player to play.
Let's build some ReasonML code for representing Yucky Chocolate

- Soon we'll wrap this up in a Module.
- Then generalize to a *module type* for all possible games of the kind we're working on
type state = (int, int); /* # rows, cols left */
let initialState = (2, 2); /* Very simple game to start with! */
type move =
  | Row(int)
  | Col(int);

type whichPlayer =
  | P1
  | P2;

let rec rowMoves: int => list(move) = fun
  | 0 => []
  | p => [Row(p), ...rowMoves(p - 1)];

let rec colMoves: int => list(move) = fun
  | 0 => []
  | p => [Col(p), ...colMoves(p - 1)];

let availableMoves : state => list(move) = ((n, k):state) =>
  rowMoves(n) @ colMoves(k);

availableMoves(initialState);
What more

• We have game state, legal moves...
• Need to take a state and a move and determine the new state
• Need to know the status of the game (Ongoing? Did someone win? Is it a draw?)
• Will soon need to know the "value" of a game-state (how good it is for player 1)
• Let's do those...
type state = (int, int);
let initial_state = (2, 2);

type move =
  | Row(int)
  | Col(int);

type which_player =
  | P1
  | P2;

let next_state = ((n, k): state, m: move): state =>
  switch (m) {
    | Row(p) when p <= n => (n - p, k)
    | Col(p) when p <= k => (n, k - p)
    | _ => failwith("Illegal move.")
  };


type status =
  | Win(whichPlayer)
  | Draw
  | Ongoing(whichPlayer);

let game_status = (s: state): status =>
  switch (s) {
    | (n, k) => ???
  };


Need to enrich state:

```ocaml
type whichPlayer = P1 | P2;
type state = (int, int, whichPlayer);
let initial_state = (2, 2, P1);
type move = Row(int) | Col(int);

let next_state = ((n, k, p): state, m: move): state =>
  switch (m, p) {
    | (Row(p), P1) when p <= n => (n - p, k, P2)
    | (Col(p), P1) when p <= k => (n, k - p, P2)
    | (Row(p), P2) when p <= n => (n - p, k, P1)
    | (Col(p), P2) when p <= k => (n, k - p, P1)
    | _ => failwith("Illegal move.")
  };

type status = Win(whichPlayer) | Draw | Ongoing(whichPlayer);

let game_status = (s: state): status =>
  switch (s) {
    | (0, 0, w) => Win(w)
    | (_, _, w) => Ongoing(w)
  };

let value = (s: state): float =>
  switch (s) {
    | (0, 0, P1) => 1.0
    | (0, 0, P2) => -1.0
    | _ => failwith("value undefined for nonterminal states")
  };
```