Review: Curried Functions

> let prefix a b = a ^ b;;
val prefix : string -> string -> string = <fun>

prefix a b ⇐ (prefix a) b
string -> string -> string ⇐ string -> (string -> string)
Review: Curried Functions

> let prefix a b = a ^ b;;
val prefix : string -> string -> string = <fun>

prefix a b <\Rightarrow> (prefix a) b
string -> string -> string \leftrightarrow string -> (string -> string)

Expressions are evaluated from left to right (left -assoc.)
The \rightarrow symbol associates to the right

Example:

> let promote = prefix "Lady ";
let promote : string -> string string = <fun>

> prefix "Ms. " "Smith";;
- : string = "Ms. Smith"

> promote "Johnson";;
- : string = "Lady Smith"
What kind of traversal is this?

**depth-first**
Breadth-First v Depth-First Tree Traversal

- move queen
- move king
- move pawn 3
- move rook

Your Move

Their Move

Your Move

Their Move
Breadth-First v Depth-First Tree Traversal

move queen
move king

move pawn 3
... checkmate
move rook

Your Move
Their Move
Your Move
Their Move
Breadth-First v Depth-First Tree Traversal

binary trees as decision trees

Look for solution nodes

- Depth-first: search one subtree in full before moving on
- Breadth-first: search all nodes at level $k$ before moving to $k + 1$

Finds all solutions — nearest first!
type 'a tree = Lf
| Br of 'a * 'a tree * 'a tree

Br(1, Br(2, Br(4, Lf, Lf), Br(5, Lf, Lf)), Br(3, Lf, Lf))
Breadth-First Tree Traversal — Using Append

let rec nbreadth = function
    | [] -> []
    | Lf :: ts -> nbreadth ts
    | Br (v, t, u) :: ts ->
        v :: nbreadth (ts @ [t; u])

Keeps an enormous queue of nodes of search

Wasteful use of append

25 SECS to search depth 12 binary tree (4095 labels)

* careful: assumes depth starts at 1
Breadth-First Tree Traversal — Using Append

Notation in this example:

\[ \textit{Br}(v_A, t_B, t_C) \text{ is a tree } t_A \text{ with root value } v_A \text{ and subtrees } t_B, t_C \]

\[
\begin{align*}
nbreadth([t_A]) & \\
v_A :: nbreadth([], t_B, t_C) & \\
v_A :: nbreadth([t_B; t_C]) & \\
v_A :: v_B :: nbreadth([t_C; t_D; t_E]) & \\
v_A :: v_B :: v_C :: nbreadth([t_D; t_E; Lf; Lf]) & \\
v_A :: v_B :: v_C :: nbreadth([t_D; t_E; Lf; Lf]) & \\
... & \\
\end{align*}
\]
Breadth-First Tree Traversal — Using Append

Notation in this example:

$Br(v_A, t_B, t_C)$ is a tree $t_A$ with root value $v_A$ and subtrees $t_B, t_C$

\[
\begin{align*}
nbreadth([t_A]) \\
v_A :: nbreadth([] @ [t_B; t_C]) \\
v_A :: nbreadth([t_B; t_C]) \\
v_A :: v_B :: nbreadth([t_C] @ [t_D; t_E]) \\
v_A :: v_B :: nbreadth([t_D; t_E]) \\
v_A :: v_B :: v_C :: nbreadth([t_D; t_E] @ [Lf; Lf]) \\
v_A :: v_B :: v_C :: nbreadth([t_D; t_E; Lf; Lf]) \\
\ldots
\end{align*}
\]

(* ts is empty *)

(* put root value into list *)

(* execute append *)

(* append new subtrees *)

first arg of append grows!
Breadth-First Tree Traversal — Using Append

Two key operations in this example:

- Remove tree from head
- Add new subtrees to tail

The order matters:
Process what we first put into list first, before we process its descendants.

→ Find a better data-structure than ordinary list
An Abstract Data Type: Queues

We want: efficient FIFO data-structure

- `qempty` is the *empty queue*
- `qnull` *tests* whether a queue is empty
- `qhd` *returns* the element at the *head* of a queue
- `deq` *discards* the element at the *head* of a queue
- `enq` *adds* an element at the *end* of a queue
Efficient Functional Queues: Idea

Goal: avoid \( q@[x] \) since \( O(\text{length}(q)) \)

Key idea: reverse back half of list!

Represent the queue \( x_1 \ x_2 \ \ldots \ x_m \ y_n \ \ldots \ y_1 \) by a pair of lists

\[ ([x_1, x_2, \ldots, x_m], \ [y_1, y_2, \ldots, y_n]) \]

Add new items to \textit{rear list}

Remove items from \textit{front list}; if empty move \textit{rear} to \textit{front}

\textit{Amortized} time per operation is \( O(1) \)
Efficient Functional Queues: Idea

Goal: \[\text{deq} \quad [1; 2; 3; 4; 5; 6] \quad \text{enq} \quad 7\]

Functional queue: \(((1; 2; 3), [6; 5; 4])\)

pattern-match and discard

\[\text{cons} \quad 7\]

\[1 :: [2; 3] \quad 7 :: [6; 5; 4]\]

Result: \(((2; 3), [7; 6; 5; 4])\)

Rationale of amortized cost, for a queue of length \(n\):

- \(n\) \text{enq}, \(n\) \text{deq} operations
- \(2n\) cons operations for queue of length \(n\)
- \(O(1)\) cost per operation
type 'a queue = Q of 'a list * 'a list

let norm = function
  | Q ([], tls) -> Q (List.rev tls, [])
  | q -> q

let qnull q = (q = Q ([], []))

let enq (Q (hds, tls)) x =
  norm (Q (hds, x::tls))

exception Empty

let deq = function
  | Q (x:::hds, tls) -> norm (Q (hds, tls))
  | _ -> raise Empty
Breadth-First Tree Traversal — Using Queues

let rec breadth q =
  if qnull q then []
else
  match qhd q with
  | Lf -> breadth (deq q)
  | Br (v, t, u) ->
    v :: breadth (enq (enq (deq q) t) u)

0.14 secs to search depth 12 binary tree (4095 labels)

200 times faster!

* careful: assumes depth starts at 1
Iterative Deepening: Another Exhaustive Search

Breadth-first search examines $O(b^d)$ nodes:

**General formula:**

$$1 + b + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1}$$

$b = \text{branching factor}$$d = \text{depth}$

**For binary tree:** $2^{d+1} - 1$

Space and time complexity: $O(b^d)$

* careful: assumes depth starts at 0
Idea behind iterative deepening:
• Use DFS to get benefits of BFS
• Recompute nodes at depth $d$ instead of storing them
• Complexity: $\frac{b}{(b - 1)}$ times that for BFS (if $b > 1$)
• Space requirement at depth $d$ drops from $b^d$ to $d$

Recall depth-first search:

Space complexity: $O(d)$
Another Abstract Data Type: Stacks

- **empty** is the *empty stack*
- **null** *tests* whether a stack is empty
- **top** *returns* the element at the *top* of a stack
- **pop** *discards* the element at the *top* of a stack
- **push** *adds* an element at the *top* of a stack
1. **Depth-first**: use a stack  
   (efficient but incomplete)

2. **Breadth-first**: use a *queue*  
   (uses too much space!)

3. **Iterative deepening**: use (1) to get benefits of (2)  
   (trades time for space)

4. **Best-first**: use a *priority queue*  
   (heuristic search)

*The data structure determines the search!*