CSCI 2041: Lists and Recursion

Chris Kauffman Chris Kauffman

Last Updated:
Fri Dec 28 22:33:03 CST 2018

Logistics

- OCaml System Manual: 1.1 - 1.3
- Practical OCaml: Ch 1-2
- OCaml System Manual: 25.2 (Pervasives Modules)
- Practical OCaml: Ch 3, 9

Goals

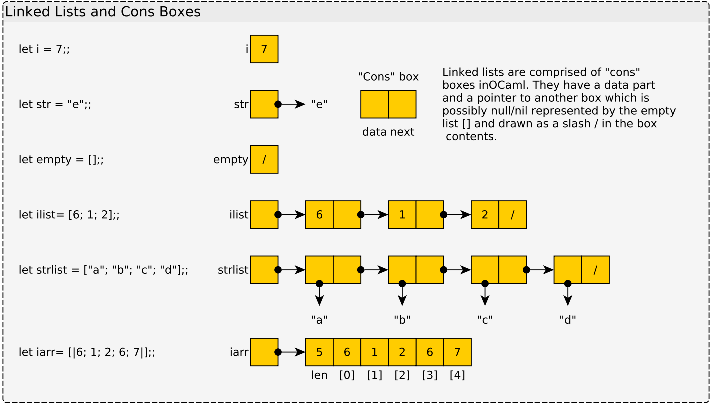
- Linked List data structure
- Recursive Functions
- Nested Scope

Assignment 1

- Due Wed 9/19 ~~Monday 9/17~~
- Note a few updates announced on Piazza / Changelog
- Questions?

Lists in Functional Languages

- Long tradition of **Cons boxes** and **Singly Linked Lists** in Lisp/ML languages
- Immediate list construction of with square braces: `[1;2;3]`
- Note **unboxed** `ints` and **boxed** strings and lists in the below[1]



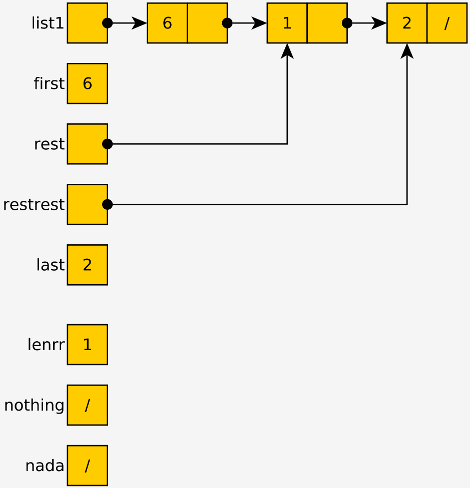
[1] "Boxed" means a pointer to data appears in the associated memory cell.

List Parts with Head and Tail

- `List.hd list` : "head", returns the first data element
- `List.tl list` : "tail", returns the remaining list

Accessing List Parts with List.hd and List.tl

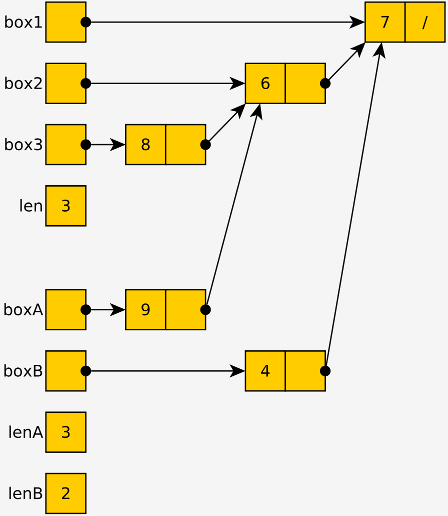
```
let list1 = [6; 1; 2];;

let first = List.hd list1;;

let rest = List.tl list1;;

let restrest = List.tl rest;;

let last = List.hd restrest;;


let lenrr = List.length restrest;;

let nothing = List.tl restrest;;

let nada = [];;
```



List Construction with "Cons" operator ::

Constructing a list with successive "cons" applications

```
let box1 = 7 :: [];;

let box2 = 6 :: box1;;

let box3 = 8 :: box2;;

let len = List.length box3;;


let boxA = 9 :: box2;;

let boxB = 4 :: box1;;

let lenA = List.length boxA;;

let lenB = List.length boxB;;
```



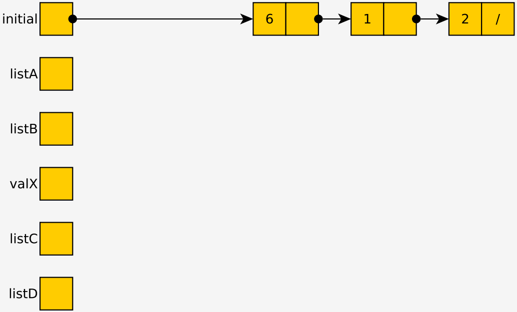
Immutable Data

- Lists are **immutable** in OCaml
  - Cannot change list contents once created
  - `let` bindings are also immutable
- Immutable data is certainly a disadvantage if you want to change it (duh)
- Immutability creates some significant advantages
  - Easier reasoning: it won't change
  - Compiler may be able to optimize based on immutability
  - Can share structure safely to reduce memory usage
- Will have more to say later about trade-offs with immutability (sometimes called "persistent data")

Exercise: List Construction/Decomposition

Fill in the Picture

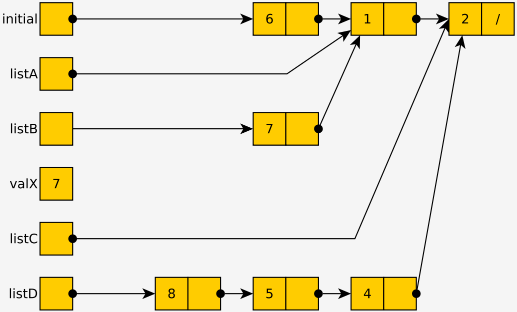
```
let initial= [6; 1; 2];;

let listA = List.tl initial;;

let listB = 7 :: listA;;

let valX = List.hd listB;;

let listC = (List.tl (List.tl listB));;

let listD= 8 :: 5 :: 4 :: listC;;
```



Answers: List Construction/Decomposition

Fill in the Picture: ANSWERS

```
let initial= [6; 1; 2];;

let listA = List.tl initial;;

let listB = 7 :: listA;;

let valX = List.hd listB;;

let listC = (List.tl (List.tl listB));;

let listD= 8 :: 5 :: 4 :: listC;;
```



Recursive Functions

- Introduce with recursive bindings with `let rec ...`
- Make use of a function in its own definition
- Will discuss how recursive functions actually "work" later

```
(* rec_funcs.ml : example recursive functions *)

(* sum the numbers 1 to n using recursion *)
let rec sum_1_to_n n =
  if n=1 then                            (* base case, reached 1 *)
    1                                    (* return 1 *)
  else                                   (* recursive case *)
    let below = n-1 in                   (* start point for nums below *)
    let sum_below = sum_1_to_n below in  (* recurse on nums below *)
    let ans = n+sum_below in             (* add on current n  *)
    ans                                  (* return as answer *)
;;

(* terse version of the same function *)
let rec sum_1_to_n n =
  if n=1 then
    1                                    (* base case *)
  else
    n + (sum_1_to_n (n-1))               (* recursive case *)
;;
```

Recursive Functions and Lists

- Typically do NOT iterate with linked lists directly
- Recurse on them for many basic functionalities like `length`

```
(* rec_listfuncs.ml : recursive functions on lists *)

(* Count the number of elements in a linked list *)
let rec list_length list =
  if list = [] then                        (* base case: empty list *)
    0                                      (* has length 0 *)
  else                                     (* recursive case *)
    let rest = List.tl list in             (* peel of tail *)
    let len_rest = list_length rest in     (* recursive call *)
    let ans = 1 + len_rest in              (* add on for current elem *)
    ans                                    (* return as answer *)
;;

(* terse version of the above *)
let rec list_length list =
  if list = [] then                        (* base case *)
    0
  else
    1 + (list_length (List.tl list))       (* recursive case *)
;;
```

Exercise: Counting Elements

- Below function counts how many times `elem` occurs in `list`
- Identify where the Base and Recursive cases appear in code
- Which line/lines have recursive calls?
- Explain why two if/else statements are needed

```
1  (* Count how many times elem appears in lst *)
2  let rec count_occur elem lst =
3    if lst = [] then
4      0
5    else
6      let first = List.hd lst in
7      let rest  = List.tl lst in
8      let rest_count = count_occur elem rest in
9      if elem = first then
10       1 + rest_count
11     else
12       rest_count
13 ;;
```

Answers: Counting Elements

- First `if/else` separates base and recursive cases
- Second `if/else` separates equal element (add one) form unequal
- Line 8 has recursive call

```
1  (* commented version of the above *)
2  let rec count_occur elem lst =
3    if lst = [] then                        (* base case: empty list *)
4      0                                     (* 0 occurrences *)
5    else                                    (* recursive case *)
6      let first = List.hd lst in            (* peel of head *)
7      let rest  = List.tl lst in            (* and tail of list *)
8      let rest_count = count_occur elem rest in (* count occurences in rest *
9      if elem = first then                  (* if current elem matches *)
10       1 + rest_count                      (* add 1 and return *)
11     else                                  (* otherwise *)
12       rest_count                          (* count in rest of lsit *)
13 ;;
```

Use Cons to Construct New Lists during Recursion

```
(* Create a new list which has list1 followed by list2; the builtin @
   operator does this via list1 @ list2; it functions similarly to the
   below version *)
let rec append_lists list1 list2 =
  if list1 = [] then                    (* base case: nothing in list1 *)
    list2                               (* just list2 *)
  else                                  (* recursive case *)
    let first = List.hd list1 in        (* get first and rest of list1 *)
    let rest  = List.tl list1 in
    let app_rest =                      (* answer for rest of list *)
            append_lists rest list2 in  (* recursive call *)
    let app_all = first :: app_rest in  (* cons on first elem to rest *)
    app_all
;;

(* terse version of the above *)
let rec append_lists list1 list2 =
  if list1 = [] then
    list2
  else
    (List.hd list1)  ::  (append_lists (List.tl list1) list2)
(*   |---first---| |Cons|               |---rest----|         *)
(*                        |--------recursive call----------|  *)
;;
```

Nesting Function Definitions

- Functions can be nested, e.g. defined in the local scope of another function

```
(* nested_funcs.ml : demonstrate nested functions *)

(* Return the sum of two factorials. Uses an internal function
   definition to compute factorials of parameters. *)
let sum_factorials n m =

  (* compute factorial recursively *)
  let rec fact i =                (* local recursive function *)
    if i<=1 then
      1                           (* base case *)
    else
      i * (fact (i-1))            (* recursive case *)
  in                              (* end local function definition *)

  let nfact = fact n in           (* call fact on n*)
  let mfact = fact m in           (* call fact on m *)
  nfact+mfact                     (* return sum of factorials *)
;;
(* end of function scope: fact no longer available *)
(* sum_factorials IS available, top-level binding *)
```

More examples in `nested_funcs.ml`

Combination Punch: List Functions with Recursive Helpers

- Frequently see all 3 techniques used for list functions
- Example: printing elements by index of a string list
- To properly recurse, must pass an extra paramter: index `i`
- Define a recursive helper function with additional params
- Call the recursive helper function to do the work

```
(* Print the number the index and element for a string list. Uses a
   nested recursive helper function. *)
let print_elems_idx strlist =
  let rec helper i lst =                (* recursive helper: 2 params *)
    if lst != [] then                   (* if any list left *)
      let first = List.hd lst in        (* grab first element *)
      let rest =  List.tl lst in        (* and rest of list *)
      Printf.printf "index %d : %s\n" i first;    (* print  *)
      helper (i+1) rest                 (* recurse on remaining list *)
  in                                    (* end helper definition *)
  helper 0 strlist;                     (* call helper starting at 0 *)
;;
```

Exercise: Elements Between

```
(* Create a list of the elements between the indices start/stop in the
   given list. Uses a nested helper function for most of the work. *)
let elems_between start stop list =
  let rec helper i lst =
    if i > stop then
      []
    else if i < start then
      helper (i+1) (List.tl lst)
    else
      let first = List.hd lst in
      let rest =  List.tl lst in
      let sublst = helper (i+1) rest in
      first :: sublst
  in
  helper 0 list
;;
```

- Describe the types for the parameters to function `elems_between`
- Describe the types for the parameters to function `helper`
- Where is the end of the definition of `helper`? Where is it used?
- What 3 situations are handled in the `if/else` block?
- How are the params of `helper` used?

Answers: Elements Between

```
let elems_between start stop list =      (* int -> int -> 'a list            *)
  let rec helper i lst =                 (* int -> 'a list -> 'a list        *)
    if i > stop then                     (* case for after stop index        *)
      []                                 (* end of possible elems between    *)
    else if i < start then               (* before the start index           *)
      helper (i+1) (List.tl lst)         (* recurse further along lst        *)
    else                                 (* case of start <= i <= stop       *)
      let first = List.hd lst in         (* get head and tail                *)
      let rest =  List.tl lst in
      let sublst = helper (i+1) rest in  (* recurse further to get sublst    *)
      first :: sublst                    (* cons first onto sublst, return   *)
  in                                     (* end helper definition            *)
  helper 0 list                          (* call helper at beginning of list *)
;;
```

- `helper` traverses list from beginning, eventually produces a sublist
- Param `i` is index into `list`, param `lst` is remainder of `list`
- When `i<start`, recurses further into `list`
- When `i>start`, returns empty list: no elements between after `stop`
- Between `start/stop` `helper` recurses then cons's on an element to the resulting list which is returned