CMSC 330: Organization of Programming Languages

Functional Programming with Lists
Lists in OCaml

• The basic data structure in OCaml
  – Lists can be of *arbitrary length*
    • Implemented as a linked data structure
  – Lists must be *homogeneous*
    • All elements have the same type

• Operations
  – Construct lists
  – Destruct them via pattern matching
Constructing Lists

Syntax

- `[]` is the empty list (pronounced “nil”)
- `e1::e2` prepends element `e1` to list `e2`
  - Operator `::` is pronounced "cons" (both from LISP)
  - `e1` is the head, `e2` is the tail
- `[e1; e2; ...; en]` is *syntactic sugar* for
  `e1::e2::...::en::[]`

Examples

```
3::[]   (* The list [3] *)
2::(3::[])   (* The list [2; 3] *)
[1; 2; 3] (* The list 1::(2::(3::[])) *)
```
Constructing Lists

Evaluation

• [] is a value
• To evaluate \( e_1 : : e_2 \), evaluate \( e_1 \) to a value \( v_1 \), evaluate \( e_2 \) to a (list) value \( v_2 \), and return \( v_1 : : v_2 \)

Consequence of the above rules:

• To evaluate \([e_1;...;e_n]\), evaluate \( e_1 \) to a value \( v_1 \), ...., evaluate \( e_n \) to a value \( v_n \), and return \([v_1;...;v_n]\)
Examples

# let y = [1; 1+1; 1+1+1] ;;
val y : int list = [1; 2; 3]

# let x = 4::y ;;
val x : int list = [4; 1; 2; 3]

# let z = 5::y ;;
val z : int list = [5; 1; 2; 3]

# let m = "hello"::"bob"::[];;
val z : string list = ["hello"; "bob"]
Typing List Construction

Nil:

\[
\text{[]} : \text{'a list}
\]
i.e., empty list has type \( t \text{ list} \) for any type \( t \)

Cons:

If \( e_1 : t \) and \( e_2 : t \text{ list} \) then \( e_1 :: e_2 : t \text{ list} \)

*With parens for clarity:*

If \( e_1 : t \) and \( e_2 : (t \text{ list}) \) then \( (e_1 :: e_2) : (t \text{ list}) \)
Examples

```ml
# let x = [1;"world"] ;;
This expression has type string but an expression was expected of type int

# let m = [[1];[2;3]];;
val y : int list list = [[1]; [2; 3]]

# let y = 0::[1;2;3] ;;
val y : int list = [0; 1; 2; 3]

# let w = [1;2]::y ;;
This expression has type int list but is here used with type int list list
  • The left argument of :: is an element, the right is a list
  • Can you construct a list y such that [1;2]::y makes sense?
```
Lists in Ocaml are Linked

- \([1;2;3]\) is represented above
  - A nonempty list is a pair (element, rest of list)
  - The element is the head of the list
  - The pointer is the tail or rest of the list
    - ...which is itself a list!

- Thus in math (i.e., inductively) a list is either
  - The empty list \([\ ]\)
  - Or a pair consisting of an element and a list
    - This recursive structure will come in handy shortly
Consider a Linked List in C

```c
struct list {
    int elt;
    struct list *next;
};
...
struct list *l;
...
i = 0;
while (l != NULL) {
    i++;
    l = l->next;
}
```
Lists of Lists

• Lists can be nested arbitrarily
  – Example: [ [9; 10; 11]; [5; 4; 3; 2] ]
    • (Type int list list)
Lists are Immutable

- No way to *mutate* (change) an element of a list
- Instead, build up new lists out of old, e.g., using ::

```plaintext
let x = [1;2;3;4]
let y = 5::x
let z = 6::x
```
Quiz 1

What is the type of the following expression?

```plaintext
[1.0; 2.0; 3.0; 4.0]
```

A. array
B. list
C. int list
D. float list
Quiz 1

What is the type of the following expression?

\[ [1.0; 2.0; 3.0; 4.0] \]

A. array
B. list
C. int list
D. float list
Quiz 2

What is the type of the following expression?

31::[3]

A. int
B. int list list
C. int list
D. error
Quiz 2

What is the type of the following expression?

31::[3]

A. int
B. int list list
C. int list
D. error
Quiz 3

What is the type of the following expression?

\[
[[[]; []; [1.3; 2.4]]]
\]

A. \textit{int list}
B. \textit{float list list list list}
C. \textit{float list list}
D. \textit{error}
Quiz 3

What is the type of the following expression?

```
[[[]; []; [1.3; 2.4]]]
```

A. `int list`
B. `float list list list`
C. `float list list`
D. `error`
Quiz 4

What is the type of the following definition?

```haskell
let f x = x :: (0 :: [])
```

A. int -> int
B. int list
C. int -> int list
D. int list -> int list
Quiz 4

What is the type of the following definition?

```
let f x = x::(0::[ ])
```

A. \text{int} \to \text{int}
B. \text{int list}
C. \text{int} \to \text{int list}
D. \text{int list} \to \text{int list}
Pattern Matching

• To pull lists apart, use the `match` construct

• Syntax

  ```plaintext
  match e with
  | p1 -> e1
  | ...
  | pn -> en
  ```

• `p1...pn` are `patterns` made up of `[]`, `::`, constants, and `pattern variables` (which are normal OCaml variables)

• `e1...en` are `branch expressions` in which pattern variables in the corresponding pattern are bound
Pattern Matching Semantics

match e with
| p1 -> e1
| ...
| pn -> en

- Evaluate e to a value v
- If p1 matches v, then evaluate e1 to v1 and return v1
- ...
- Else if pn matches v, then evaluate en to vn and return vn
- Else, no patterns match: raise Match_failure exception

- (When evaluating branch expression ei, any pattern variables in pi are bound in ei, i.e., they are in scope)
Pattern Matching Example

let is_empty l =
    match l with
    [] -> true
    | (h::t) -> false

- Example runs
  - is_empty []  (* evaluates to true *)
  - is_empty [1]  (* evaluates to false *)
  - is_empty [1;2](* evaluates to false *)
Pattern Matching Example (cont.)

let hd l =
    match l with
    (h::t) -> h

• Example runs
  - hd [1;2;3] (* evaluates to 1 *)
  - hd [2;3]  (* evaluates to 2 *)
  - hd [3]   (* evaluates to 3 *)
  - hd []    (* Exception: Match_failure *)
Quiz 5

To what does the following expression evaluate?

```
match ["doz";"zar"] with
  [] -> "kitteh"
| h::t -> h
```

A. "kitteh"
B. "zar"
C. "doz"
D. []
Quiz 5

To what does the following expression evaluate?

```haskell
match ["doz";"zar"] with
  [] -> "kitteh"
| h::t -> h
```

A. "kitteh"
B. "zar"
C. "doz"
D. []
"Deep" pattern matching

- You can nest patterns for more precise matches
  - \texttt{a::b} matches lists with \textbf{at least one} element
    - Matches \([1;2;3]\), binding \texttt{a} to \texttt{1} and \texttt{b} to \([2;3]\)
  - \texttt{a::[]} matches lists with \textbf{exactly one} element
    - Matches \([1]\), binding \texttt{a} to \texttt{1}
    - Could also write pattern \texttt{a::[]} as \([a]\)
  - \texttt{a::b::[]} matches lists with \textbf{exactly two} elements
    - Matches \([1;2]\), binding \texttt{a} to \texttt{1} and \texttt{b} to \texttt{2}
    - Could also write pattern \texttt{a::b::[]} as \([a;b]\)
  - \texttt{a::b::c::d} matches lists with \textbf{at least three} elements
    - Matches \([1;2;3]\), binding \texttt{a} to \texttt{1}, \texttt{b} to \texttt{2}, \texttt{c} to \texttt{3}, and \texttt{d} to \([\ ]\)
    - \textit{Cannot} write pattern as \([a;b;c]::d\) (why?)
Pattern Matching – Wildcards

• An underscore _ is a wildcard pattern
  – Matches anything
  – But doesn’t add any bindings
  – Useful to hold a place but discard the value
    • i.e., when the variable does not appear in the branch expression

• In previous examples
  – Many values of \( h \) or \( t \) ignored
  – Can replace with wildcard _
Pattern Matching – Wildcards (cont.)

• Code using _
  
  - let is_empty l = match l with  
    
      [] -> true | (____) -> false  
  
  - let hd l = match l with (h::__) -> h  
  
  - let tl l = match l with (__::t) -> t  

• Outputs
  
  - is_empty[1] (* evaluates to false *)  
  
  - is_empty[ ] (* evaluates to true *)  
  
  - hd [1;2;3] (* evaluates to 1 *)  
  
  - tl [1;2;3] (* evaluates to [2;3] *)  
  
  - hd [1] (* evaluates to 1 *)  
  
  - tl [1] (* evaluates to [ ] *)
Pattern Matching – An Abbreviation

• \texttt{let f p} = \texttt{e}, where \texttt{p} is a pattern
  – is shorthand for \texttt{let f x = match x with p -> e}

• Examples
  – \texttt{let hd (h:::_)} = \texttt{h}
  – \texttt{let tl (_,::t)} = \texttt{t}
  – \texttt{let f (x::y:::)} = \texttt{x + y}
  – \texttt{let g [x; y]} = \texttt{x + y}

• Useful if there’s only one acceptable input
Pattern Matching Typing

- If $e$ and $p_1$, ..., $p_n$ each have type $ta$
- and $e_1$, ..., $e_n$ each have type $tb$
- Then entire $\text{match}$ expression has type $tb$

Examples

- **type:** $'a \text{ list} \rightarrow 'a$
  
  let hd $l =$
  
  \[
  \text{match } l \text{ with } \begin{cases} 
  (h::\_ ) \rightarrow h \\
  \end{cases}
  \]

  $ta = 'a \text{ list}$  

  $tb = 'a$

- **type:** $\text{int list} \rightarrow \text{int}$
  
  let rec sum $l =$
  
  \[
  \text{match } l \text{ with } \begin{cases} 
  [ ] \rightarrow 0 \\
  (h::t) \rightarrow h+\text{sum } t \\
  \end{cases}
  \]

  $ta = \text{int list}$  

  $tb = \text{int}$
Polymorphic Types

• The `sum` function works only for `int` lists
• But the `hd` function works for *any type of list*
  - `hd [1; 2; 3]` (* returns 1 *)
  - `hd ["a"; "b"; "c"]` (* returns "a" *)

• OCaml gives such functions *polymorphic* types
  - `hd : 'a list -> 'a`
    - this says the function takes a list of *any* element type `'a`, and returns something of that same type

• These are basically generic types in Java
  - `'a list` is like `List<T>`
Examples Of Polymorphic Types

• let tl (_::_:t) = t
  # tl [1; 2; 3];;
  - : int list = [2; 3]
  # tl [1.0; 2.0];;
  - : float list = [2.0]
  (* tl : 'a list -> 'a list *)

• let fst x y = x
  # fst 1 "hello";;
  - : int = 1
  # fst [1; 2] 1;;
  - : int list = [1; 2]
  (* fst : 'a -> 'b -> 'a *)
Examples Of Polymorphic Types

• let hds (x::_) (y::_) = x::y::[
  # hds [1; 2] [3; 4];;
  - : int list = [1; 3]
  # hds ["kitty"] ["cat"];;
  - : string list = ["kitty"; "cat"]
  # hds ["kitty"] [3; 4] -- type error
  (* hds: 'a list -> 'a list -> 'a list *)

• let eq x y = x = y  (* let eq x y = (x = y) *)
  # eq 1 2;;
  - : bool = false
  # eq "hello" "there";;
  - : bool = false
  # eq "hello" 1  -- type error
  (* eq : 'a -> 'a -> bool *)
Quiz 6

What is the type of the following function?

```ml
let f x y =
  if x = y then 1.0 else 2.0
```

A. 'a -> 'b -> float
B. 'a -> 'a -> bool
C. 'a -> 'a -> float
D. float
Quiz 6

What is the type of the following function?

```ocaml
let f x y =
  if x = y then 1.0 else 2.0
```

A. 'a -> 'b -> float
B. 'a -> 'a -> bool
C. 'a -> 'a -> float
D. float
Missing Cases

• Exceptions for inputs that don’t match any pattern
  – OCaml will warn you about non-exhaustive matches

• Example:

```ocaml
# let hd l = match l with (h::_) -> h;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]

# hd [];;
Exception: Match_failure ("", 1, 11).
```
Pattern matching is **AWESOME**

1. You can’t forget a case
   - Compiler issues inexhaustive pattern-match warning

2. You can’t duplicate a case
   - Compiler issues unused match case warning

3. You can’t get an exception
   - Can’t do something like `List.hd []`

4. Pattern matching leads to elegant, concise, beautiful code
More Examples

• let f l =
  match l with (h1::(h2::_)) -> h1 + h2
  - f [1;2;3]
  - (* evaluates to 3 *)

• let g l =
  match l with [h1; h2] -> h1 + h2
  - g [1; 2]
  - (* evaluates to 3 *)
  - g [1; 2; 3]
  - (* error! no pattern matches *)
Pattern Matching Lists of Lists

• You can do pattern matching on these as well

• Examples
  - let addFirsts ((x::_) :: (y::_) :: _) = x + y
    • addFirsts [ [1; 2; 3]; [4; 5]; [7; 8; 9] ] = 5
  
  - let addFirstSecond ((x::_):_ :: (_::y::_) :: _) = x + y
    • addFirstSecond [ [1; 2; 3]; [4; 5]; [7; 8; 9] ] = 6

• Note: You probably won’t do this much or at all
  – You’ll mostly write recursive functions over lists
  – We’ll see that soon
Lists and Recursion

• Lists have a recursive structure
  – And so most functions over lists will be recursive

```
let rec length l = match l with
  [] -> 0
| (_::t) -> 1 + (length t)
```

– This is just like an inductive definition
  • The length of the empty list is zero
  • The length of a nonempty list is 1 plus the length of the tail

– Type of length?
  • `'a list -> int`
More Examples

• \texttt{sum l (* sum of elts in l *)}
  \[
  \text{let rec sum l = match l with} \\
  \quad [] \rightarrow 0 \\
  \quad | (x::xs) \rightarrow x + (\text{sum xs})
  \]

• \texttt{negate l (* negate elements in list *)}
  \[
  \text{let rec negate l = match l with} \\
  \quad [] \rightarrow [] \\
  \quad | (x::xs) \rightarrow (-x) :: (\text{negate xs})
  \]

• \texttt{last l (* last element of l *)}
  \[
  \text{let rec last l = match l with} \\
  \quad [x] \rightarrow x \\
  \quad | (x::xs) \rightarrow \text{last xs}
  \]
More Examples (cont.)

(* return a list containing all the elements in the list l followed by all the elements in list m *)

- append l m
  
  let rec append l m = match l with
  
  | [] -> m
  
  | (x::xs) -> x::(append xs m)

- rev l  (* reverse list; hint: use append *)
  
  let rec rev l = match l with
  
  | [] -> []
  
  | (x::xs) -> append (rev xs) [x]

- rev takes $O(n^2)$ time. Can you do better?
A Clever Version of Reverse

let rec rev_helper l a = match l with
  | [] -> a
  | (x::xs) -> rev_helper xs (x::a)
let rev l = rev_helper l []

• Let’s give it a try

rev [1; 2; 3] →
rev_helper [1;2;3] [] →
rev_helper [2;3] [1] →
rev_helper [3] [2;1] →
rev_helper [] [3;2;1] →
[3;2;1]