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”)
- \(e_1 :: e_2\) prepends element \(e_1\) to list \(e_2\)
  - Operator \(::\) is pronounced "cons"
  - \(e_1\) is the head, \(e_2\) is the tail
- \([e_1; e_2; \ldots; e_n]\) is syntactic sugar for \(e_1 :: e_2 :: \ldots :: e_n :: [\ ]\)

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 \([e_1; \ldots; e_n]\), evaluate \(e_1\) to a value \(v_1\), \ldots, evaluate \(e_n\) to a value \(v_n\), and return \([v_1; \ldots; v_n]\)
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 m : string list = ["hello"; "bob"]
Typing List Construction

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

Cons:
If \( e_1 : t \) and \( e_2 : t \) list then \( e_1 : : e_2 : t \) list

With parens for clarity:
If \( e_1 : t \) and \( e_2 : (t \) list) then \((e_1 : : e_2) : (t \) list)
Examples

# 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
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 `::`

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

What is the type of the following expression?

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

A. array
B. list
C. float list
D. int list
What is the type of the following expression?

\[1.0; 2.0; 3.0; 4.0]\n
A. array
B. list
C. float list
D. int list
Quiz 2

What is the type of the following expression?

31::[3]

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

What is the type of the following expression?

31::[3]

A. int
B. int list
C. int list list
D. error
What is the type of the following definition?

\[
\text{let } f \ x = \text{"alien"}::[x]
\]

A. string -> string
B. string list
C. string list -> string list
D. string -> string list
Quiz 3

What is the type of the following definition?

\[
\text{let } f \ x = \text{“alien”}::[x]
\]

A. string -> string
B. string list
C. string list -> string list
D. string -> string list
Pattern Matching

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

• Syntax

```
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
Pattern Matching Example

```ml
let is_empty l =
    match l with
    | [] -> true
    | _ -> 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 4

To what does the following expression evaluate?

```
match [1;2;3] with
  [ ] -> [0]
  | h::t -> t
```

A. [ ]
B. [0]
C. [1]
D. [2;3]
Quiz 4

To what does the following expression evaluate?

```plaintext
cmpatch [1;2;3] with
    []  -> [0]
| h::t  -> t
```

A. []
B. [0]
C. [1]
D. [2;3]
"Deep" pattern matching

- You can nest patterns for more precise matches
  - `a::b` matches lists with **at least one** element
    - Matches `[1;2;3]`, binding `a` to 1 and `b` to `[2;3]`
  - `a::[]` matches lists with **exactly one** element
    - Matches `[1]`, binding `a` to 1
    - Could also write pattern `a::[]` as `[a]`
  - `a::b::[]` matches lists with **exactly two** elements
    - Matches `[1;2]`, binding `a` to 1 and `b` to 2
    - Could also write pattern `a::b::[]` as `[a;b]`
  - `a::b::c::d` matches lists with **at least three** elements
    - Matches `[1;2;3]`, binding `a` to 1, `b` to 2, `c` to 3, and `d` to []
    - *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 \texttt{h} or \texttt{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 *)
  - `hd [1]` (* evaluates to 1 *)
  - `tl [1;2;3]` (* evaluates to [2;3] *)
  - `tl [1]` (* evaluates to [ ] *)
Quiz 5

To what does the following expression evaluate?

```haskell
match [1;2;3] with
  | 1::[]    -> [0]
  | _::_     -> [1]
  | 1::_::[] -> []
  | 1::_::[] -> []
```

A. []
B. [0]
C. [1]
D. [2;3]
To what does the following expression evaluate?

```
match [1;2;3] with
    | 1::[]        -> [0]
    | __::__       -> [1]
    | 1::__::[]    -> []
```

A. []
B. [0]
C. [1]
D. [2;3]
Pattern Matching – An Abbreviation

• \( \text{let } f \ p = e, \) where \( p \) is a pattern
  – is shorthand for \( \text{let } f \ x = \text{match } x \text{ with } p \rightarrow e \)

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

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

- If $e$ and $p_1, \ldots, p_n$ each have type $ta$
- and $e_1, \ldots, e_n$ each have type $tb$
- Then entire `match` expression has type $tb$

Examples

**type**: `‘a list -> ‘a`

```ml
let hd l =  
match l with
  (h::_) -> h

let rec sum l =  
match l with
  [] -> 0
  | (h::t) -> h+sum
```

**type**: `int list -> int`

```ml
let rec sum l =  
match l with
  [] -> 0
  | (h::t) -> h+sum
```

$ta = ‘a list$ $tb = ‘a$

$ta = int list$ $tb = int$
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 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 *)
What is the type of the following function?

```
let f x y =
    if x = y then 1 else 0
```

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

What is the type of the following function?

```latex
let f x y =
  if x = y then 1 else 0
```

A. 'a -> 'b -> int
B. 'a -> 'a -> bool
C. 'a -> 'a -> int
D. int
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
Lists and Recursion

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

- 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

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

- **sum l (* sum of elts in l *)**
  
  ```ocaml
  let rec sum l = match l with
      [] -> 0
    | (x::xs) -> x + (sum xs)
  ```

- **negate l (* negate elements in list *)**
  
  ```ocaml
  let rec negate l = match l with
      [] -> []
    | (x::xs) -> (-x) :: (negate xs)
  ```

- **last l (* last element of l *)**
  
  ```ocaml
  let rec last l = match l with
      [x] -> x
    | (x::xs) -> 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`
  
  ```ocaml
  let rec append l m = match l with
    [] -> m
  | (x::xs) -> x::(append xs m)
  ```

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

• `rev` takes $O(n^2)$ time. Can you do better?