Lists and Pattern Matching

Prof. Greg Morrisett, Dean of CIS
Fall 2015

Today’s music: "Blank Space" by Taylor Swift
Review

Previously in 3110:

• **Functions**: definition, application, anonymous, higher-order
• **Variables**: bindings, scope

Today:

• **Lists**: OCaml's awesome built-in datatype
• **Pattern matching**: an awesome feature not found in most imperative languages
Lists: An introduction

```ocaml
let lst = [1;2;3]
let empty = []

let longer = 5::lst
let another = 5::1::2::3::[]

let rec sum xs =
  match xs with
  | [] -> 0
  | h::t -> h + sum t

let six = sum lst
let zero = sum empty
```
Lists: An introduction

```plaintext
let lst = ["abc"; "def"; "ghi"]

let rec concat ss =
  match ss with
  | [] -> ""
  | s::ss' -> s ^ (concat ss ss')

let a_i = concat lst
```
Building lists

Syntax:
• [ ] is the empty list
• e1::e2 prepends element e1 to list e2
• [e1; e2; ...; en] is syntactic sugar for e1::e2::...::en::[]

[ ] is pronounced "nil"
:: is pronounced "cons" (both from LISP)

Syntactic sugar: redundant kind of syntax that makes program "sweeter" or easier to write
“Syntactic sugar causes cancer of the semi-colon.”

First recipient of the Turing Award

for his “influence in the area of advanced programming techniques and compiler construction”

1922-1990
Building 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 $[e_1; \ldots; e_n]$, evaluate $e_1$ to a value $v_1$, $\ldots$, evaluate $e_n$ to a value $v_n$, and return $[v_1; \ldots; v_n]$. 
Building lists

New types:
For any type $t$, the type $t\ \text{list}$ describes lists where all elements have type $t$
• $[1;2;3] : \text{int list}$
• $[\text{true}] : \text{bool list}$
• $[\[1+1;2-3\];[3*7]] : \text{int list list}$

Nil:
[]:'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 $(e_1::e_2) : (t\ \text{list})$
Accessing lists

A list can only be:
• nil, or
• the cons of an element onto another list

Use **pattern matching** to access list in one of those ways:

```plaintext
let empty lst =
  match lst with
  | []      -> true
  | h::t    -> false
```
Recursion!

Functions over lists are usually recursive: only way to “get to” all the elements

• What should the answer be for the empty list?
• What should the answer be for a non-empty list?
  – Typically in terms of the answer for the tail of the list
Example list functions

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

let rec length xs =
    match xs with
    | [] -> 0
    | h::t -> 1 + length t

let rec append lst1 lst2 =
    match lst1 with
    | [] -> lst2
    | h::t -> h::(append t lst2)

(* append is available as operator @ *)
```
Lists are immutable

• No way to *mutate* an element of a list
• Instead, build up new lists out of old
e.g., :: and @
Match expressions

Syntax:

```
match e with
  | p1  -> e1
  | p2  -> e2
  | ... 
  | pn  -> en
```

the $\texttt{pi}$ are patterns
the first pipe is optional
line breaks are optional

e.g.,

```
let empty lst =
  match lst with []  -> true | h::t   -> false
```
Patterns

Patterns have their own **syntax**

For now, a pattern can be any of these:

- a variable name (e.g., \texttt{x})
- \[ \]
- \texttt{p1::p2}
- an underscore \_ 

As we learn more data structures, we'll learn more patterns
Patterns

Patterns match values

Intuition of matching is that pattern "looks like" the value, if variables in the pattern are replaced by pieces of the value

•  [ ] looks like  [ ]
•  h::t looks like 2::3
•  x looks like  [ 1; 2; 3 ]
•  _ looks like anything

...we'll make this precise later
**Match expressions**

**Evaluation:**

- Evaluate $e$ to a value $v$
- If $p_1$ matches $v$, then evaluate $e_1$ to a value $v_1$ and return $v_1$
- Else, if $p_2$ matches $v$, then evaluate $e_2$ to a value $v_2$ and return $v_2$
- ...  
- Else, if $p_n$ matches $v$, then evaluate $e_n$ to a value $v_n$ and return $v_n$
- Else, if no patterns match, raise an exception

When evaluating branch expression $e_i$, any pattern variables that matched are in scope

Type checker will warn you if you write an *inexhaustive pattern match*

...so you can prevent exceptions from being raised at runtime by fixing your code when compiler warns you
**Match expressions**

```plaintext
match e with
  | p1 -> e1
  | p2 -> e2
  | ...
  | pn -> en
```

**Type-checking:**
If $e$ and $p_1 \ldots p_n$ have type $\text{ta}$ and $e_1 \ldots e_n$ have type $\text{tb}$ then entire match expression has type $\text{tb}$
Pattern matching

The pattern [] matches the value [] and nothing else

```haskell
match [] with
| []   -> 0
| h::t -> 1  (* evaluates to 0 *)
```

```haskell
match [] with
| h::t  -> 0
| []    -> 1  (* evaluates to 1 *)
```
Pattern matching

The pattern \texttt{h::t} matches any list with at least one element, and binds that element to \texttt{h}, and any remaining list to \texttt{t}

\begin{verbatim}
match [1;2;3] with
|  []   -> 0
|  h::t -> h  (* evaluates to 1 *)
\end{verbatim}

\begin{verbatim}
match [1;2;3] with
|  []   -> [0]
|  h::t -> t  (* evaluates to [2;3] *)
\end{verbatim}
A tricky pattern match

What's wrong with this code?

```ml
let rec drop_val v l =
match l with
| [] -> []
| v :: l' -> drop_val v l'
| h :: t -> drop_val v t
```

*Hint: compiler warning (as configured in VM)*
A tricky pattern match

What's wrong with this code?

```ocaml
let rec drop_val v l =
  match l with
  | [] -> []
  | v::l' -> drop_val v l'
  | h::t -> drop_val v t
```

*The v in the pattern shadows the argument v*
A tricky pattern match

let rec drop_val v l =
  match l with
  | [] -> []
  | h::t -> let t' = drop_val v t in
            if h=v then t' else h::t'
Deep pattern matching

• Pattern `a::[]` matches all lists with exactly one element
• Pattern `a::b` matches all lists with at least one element
• Pattern `a::b::[]` matches all lists with exactly two elements
• Pattern `a::b::c::d` matches all lists with at least three elements
Accessing lists, with poor style

• Two library functions that return head and tail
  \texttt{List.hd}, \texttt{List.tl}

• \textbf{Not idiomatic} to apply directly to a list
  – Because they throw exceptions; you can easily write buggy code
  – Whereas pattern matching guarantees no exceptions when destructing list; it’s hard to write buggy code!
Why pattern matching is AWESOME

1. You can’t forget a case
   (inexhaustive pattern-match warning)
2. You can’t duplicate a case
   (unused match case warning)
3. You can’t get an exception
   from forgetting to test the variant
   (e.g., `hd []`)
4. Pattern matching leads to elegant, concise, beautiful code
Functions that immediately match

Instead of

```ml
let f x =
    match x with
    | p1 -> e1
    | ...  
    | pn -> en
```

can use another piece of syntactic sugar

```ml
let f = function
| p1 -> e1
| ...  
| pn -> en
```
Tail recursion

# length [0; 1; ...; 1_000_000];;
Stack overflow during evaluation (looping recursion?).

Why?

let rec length xs =
    match xs with
    | [] -> 0
    | h::t -> 1 + length t
Tail recursion

Solution: When recursive call is the only thing left to do in computation, compiler reuses the stack frame. Reduces space from $O(n)$ to $O(1)$.

```ml
let rec length_plus_n n = function
| [] -> n
| h::t -> length_plus_n (n+1) t

let length_tr = length_plus_n 0
```
Lists (recap)

- Syntax: \[ \] :: [a; b; c]
- Semantics: building with nil and cons, accessing with pattern matching
- Idioms: recursive functions with pattern for nil and for cons, \texttt{function} syntactic sugar, tail recursion
- Library: awesome higher-order functions in OCaml standard library (next time)
Upcoming events

• [Monday and Tuesday] Recitations canceled because of Labor Day
• [next Thursday] A1 due

This is awesome.

THIS IS 3110