Lists and Pattern Matching

Prof. Clarkson
Fall 2016

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

Previously in 3110:
• **Functions:** definition, application, anonymous, partial application

Today:
• **Lists:** a built-in datatype
• **Pattern matching:** an awesome feature not found in most imperative languages
Lists

```ml
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
```
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
Alan J. Perlis

“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
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 \([e_1; \ldots; e_n]\), evaluate \(e_1\) to a value \(v_1\), ...., evaluate \(e_n\) to a value \(v_n\), and return \([v_1; \ldots; v_n]\)
List types

For any type \( t \), the type \( t \ \texttt{list} \) describes lists where all elements have type \( t \)

- \( [1;2;3] : \texttt{int\ list} \)
- \( [\text{true}] : \texttt{bool\ list} \)
- \( [[1+1;2-3];[3*7]] : \texttt{int\ list\ list} \)
List types

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 \( (e_1 :: e_2) : (t \ list) \)
Question

What is the type of \(31::[10]\)?

A. int
B. int list
C. int int list
D. int list list
E. Not well-typed
Question

What is the type of $31: : [10]$?

A. int

B. int list

C. int int list

D. int list list

E. Not well-typed
Destructing 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:

```ocaml
let empty lst =
  match lst with
  | [] -> true
  | h::t -> false
```
Example list functions

```ocaml
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 @ *)
```
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
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
```
Match expressions

the first vertical bar is optional
line breaks are optional

e.g.,

```latex
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., \( x \))
• [ ]
• \( p_1 : : p_2 \)
• 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

- [ ] matches [ ] and nothing else
- h:t matches [2] as well as [1;3] and [9;8;7] ...
- x matches all the above
- _ matches everything
  (that's the underscore character, called wildcard)
**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
Match expressions

Type-checking:
If $e$ and $p_1 \ldots p_n$ have type $\mathit{ta}$
and $e_1 \ldots e_n$ have type $\mathit{tb}$
then entire match expression has type $\mathit{tb}$
To what value does the above expression evaluate?

A. “zar”
B. “doz”
C. “kitteh”
D. []
E. h
To what value does the above expression evaluate?

A. “zar”
B. “doz”
C. “kitteh”
D. []
E. h

```plaintext
match ["zar";"doz"] with
  []    -> "kitteh"
  | h::t -> h
```
"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, List.tl}

• \textbf{Not idiomatic} to apply directly to a list
  – Because they raise exceptions; \textit{you can easily write buggy code}
  – Whereas pattern matching guarantees no exceptions when destructing list; \textit{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., \texttt{hd \ [ \ ]})
4. Pattern matching leads to elegant, concise, beautiful code
Functions that immediately match

Instead of

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

can use another piece of syntactic sugar

```haskell
let f = function
    | p1 -> e1
    | ...  
    | pn -> en
```
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, function syntactic sugar  

• Library: awesome higher-order functions in OCaml standard library (next week)
Upcoming events

• [Wed] A1 out:
  • Wed rec and Thur lecture cover remaining material needed
  • Due following Wed
• [soon] PINs issued to waitlist

This is awesome.

THIS IS 3110