Lists

Today’s music: "Blank Space" by Taylor Swift
I could show you incredible things // Magic, madness, heaven, sin
So it's gonna be forever // Or it's gonna go down in flames //
You can tell me when it's over // If the high was worth the pain
Attendance

- Practiced with i>clickers (Thur) and one-minute-memo (Mon)
- Starting today counts toward grade
- **Important thing:** you are present and participating most of the time
- **Unimportant things:** you are here all the time, you have a reason to be elsewhere on a given day
  - Generous, unspecified number of absences will be ignored at the end of the semester
Review

Previously in 3110:

• **Functions:** definition, application, anonymous, partial application

Today:

• **Lists:** a built-in datatype

• **Pattern matching:** an incredible feature not found in most imperative languages
Lists

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
• \( e_1::e_2 \) prepends element \( e_1 \) to list \( e_2 \)
• \( [e_1; e_2; ...; e_n] \) is syntactic sugar for \( e_1::e_2::...::e_n::[] \)

\( [] \) 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
Building lists

Evaluation:
• \([\ ]\) is a value
• To evaluate \(\mathbf{e}_1::\mathbf{e}_2\), evaluate \(\mathbf{e}_1\) to a value \(\mathbf{v}_1\), evaluate \(\mathbf{e}_2\) to a (list) value \(\mathbf{v}_2\), and return \(\mathbf{v}_1::\mathbf{v}_2\)

Consequence of the above rules:
• To evaluate \([\mathbf{e}_1; \ldots; \mathbf{e}_n]\), evaluate \(\mathbf{e}_1\) to a value \(\mathbf{v}_1\), \ldots, evaluate \(\mathbf{e}_n\) to a value \(\mathbf{v}_n\), and return \([\mathbf{v}_1; \ldots; \mathbf{v}_n]\)
List types

For any type $\mathbf{t}$, the type $\mathbf{t}$ list describes lists where all elements have type $\mathbf{t}$

- $[1;2;3]$ : int list
- $[true]$ : bool list
- $[[1+1;2-3];[3*7]]$ : int list list
List types

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

Cons:
If e1 : t and e2 : t list then e1::e2 : t list

With parens for clarity:
If e1 : t and e2 : (t list) then (e1::e2) : (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
Accessing parts of 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
```
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 is your new bff

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

Sometimes tail recursion becomes important (see notes)
Lists are immutable

- No way to *mutate* an element of a list
- Instead, build up new lists out of old
e.g., :: and @
Lists are linked

• i.e., *singly-linked lists*

• Data structures (like languages and pole arms) are tools: none is perfect

• Singly-linked lists are good for sequential access of short-to-medium length lists
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.,
let empty lst =

match lst with [] -> true | h::t -> false

(* though lst=[] would be better *)
Patterns

Patterns have their own syntax

For now, a pattern can be any of these:
• a variable name (e.g., x)
• [ ]
• 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

- [ ] 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*)
  - (it's like a blank space)
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 e with
| p1 -> e1
| p2 -> e2
| ...
| pn -> en
Match expressions

Type-checking:
If $e$ and $p_1 \ldots p_n$ have type $ta$
and $e_1 \ldots e_n$ have type $tb$
then entire match expression has type $tb$

match $e$ with
| $p_1$ -> $e_1$
| $p_2$ -> $e_2$
| ...
| $p_n$ -> $e_n$
To what value does the above expression evaluate?
A. “taylor”
B. “swift”
C. “1989”
D. []
E. h
match ["taylor";"swift"] with
| []  -> "1989"
| h::t -> h

To what value does the above expression evaluate?
A. “taylor”
B. “swift”
C. “1989”
D. []
E. 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}, \texttt{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 accessing list; \textit{it’s hard to write buggy code!}
  – Gray area: when invariant or precondition guarantees list is non-empty
Why pattern matching is INCREDIBLE

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
   (e.g., `hd []`)

4. Pattern matching leads to elegant, concise, beautiful code
Functions that immediately match

Instead of

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

can use another piece of syntactic sugar

```plaintext
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

• None for today

This is incredible.

THIS IS 3110