CS 3110

Lists

Prof. Clarkson

Fall 2017

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
- `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

1922-1990

"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"

Building lists

Evaluation:

- `[]` is a value
- To evaluate `e1::e2`, evaluate `e1` to a value `v1`, evaluate `e2` to a (list) value `v2`, and return `v1::v2`

Consequence of the above rules:

- To evaluate `[e1; ...; en]`, evaluate `e1` to a value `v1`, ..., evaluate `en` to a value `vn`, and return `[v1; ...; vn]`

List types

For any type **`t`**, the type **`t list`** describes lists where all elements have type **`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:

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

Example list functions

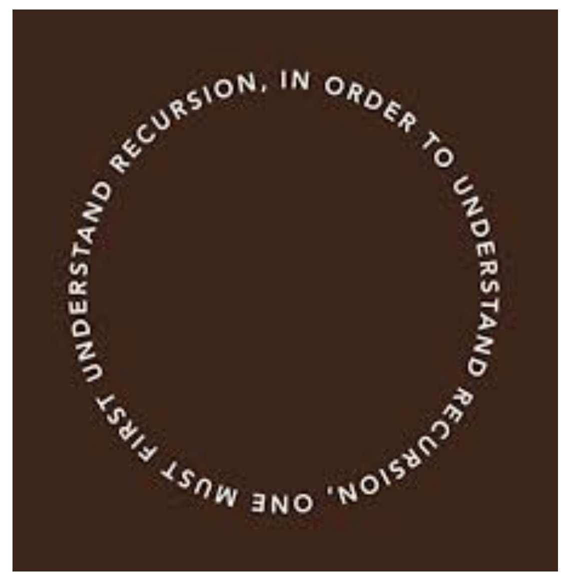
```
let rec sum xs =
  match xs with
  | [] -> 0
  | h::t -> h + sum t

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

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

Evaluation:

- Evaluate **`e`** to a value **`v`**
- If **`p1`** matches **`v`**, then evaluate **`e1`** to a value **`v1`** and return **`v1`**
- Else, if **`p2`** matches **`v`**, then evaluate **`e2`** to a value **`v2`** and return **`v2`**
- …
- Else, if **`pn`** matches **`v`**, then evaluate **`en`** to a value **`vn`** and return **`vn`**
- Else, if no patterns match, raise an exception

When evaluating branch expression **`ei`**, any pattern variables that matched are in scope

Match expressions

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

Type-checking:

If `e` and `p1...pn` have type `ta`
and `e1...en` have type `tb`
then entire match expression has type `tb`

Question

```
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

Question

```
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
  `List.hd, List.tl`

- **Not idiomatic** to apply directly to a list
  - Because they raise exceptions; you can easily write buggy code
  - Whereas pattern matching guarantees no exceptions when accessing list; 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

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

can use another piece of syntactic sugar

```
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