Lecture 4

Thursday, September 2, 2021 1:55 PM

Structuring Data with Types

Basic types 1 : int, float, char, bool

Basic types 2 : unit, function (___ → ___)

Standard types 1: string, pair, tuple,
list, exn, option

Standard types 2: array

Custom types: variants , records
$\downarrow$
Algebraic Data Types (struct in C)

---

type status = Valid | Invalid

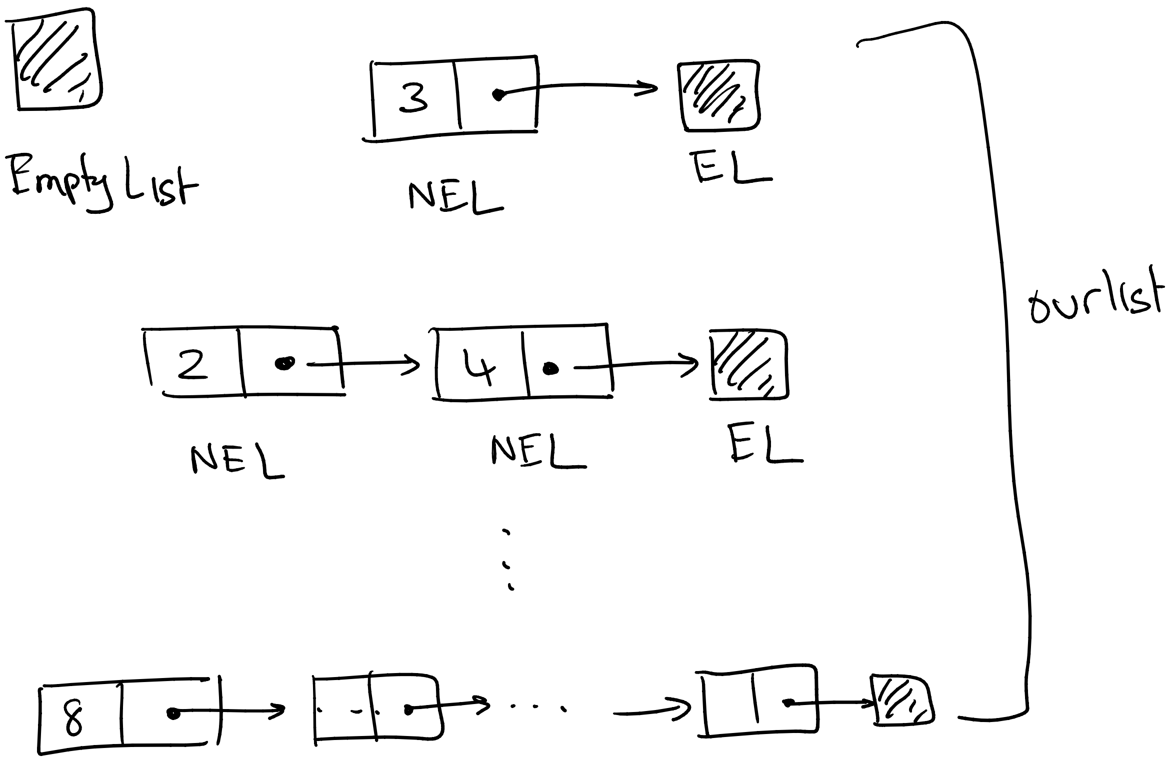
$\uparrow$
"or"

- Values of type status can be either Valid or Invalid
- "That is all they can be".

- That is all they can be.
  "Closed world assumption".

Recursive Types

```
type ourlist = EmptyList
             | Not EmptyList of int * ourlist.
               NEL
```

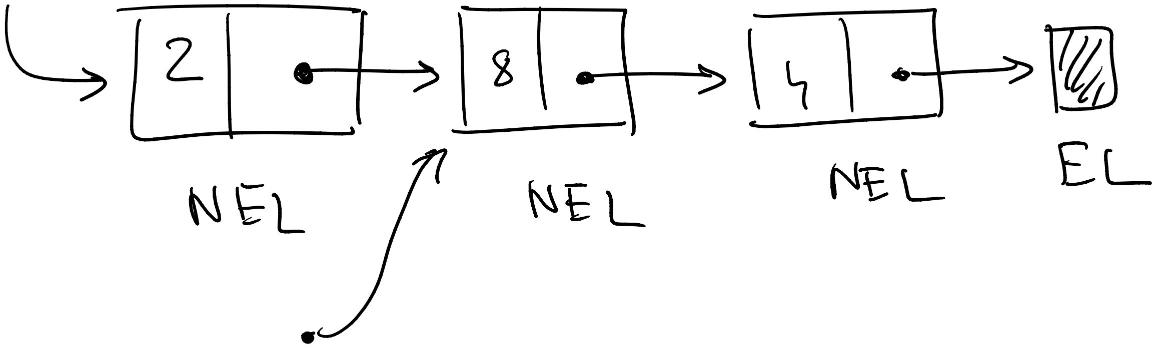


```
let cons hd tl = NEL (hd, tl)

let car l = match l with
```

```
| EL -> ???
| NEL(hd, _) -> hd
```

```
let cdr l = match l with
                | EL -> ???
                | NEL(_, tl) -> tl
```



Computing the length of a list

```
let ourlen l =
  match l with
   | EL -> 0
   | NEL(_, tl) -> 1 + ourlen tl
```

```
let rec ourlen l =
  match l with
   | EL -> 0
   | NEL(_, tl) -> 1 + ourlen tl
```

```
| NEL(_, tl) -> 1 + ourlen tl
```

???

Attempts to resolve ourlen in a scope before the present fn is defined

```
| NEL(_, tl) -> 1 + ourlen tl
```

Adds ourlen to scope before present definition

To be able to print ADTs

Magic: Use [@@ deriving show] after definition

Concatenating lists

```
let rec concat l1 l2 =
    match l1 with
    | [] -> l2
```

[2;9;3] [4;2;8]

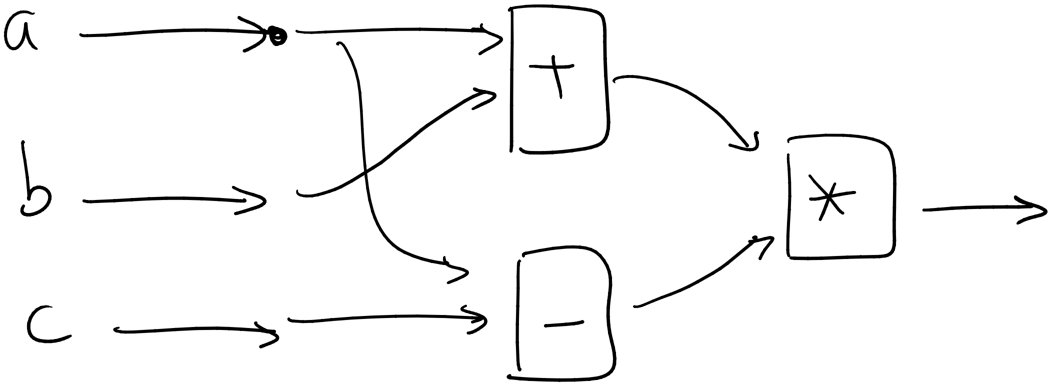
⇒ [2; 9; 3 ;4;2;8]

$$| \; [\,] \rightarrow \underline{\quad\quad l_2 \quad\quad} \Rightarrow [2;\ \underline{9;3};\ \underline{4;2;8}]$$

$$| \; hd :: tl_1 \rightarrow \underline{hd :: (\text{concat}}\ \underbrace{tl_1}\ l_2)$$

Recursive definition terminates because first argument is strictly decreasing.

$$\text{let funny\_foo } a\ b\ c = (a \underline{+} b) \underline{*} (a - c)$$



- No matter how big a, b, c are, this circuit will always be 3 <u>gates</u> big.

- In the absence of recursion, all computations result in circuits of bounded size.

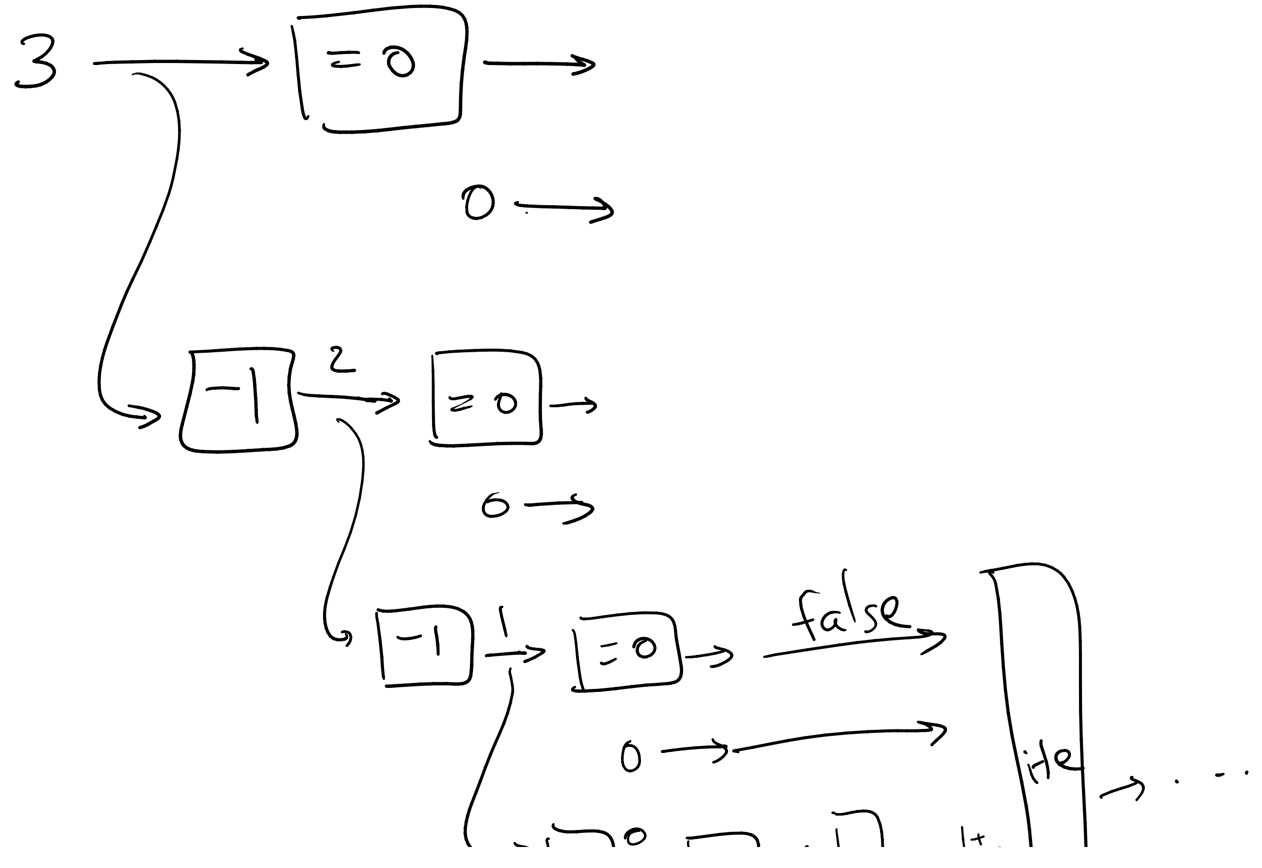
```
let sum n =
    if n = 0 then 0
    else n + sum (n-1)
```

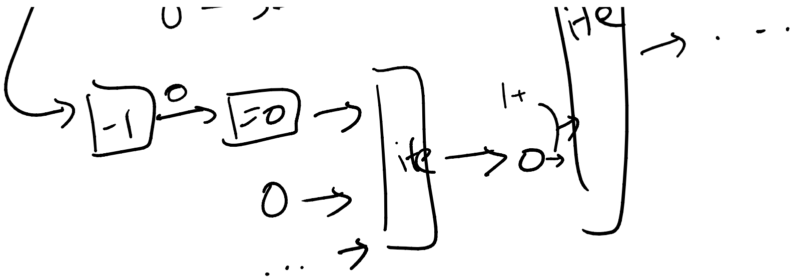
$$\sum_{i=0}^{n} i$$

0 + 1 + 2 + ... + n

---

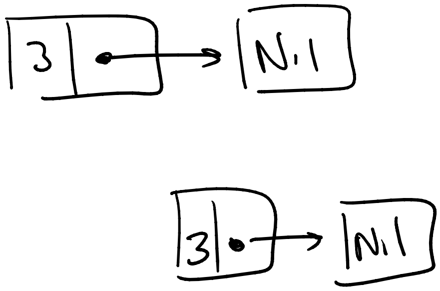
sum 3





Recursion provides the ability to build circuits that are arbitrarily large.

Structural equality	vs. Physical equality
=, <>	==, != (Don't worry about these for now.)



① Checks if values have the same shape

② If so, recurses

③ Might not terminate for cyclic structures

Merging sorted lists

```
let rec merge l1 l2 =
(* Assume that l1 is sorted *)
(* Assume that l2 is sorted *)
(* We want to merge them into a single
sorted list *)
match l1, l2 with
| [], _ -> l2
| _, [] -> l1
| hd1 :: tl1, hd2 :: tl2 ->
if hd1 < hd2
then hd1 :: merge tl1 l2
else hd2 :: merge l1 tl2
```

Quick-and-dirty QuickSort

```
let rec qsort l =
match l with
| [] -> []
| hd :: tl ->
let tl1 = List.filter ((>) hd) tl in
let tl2 = List.filter ((<=) hd) tl in
let stl1 = qsort tl1 in
let stl2 = qsort tl2 in
stl1 @ [hd] @ stl2
```

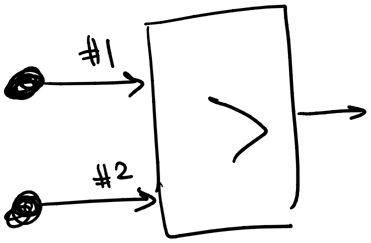
All elements of tl which are smaller than hd

All elements of tl which are at least as large as hd

Sorted versions of tl_1 & tl_2 resp.

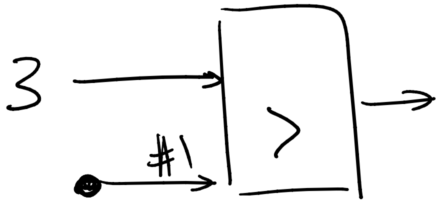
tl_1 & tl_2 contain strictly fewer elements than l.

Concatenate everything



Returns true iff

#1 > #2



Returns true iff

3 > #1