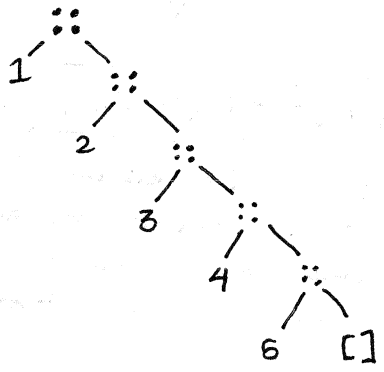
* ~~Fold-left~~ List structure:

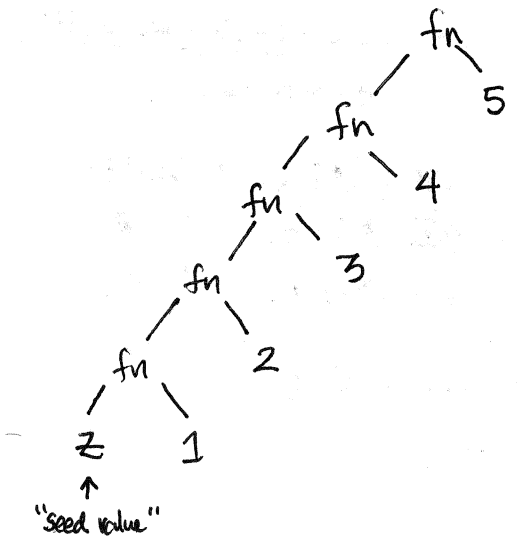
- everything can be reduced to a set of cons onto the empty list:

$$[1;2;3;4;5] \rightarrow 1::2::3::4::5::[]$$

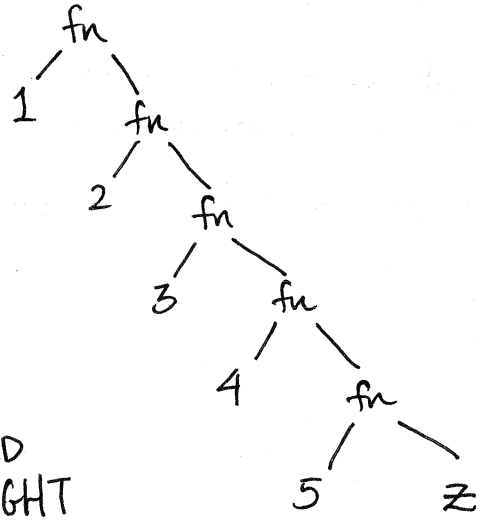
This can be represented in tree form =>



This helps us visualize the next 2 functions, fold_left and fold_right:



FOLD LEFT



FOLD RIGHT

$('a \rightarrow 'b \rightarrow 'a) \rightarrow 'a \rightarrow 'b\ list \rightarrow 'a$

$('a \rightarrow 'b \rightarrow 'b) \rightarrow 'a\ list \rightarrow 'b \rightarrow 'b$

HIGHER-ORDER FUNCTIONS / 8 APR 2011

Stephen Kou
CS 131

* The main concept: FUNCTIONS ARE VALUES! They can go anywhere that a number can go (conceptually). ↑ "FIRST-CLASS FUNCTIONS"
  - Local variables via 'let'
  - Arguments to functions
  - Return from function

* Currying: A fundamental idea is instead of ~~taking~~ a function taking multiple arguments, we can have a chain of functions, each taking a single argument!

`(A * B * C) → D`

- single function, takes a tuple of 3 elements.
- must be called with all the elements!

```
let func = fun (a,b,c) → ...
```

"UNCURRIED"

`A → (B → (C → D))`

First function takes an 'A' and returns a function that takes a 'B', which returns a function that takes a C, that returns a ~~function~~ D.

```
let func = fun a b c → ...
```

"CURRIED"

* Functions that take functions as input, or output functions, are called HIGHER-ORDER FUNCTIONS.
  - Analog to C/C++: Function pointers can be used to emulate higher-order functions.

```
D foo(A a, B b, C c) {
  ...
}
```

A NON-CURRIED FUNCTION

```
typedef D (*f)(C); C_D;
typedef C_D (*g)(B) B_C;
typedef B_C (*h)(A) A_B;

B_C foo(A a) {
  ...
}
```

EMULATING CURRYING

BUILT-IN HIGHER ORDER FUNCTIONS (Key for understanding Homework 2!!)

* The List module in OCaml contains many useful higher-order functions. For example:

filter:	`('a → bool) → 'a list → 'a list`	`("filter p l")` (p: func, l: list)
find:	`('a → bool) → 'a list → 'a list`	`("find p l")`
map:	`('a → 'b) → 'a list → 'b list`	`[fn a1; fn a2; fn a3; ...; fn an]`
map2:	`('a → 'b → 'c) → 'a list → 'b list → 'c list`	`[fn a1 b1; fn a2 b2; ... ; fn an bn]`