2: EVALUATION

"Perfection is achieved, not when there is nothing left to add, but when there is nothing left to take away."

Antoine de Saint-Exupéry, 1939

Previously we calculated the cost of isort [3,2,1]:

```
> isort :: [Int] -> [Int]
> isort []
> isort (x:xs) = insert x (isort xs)
```

```
> insert :: Int -> [Int] -> [Int]
> insert x [] = [x]
> insert x (y:ys)
>   | x <= y = x:y:ys
>   | otherwise = y: insert x ys
```

We had the following evaluation:

```
    isort [3,2,1]
⇝
    insert 3 (isort [2,1])
⇝
    insert 3 (insert 2 (isort [1]))
⇝
    insert 3 (insert 2 (insert 1 (isort [])))
```

↝

insert 3 (insert 2 (insert 1 []))

↝

insert 3 (insert 2 [1])

↝

insert 3 (1 : insert 2 [])

↝

insert 3 (1 : [2])

↝

head (1 : insert 3 [2])

normal order ↝ (C application / strict)

head (1 : 2 : insert 3 [])

↝

head (1 : 2 : [3])

What is the cost of the following?

```
> minimum :: [Int] -> Int
> minimum xs = head (isort xs)
```

minimum [3,2,1]

↝

head (isort [3,2,1])

↝

... { 10 steps }

↝

head [1, 2, 3]

↝

1

13 steps.

The number of steps depends on the strategy:

* Applicative Order (Strict)
  - reduce the left-most, inner-most first

* Normal Order (Lazy)
  - reduce the left-most, outer-most first

Here is a simple expression language:

e ::= x — variables
| k — constants
| f e_1 ... e_n — application
| if e then e_1 else e_2 — conditional.

Assume functions always of the form:

f x_1 ... x_n = e

```
insert x xs = if null xs
                then x:[]
                else if x ≤ head xs
                       then x:xs
                       else insert x (tail xs)
                            ↑
                          head xs :
```

To analyse the (strict) time complexity of our expression language, we need a function T.

$T_f\ x_1 \dots x_n$ time it takes to evaluate $f\ x_1 \dots x_n$

(or)

$T(f)\ x_1 \dots x_n$

We define T by:

$$T(f)\ x_1 \dots x_n = 0 \quad \text{when } f \text{ is primitive.}$$

Other functions are of the form:

$$f\ x_1 \dots x_n = e$$

$$T(f)\ x_1 \dots x_n = 1 + T(e)$$

$T(x) = 0$

$T(k) = 0$

$T(f\ e_1 \dots e_n) = T(f)\ e_1 \dots e_n + T(e_1) + \dots + T(e_n)$

$T(\text{if } e \text{ then } e_1 \text{ else } e_2) = T(e) + \text{if } e \text{ then } T(e_1) \text{ else } T(e_2)$

Let's calculate the cost of length xs

```
> length xs = if null xs then 0
>                        else 1 + length (tail xs).
```

$T(\text{length xs})$

$=$

$T(\text{length})\ xs + T(xs)$

$=$

$1 + T(\text{if null xs then 0 else 1 + length (tail xs)})$

$= 1 + T(\text{null xs}) + \text{if null xs then } T(0) \text{ else } T(1 + \text{length (tail xs)})$ (0)

$= 1 + \text{if null xs then 0 else } T(1 + \text{length (tail xs)})$

$= 1 + \text{if null xs then 0 else } T(+)\ 1\ (\text{length (tail xs)})$

$$+ \; T(1) + T(\text{length}\ (\text{tail}\ xs))$$

$$= 1 + \text{if}\ \text{null}\ xs\ \text{then}\ 0\ \text{else}\ T(\text{length})\ (\text{tail}\ xs) + T(\text{tail})\ xs$$

Therefore, we have:

$$T(\text{length})\ \underset{n}{xs} = 1 + \text{if}\ \text{null}\ xs\ \text{then}\ 0\ \text{else}\ T(\text{length})\ \underset{n-1}{(\text{tail}\ xs)}$$

---

composition rule:

The cost of $f(g(x))$ is given by $T(f(g(x)))$

$$\begin{aligned} T(f(g(x))) &= T(f)\ (g\ x) + T(g\ x) \\ &= T(f)\ (g\ x) + T(g)\ x + T(x) \\ &= T(f)\ (g\ x) + T(g)\ x \end{aligned}$$

Consider foldl:

foldl (⊕) ϕ [x₁ ... xₙ]

=

(((ϕ ⊕ x₁) ⊕ x₂) ⋯ ⊕ xₙ)

```
> foldl :: (b → a → b) → b → [a] → b
> foldl f k [] = k
> foldl f k (x:xs) = foldl f (f k x) xs
```

```
> sum :: [Int] → Int
> sum xs = foldl (+) 0
```

How many steps does this take?

sum [1, 2, 3]

⇝

foldl (+) 0 [1, 2, 3]

⇝

foldl (+) (0+1) [2, 3]

⇝

foldl (+) ((0+1)+2) [3]

⇝

foldl (+) (((0+1)+2)+3) []

} this is costly -

⇝
((0+1)+2)+3)

⇝
(1+2)+3

⇝
3+3

⇝
6

strictness

```
> foldl' f k (x:xs) = foldl' f $! (f k x) xs
```