Ling 185A

Winter 2021

Week 2: Recursion

Section plan

1. Review: HW1 #14
2. Brief recap on recursion & recursive functions on Numb
3. `IntList` type and `[Int]`

I. Functions as "arguments"

Remember in our hw, you were asked to evaluate the following expression:

```
f ((\fn -> fn Rock) (\x -> whatItBeats x))
```

with `f` being defined as `f = \s -> case s of {Rock -> 334; Paper -> 138; Scissors -> 99}`. We see `f` is a *lambda expression*, which could be potentially reduced to a *case expression*.

`f`	`((\fn -> fn Rock) (\x -> whatItBeats x))`
↑	↑
λ-expression	some expression e

- Let's forget about this `f` for a second and focus on e.

`((\fn -> fn Rock)`	`(\x -> whatItBeats x))`
↑	↑
λ-expression	some expression e_1

To do *lambda reduction*, a lambda expression should be adjacent to some expression as in `(\v -> e) e'`. You want to substitute any free occurrences of the variable v in the expression e with e'.

NOTE: a lambda expression, even it seems to be a function in nature, is also an expression! In other words, functions are not really that special.

In this current example, `fn` is the variable name. To do lambda reduction, we want to substitute any free occurrences of `fn` with the lambda expression e_1 in the expression at the right side of the arrow, which leads to:

```
(\x -> whatItBeats x) Rock
whatItBeats Rock              lambda reduction
```

II. Recursion

1. Broadly speaking, recursion is when something is defined in terms of **itself**.

   The point is that some problem-solving mechanisms/instances of the same type are very similar to *itself*. In order to solve the problem/define the type, you can depend on the same mechanism/the same thing with *smaller size*.

2. **Recursive types**: when values that may contain other values of the same type

   Example:

```
data Numb = Z | S Numb deriving Show
```

```
S (S Z) ::  Numb
    ↑
(S Z) ::  Numb
    ↑
Z ::  Numb   ← base case: a form that does not contain an instance of that type.
```

```
data Form = T | F | Neg Form | Cnj Form Form | Dsj Form Form deriving Show
```

Recursive functions: when a function calls itself.

3. Ingredients for writing recursive functions/types/expressions

   - To avoid getting lost in infinite recursion in the computation, you need identify one/more base case(s) (often 0, 1, Z, (S Z), an empty list, a list with one element, empty string etc.).
   - Think about how the problem can be divided into the same problem but with smaller size.
   - Think about how the solutions to sub-problems can be combined into a solution to the bigger problem.

Exercise:

- `lessThanOrEq`: outputs whether the first Numb it sees is less than or equal to the second Numb.
- `difference`: outputs the absolute difference between the two Numbs.

III. `IntList` and `[Int]`

Now, we can define our own lists, say integers using recursion.

```
data IntList = Empty | NonEmpty Int IntList deriving Show
```

For example, the list containing 5 followed by 7 followed by 2 (and nothing else) would be represented as:

```
NonEmpty 5 (NonEmpty 7 (NonEmpty 2 Empty)) ::  IntList
                    ↑
           NonEmpty 7 (NonEmpty 2 Empty)) ::  IntList
                              ↑
                     (NonEmpty 2 Empty) ::  IntList
                                  ↑
                                  Empty ::  IntList ← base case
```

Now, let's see what we can do with this data type. The following function calculates the sum of such a list of integers.

```
total ::  IntList -> Int
total = \l -> case l of
              Empty -> 0
              NonEmpty x rest -> x + total rest
```

Exercise:

Now I want to write a recursive function `contains`, which checks whether there's any `Int` in this `IntList` that meet certain requirements (e.g >2; is 0 or not...).

```
contains ::  (Int -> Bool) -> (IntList -> Bool)
```

Haskell has a built-in type `[Int]` to represent lists, which uses some compact syntax. The compact syntax is convenient, but it can obscure the fact that this built-in type actually has exactly the same kind of recursive structure as this `IntList` type. Using this built-in type, we write the list containing 5 then 7 then 2 as following, which can be further rewritten as `[5, 7, 2]` for convenience.

```
5 :  ( 7 :  (2 :  []))
```

- `[]` : Empty
- the built-in type uses the colon ("cons") instead of NonEmpty; and the colon is written between its two arguments, unlike `NonEmpty` and other constructors we've seen.

```
total ::  [Int] -> Int
total = \l -> case l of
              [] -> 0
              x:rest -> x + total rest
```