Symbolic & Declarative Computing / Artificial Intelligence (COS5012-B)

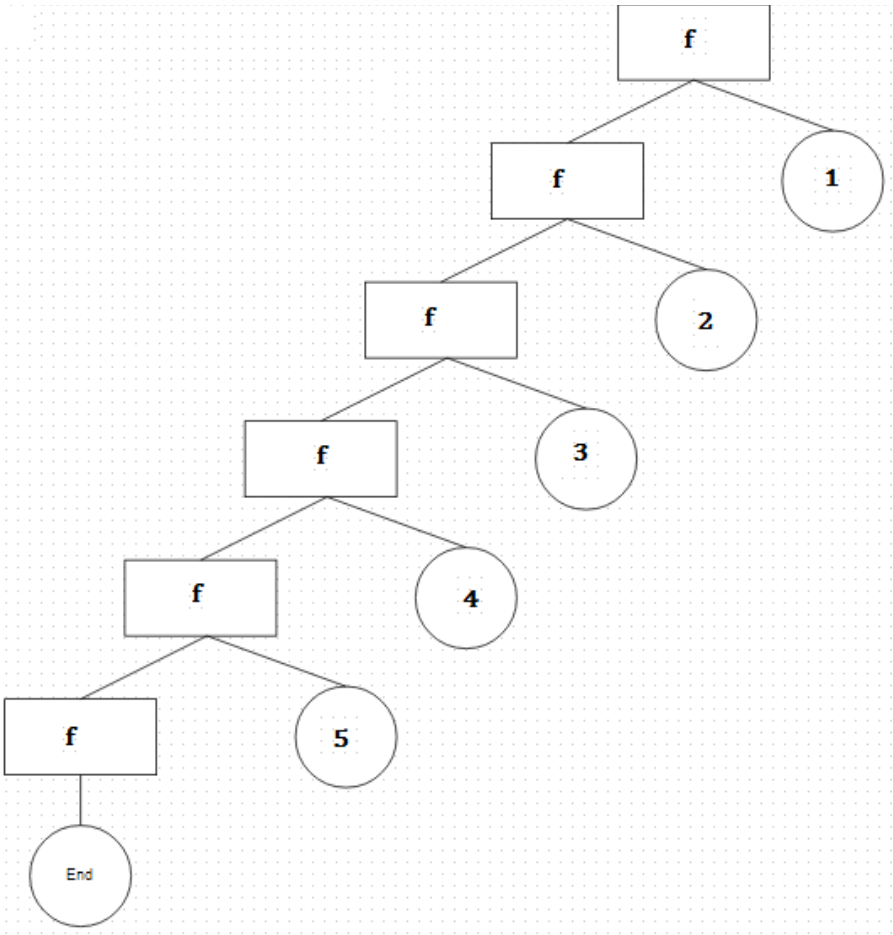
November 03, 2016 Week 6 Lab3

LAB MANUAL: APPLICATION OF HIGHER ORDER FUNCTIONS

In this tutorial you will learn about

- Difference between left recursion and right recursion
- Lambda Functions
- Application of higher order functions (zipWith, map, filter, fold)

Left Recursion vs. Right Recursion



Exercise

- Draw a recursion tree for right recursion.
- Draw a recursion tree for the following expression

  `foldr (count) 0 "Haskell"`

  Implementation of count function is given below

```
count :: a -> Int -> Int
count x y = 1 + y
```

- What does the above `foldr` command computes?
- Can the same count function work with `foldl`? If yes try to use it and if not try to implement a new one.

Lambdas

Lambdas are basically anonymous functions that are used because we need some functions only once. Normally, we make a lambda with the sole purpose of passing it to a higher-order function. For example the length function developed with fold can be re-implemented as below:

```
lengthr' lst = foldr (\x y -> 1 + y) 0 lst
lengthl' lst = foldl (\x y -> x + 1) 0 lst
```

- Lambdas can take any number of parameters
- Pattern matching can be done in lambdas. The only difference is that you can't define several patterns for one parameter, like making a `[]` and a `(x:xs)` pattern for the same parameter and then having values fall through. If a pattern matching fails in a lambda, a runtime error occurs, so be careful when pattern matching in lambdas!

Exercise

1. Define a function `squareall :: [Int] -> [Int]` which takes a list of integers and produces a list of the squares of those integers. For example, `squareall [6, 1,(-3)] = [36, 1, 9]`

2. Define a function **`nestedreverse`** which takes a list of strings as its argument and reverses each element of the list and then reverses the resulting list. Thus,

   `nestedreverse ["in", "the", "end"] = ["dne", "eht", "ni"]`

3. Define a function `atfront :: a -> [[a]] -> [[a]]` which takes an object and a list of lists and sticks the object at the front of every component list. For example,

   `atfront 7 [[1,2], [], [3]] = [[7,1,2], [7], [7,3]].`

4. Define a function **`lengths`** which takes a list of strings as its argument and returns the list of their lengths. For example,

   `lengths ["the", "end", "is", "nigh"] = [3, 3, 2, 4].`

5. Define a function `parity :: [String] -> [Int]` which takes a list of strings and returns a list of the integers 0 and 1 such that 0 is the nth element of the value if the nth string of the argument contains an even number of characters and 1 is the

nth element of the value if the nth string contains an odd number of characters. For example,

```
parity ["one", "two", "three", "four"] = [1, 1, 1, 0].
```

6. Define a function **`wtel`** (without the empty list) which removes every occurrence of the empty list from a list of lists. Thus,

   ```
   wtel [[1, 2], [], [1, 3]] = [[1,2], [1, 3]].
   ```

7. Define a function **`caen`** (containing an even number) which takes a list of lists of integers as its argument and removes from it every list not containing an even number. Thus,

   ```
   caen [[1,3], [2,1], [7,9], [2, 4, 8]] = [[2,1], [2, 4,8]]
   ```

8. Define a function **`wvowel`** (without vowels) which removes every occurrence of a vowel from a list of characters.

9. Define a function **`wiv`** (without internal vowels) which takes a list of strings as its argument and removes every occurrence of a vowel from each element. For example,

   ```
   wiv ["the", "end", "is", "nigh"] = ["th", "nd", "s", "ngh"]
   ```

10. Define **`reverse`**, which reverses a list, using `foldr`.

11. Using `foldr`, define a function **`remove`** which takes two strings as its arguments and removes every letter from the second list that occurs in the first list. For example,

    ```
    remove "first" "second" = "econd"
    ```

12. The function **`remdups`** removes adjacent duplicates from a list. For example,

    ```
    remdups [1, 2, 2, 3, 3, 3, 1, 1] = [1, 2, 3, 1]
    ```

    Define `remdups` using `foldr`. Give another definition using `foldl`.

13. The function **`inits`** returns the list of all initial segments of a list. Thus,

    ```
    inits "ate" = [[], "a", "at", "ate"].
    ```

    Define `inits` using `foldr`.