Dr. Greg Lavender
Department of Computer Sciences
University of Texas at Austin
lavender@cs.utexas.edu
Classic Recursive Functions

- Euclid’s Greatest Common Divisor (GCD) function
- Factorial function
- Fibonacci function
GCD Function

The GCD of a pair of integers $(x, y)$ is defined by taking the remainder $r$ of $(\text{abs } x)$ divided by $(\text{abs } y)$. If $r$ is 0, return $x$. Otherwise compute GCD of $y$ and $r$.

$$\text{gcd} :: (\text{Int}, \text{Int}) \rightarrow \text{Int}$$

$$\text{gcd} (x, y) = \text{gcd'} (\text{abs } x) (\text{abs } y)$$

where

$$\text{gcd'} x 0 = x$$

$$\text{gcd'} x y = \text{gcd'} y (x \mod y)$$

$$\text{gcd} (-98, 16) = \text{gcd'} 16 (98 \mod 16)$$
$$= \text{gcd'} 16 2$$
$$= \text{gcd'} 2 (16 \mod 2)$$
$$= \text{gcd'} 2 0$$
$$= 2$$
Factorial Function

A recursive factorial algorithm implementing the function \( n! \) first counts down from \( n \) to 0 by recursively descending to the bottom-out condition, then performs \( n \) multiplications as the recursion ascends back up.

\( 0! = 1 \)
\( n! = n \times (n-1)! \) for all \( n > 0 \)

\begin{verbatim}
fact :: Integer -> Integer
fact n | n == 0 = 1  -- base case terminates recursion
    | n > 0  = n \times fact (n-1)
    | otherwise = error "fact: negative value for n"

fact 3 => 3 \times fact(2) => 3 \times (2 \times fact(1)) =>
3 \times (2 \times (1 \times fact(0))) => 3 \times (2 \times (1 \times 1)) =>
3 \times (2 \times 1) => 3 \times 2 => 6
\end{verbatim}
Fibonacci Function

- The Fibonacci numbers are the infinite integer sequence 0,1,1,2,3,5,8,13,21,..., in which each item is formed by adding the previous two, starting with 0 and 1.
- E.g., 0+1→1, 1+1→2, 1+2→3, 2+3→5

- The Fibonacci function is defined recursively as:

\[
\text{fib} :: \text{Integer} \rightarrow \text{Integer} \\
\text{fib} 0 = 0 \\
\text{fib} 1 = 1 \\
\text{fib} n = \text{fib}(n-2) + \text{fib}(n-1)
\]

- Notice that in computing fib n, we do two recursive calls and the sum up their results. Furthermore, the two calls are duplicative in the sense that computing fib(n-1) necessarily computes fib(n-2) all over again! Using this kind of “double” recursion is terribly inefficient.
Recursive List Data Type

- A list is a recursively defined data type with elements of some type `a`, e.g., \([1,2,3]\) is a list of type `int`

- \([\ ]\) constructs the empty list; `:` is an infix right associative list constructor operator (cons), that constructs a new list from an element of type `a` on the left and a list `[a]` on the right

\[
data \ [a] = \[\] \mid a : [a]
\]

\[
3 : [\ ] = \[3\]; \ 2 : [3] = \[2,3\]; \ 1 : [2,3] = \[1,2,3\] = 1 : 2 : 3 : [\]
\]

- let `head [a_1,a_2,...,a_n] = a_1`; tail \([a_1,a_2,...,a_n]\) = \([a_2,...,a_n]\)
- `head [] = error`; `tail [] = error`

- let `(x:xs)` match \([a_1,a_2,...,a_n]\), then \(x = a_1\), \(xs = [a_2,...,a_n]\)

- let `++` be list concatenation: \([1,2] ++ [3,4] = [1,2,3,4]\)
Recursive List Functions

length :: [a] -> Int
length [] = 0  -- empty list is the base case
length (x:xs) = 1 + length xs

sum :: (Num a) => [a] -> Int
sum [] = 0  -- empty list is the base case
sum (x:xs) = x + sum xs

mean :: (Num a) => [a] -> Float
mean lst = sum lst / length lst

-- Note: mean requires 2 traversals of the list!
-- Can we compute the mean using just one traversal?
-- let (x,y) be an ordered pair, then
-- fst (x,y) = x; snd (x,y) = y

sumlen :: (Num a) => [a] -> (Int,Int) -> (Int,Int)
sumlen [] = p
sumlen (x:xs) p = sumlen xs (x + fst(p), 1 + snd(p))

mean lst = fst(p) / snd(p) where p = sumlen lst (0,0)
Branching Recursion

“Divide & Conquer” strategy

- split a problem into two or more sub-problems; solve each sub-problem recursively; then combine the sub-results to obtain the final answer

Classic examples

- binary tree traversal
- sorting a list of numbers
Branching Recursion

Binary Tree

Preorder Traversal: [1, 2, 3, 4, 5, 6, 7]
Inorder Traversal: [3, 2, 4, 1, 6, 5, 7]
Postorder Traversal: [3, 4, 2, 6, 7, 5, 1]
Recursive Binary Tree Traversal Algorithms

data BinTree a = Leaf a | Root a (BinTree a) (BinTree a)

preorder :: BinTree a -> [a]
preorder (Leaf v) = [v]
preorder (Root v l r) = [v] ++ preorder l ++ preorder r

inorder :: BinTree a -> [a]
inorder (Leaf v) = [v]
inorder (Root v l r) = inorder l ++ [v] ++ inorder r

postorder :: BinTree a -> [a]
postorder (Leaf v) = [v]
postorder (Root v l r) = postorder l ++ postorder r ++ [v]
Simple Recursive Sorting

- Given a list of values of type `a` on which there is an ordering relation defined, permute the elements of the list so that they are ordered in either ascending or descending order.

- Example: Given the input list `[16, -99, 25, 71, 9, 3, 28]`, sort it into ascending order.

  - Step 1: select the first element of the list as a “pivot”
  - Step 2: partition the list into two sublists “left” and “right” where left = `[x | x <= pivot]` and right = `[y | y > pivot]`
  - Step 3: Recursively sort the left sublist and prepend that result to the singleton list `[pivot]`, and recursively sort the right sublist and append the result to the left++pivot list.
Recursive Sorting Example

\[
\begin{align*}
\text{sort} & \ [16, \ -99, \ 25, \ 71, \ 9, \ 3, \ 28] \\
\text{sort} & \ [-99, 9, 3] ++ [16] ++ \text{sort} \ [25, 71, 28] \\
(\text{sort} \ [] ++ [-99] ++ [9, 3]) & ++ [16] ++ \text{sort} \ [25, 71, 28] \\
(\text{sort} \ [] ++ [-99] ++ \text{sort} \ [9, 3]) & ++ [16] ++ \text{sort} \ [25, 71, 28] \\
(\text{sort} \ [] ++ [-99] ++ (\text{sort} \ [3] ++ [9] ++ \text{sort} \ [])) & ++ [16] ++ \text{sort} \ [25, 71, 28] \\
[99, \ 3, \ 9] & ++ [16] ++ \text{sort} \ [25, 71, 28] \\
[-99, \ 3, \ 9] & ++ [16] ++ (\text{sort} \ [] ++ [25] ++ \text{sort} \ [28, \ 71]) \\
[-99, \ 3, \ 9] & ++ [16] ++ (\text{sort} \ [] ++ [25] ++ (\text{sort} \ [] ++ [28] ++ \text{sort} \ [71])) \\
[-99, \ 3, \ 9] & ++ [16] ++ [25, \ 28, \ 71] \\
[-99, \ 3, \ 9, \ 16, \ 25, \ 28, \ 71]
\end{align*}
\]
Polymorphic recursive sorting function that sorts a list of elements of type `a`, where `a` is required to be ordered (i.e., has the relational operators `==`, `<`, `>`, `<=` and `>=` defined)

```haskell
sort :: (Ord a) => [a] -> [a]

sort [] = [] -- base case
sort (x:[]) = [x] -- singleton list
sort (pivot:rest) = sort left ++ [pivot] ++ sort right
    where
        left = [x | x <- rest, x <= pivot]
        right = [y | y <- rest, y > pivot]
```