**Accumulative Recursion**

- We can sometimes get a more efficient solution by giving the function one extra argument, the *accumulator*, which is used to gather the final result.
- We will need to use an extra function.
- In the case of the `dots` function, the stack recursive definition is actually more efficient.

```haskell
dots n = dots' n ""
dots' 0 acc = acc
dots' n acc = dots' (n-1) (acc ++ ".")
```

**Accumulative Recursion...**

```haskell
dots n = dots' n ""
dots' 0 acc = acc
dots' n acc = dots' (n-1) (acc ++ ".")
```

```haskell
dots 3 ⇒
dots' 3 "" ⇒
dots' 2 ("" ++ ".") ⇒ dots' 2 (".") ⇒
dots' 1 ("." ++ ".") ⇒ dots' 1 ("..") ⇒
dots' 0 (".." ++ ".") ⇒ dots' 0 ("...") ⇒
"...
```

**Stack Recursion**

- The `dots n` function returns a string consisting of `n` dots.
- The dots are “stacked” until we reach the terminating arm of the recursion. $O(n)$ items are stored on the stack.

```haskell
dots 0 = ""
dots n = "." ++ dots (n-1)
```

```haskell
dots 3 ⇒ "." ++ dots 2 ⇒
"." ++ ("." ++ dots 1) ⇒
"." ++ ("." ++ ("." ++ ("." ++ dots 0))) ⇒
"." ++ ("." ++ ("." ++ ("." ++ ")))) ⇒
"." ++ ("." ++ ")") ⇒
"." ++ ")" ⇒ "...
```
Stack Recursion Over Lists

- Stack recursive functions all look very much alike.
- All we need to do is to fill in the template below with the appropriate values and functions.
- **do** is the operation we want to apply to every element of the list.
- **combine** is the operation we want to use to combine the value computed from the head of the list, with the value produced from the tail.

---

Template:  

\[
\begin{align*}
f \ [ \ ] &= \text{final_val} \\
f \ (x:xs) &= \text{combine} \ (\text{do} \ x) \ (f \ xs)
\end{align*}
\]

---

\[
\begin{align*}
f \ [ \ ] &= \text{final_val} \\
f \ (x:xs) &= \text{combine} \ (\text{do} \ x) \ (f \ xs)
\end{align*}
\]

---

`sumlist :: [Int] -> Int`

`sumlist [] = 0`

`sumlist (x:xs) = x + sumlist xs`

**final_val**=0; **do** x = x; **combine**="+

---

`double :: [Int] -> [Int]`

`double [] = []`

`double (x:xs) = 2*x : double xs`

**final_val**=[]; **do** x = 2*x; **combine**=":

---

Stack vs. Accumulative Recursion

- Notice how with stack recursion we’re building the result on the way back up through the layers of recursion.
- This means that for each recursive call many arguments have to be “stacked”, until they can be used on the way back up.
- With accumulative recursion we’re instead building the result on the way down.
- Once we’re at the bottom of the recursion (when the base case has been reached) the result is ready and only needs to be passed up through the layers of recursion.
main xs = aux xs init_val

aux [] acc = acc
aux (x:xs) acc = aux xs (combine do x acc)

Accumulative Recursion Over Lists

• main calls aux, the function that does the actual work. main
  passes along init_val, the value used to initiate the
  accumulator.
• do is the operation we want to apply to every element of the
  list.
• combine is the operation we want to use to combine the
  value computed from the head of the list with the
  accumulator. Template:

    main xs = aux xs init_val
    aux [] acc = acc
    aux (x:xs) acc = aux xs (combine do x acc)

Slide 7–8

Example:
sumlist:
sumlist xs = sumlist' xs 0
sumlist' [] acc = acc
sumlist' (x:xs) acc = sumlist' xs (x + acc)

The reverse Function

“The reverse of an empty list is the empty list. The reverse of a
list (x:xs) is the reverse of xs followed by x.”

([ ] ++ [2]) ++ [1] ⇒ [2] ++ [1] ⇒ [2,1]

In Gofer:

reverse :: [Int] -> [Int]
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
The reverse Function...

- We can devise a more efficient solution by using accumulative recursion.
- At each step we tack the first element of the remaining list on to the beginning of the accumulator.

\[
\begin{align*}
\text{reverse} [1,2] & \Rightarrow \\
\text{reverse'} [1,2] [\emptyset] & \Rightarrow \\
\text{reverse'} [2] (1[:]) & \Rightarrow \\
\text{reverse'} [2 : [1]] & \Rightarrow [2,1]
\end{align*}
\]

Examples:

\[
\begin{align*}
\text{reverse} [1,2,3,4] & \Rightarrow \\
\text{reverse} [2,3,4] & \Rightarrow \\
(\text{reverse} [3,4] & \Rightarrow \\
(\text{reverse} [4] & \Rightarrow \\
(\text{reverse} [3,4] & \Rightarrow [4,3,2] \Rightarrow [4,3,2,1]
\end{align*}
\]

- Each list append A ++ B takes \( O(\text{length } A) \) time.
- There are \( O(n) \) applications of \text{reverse}. Each application of \text{rev} invokes \( \text{append} \) which is an \( O(1) \) operation. Total time = \( O(n) \).

In Gofer:

\[
\begin{align*}
\text{reverse} \; \text{xs} & = \text{rev} \; \text{xs} \; \emptyset \\
\text{rev} \; \emptyset \; \text{acc} & = \text{acc} \\
\text{rev} \; (x:xs) \; \text{acc} & = \text{rev} \; \text{xs} \; (x:acc)
\end{align*}
\]
Summary

- Accumulative recursion uses an extra parameter in which we collect new information as we go deeper into the recursion. The computed value is returned unchanged back up through the layers of recursion.
- Stack recursion performs much of the work on the way back up through the layers of recursion.
- Accumulative recursion is often more efficient than stack recursion.

The Offside Rule

- When does one function definition end and the next one begin?
  
  \[
  \text{square } x = x \times x + 2
  \]
  
  \[
  \text{cube } x = \ldots
  \]
  
- Textual layout determines when definitions begin and end.

Homework

- \text{occurs} \ x \ \text{xs} \ \text{returns the number of times the item} \ x \ \text{occurs in the list} \ \text{xs}.
  
  1. Write a stack recursive definition of \text{occurs}.
  2. Write an accumulative recursive definition of \text{occurs}.
  3. Try the two definitions with a large list as input. How many cells/reductions do they use?

  Template: ______________

  \text{occurs :: Int} \rightarrow \text{[Int]} \rightarrow \text{Int}
  
  ? \text{occurs} \ 1 \ [3,1,4,5,1,1,2,1]
  
  3