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. \( \mathcal{O}(n) \) items are stored on the stack.

\[
dots 0 = \\
dots n = \cdot + \dots (n-1)
\]

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.

\[
dots n = \text{dots'} n \\
dots n = \text{dots'} n \\
dots 0 \text{ acc} = \text{acc} \\
dots n \text{ acc} = \text{dots'} (n-1) (\text{acc} + \cdot)
\]
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.

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.

\textit{do} is the operation we want to apply to every element of the list.

\textit{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{\texttt{final\_val}} \\
f \, (x:xs) &= \text{\texttt{combine}} \, (\text{\texttt{do}} \, x) \, (f \, xs)
\end{align*}

Stack Recursion Over Lists...

\begin{verbatim}
final_val=0; do x = x; combine="+"

f [ ] = final_val
f (x:xs) = combine (do x) (f xs)
sumlist :: [Int] -> Int
sumlist [] = 0
sumlist (x:xs) = x + sumlist xs
double :: [Int] -> [Int]
double [] = []
double (x:xs) = 2*x : double xs
\end{verbatim}
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:

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

**Example** `sumlist`:

```haskell
sumlist xs = sumlist' xs 0
sumlist' [] acc = acc
sumlist' (x:xs) acc = sumlist' xs (x + acc)
init_val=0; do x = x;
combine="+
```

Accumulative Recursion Over Lists...

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

**Example** `maxlist`:

```haskell
maxlist [] = error("...")
maxlist (x:xs) = maxlist' xs x

maxlist' [] acc = acc
maxlist' (x:xs) acc = maxlist' xs (max x a)
init_val= head xs; do x = x;
combine="max"
```

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."

**Examples:**

```haskell
reverse [1,2] ⇒
reverse [2] ++ [1] ⇒
(reverse [] ++ [2]) ++ [1] ⇒
([], ++ [2]) ++ [1] ⇒
[2] ++ [1] ⇒ [2,1]
```

**In Gofer:**

```haskell
reverse :: [Int] → [Int]
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```
The reverse Function...

reverse [1, 2, 3, 4] ⇒
  reverse [2, 3, 4] ++ [1] ⇒
    (reverse [3, 4] ++ [2]) ++ [1] ⇒
      ((reverse [4] ++ [3]) ++ [2]) ++ [1] ⇒
        (((reverse [4] ++ [3]) ++ [2]) ++ [1] ⇒
          (((reverse [4] ++ [3]) ++ [2]) ++ [1] ⇒
            ([4, 3, 2] ++ [1] ⇒
              [4, 3, 2, 1]

Each list append A ++ B takes $O(\text{length } A)$ time.

There are $O(n)$ applications of reverse, where $n$ is the length of the list. Each application invokes append on a list of length $O(n)$. Total time = $O(n^2)$.

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.

Examples:

reverse [1, 2] ⇒
  reverse’ [1, 2] [] ⇒
    reverse’ [2] (1:[]) ⇒
      reverse’ [] (2:[]) ⇒ [2, 1]

In Gofer:

```
reverse xs = rev xs []
rev [] acc = acc
rev (x:xs) acc = rev xs (x:acc)
```

There are $O(n)$ applications of reverse. Each application of rev invokes : which is an $O(1)$ operation. Total time = $O(n)$.
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.

The Offside Rule...

The first character after the "=" opens up a box which holds the right hand side of the equation:

\[\text{square } x = \underbrace{x \times x}_{+2}\]

Any character to the left of the line closes the box and starts a new definition:

\[\text{square } x = \underbrace{x \times x}_{+2}\]
\[\text{cube } x = \ldots\]

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.

Homework

occurs \( x \) \( \text{xs} \) returns the number of times the item \( x \) 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:

\[
\text{occurs :: } \text{Int} \rightarrow \text{[Int]} \rightarrow \text{Int}
\]

Examples:

? \( \text{occurs} 1 \) \[3,1,4,5,1,1,2,1]\n3