University of Auckland

# 415.360

Functional Programming

Christian Collberg

March 11, 1996

# Recursive Functions



## Stack Recursion I

- The `dots n` function returns a string consisting of `n` dots.
- Note that the dots are "stacked", until we reach the terminating arm of the recursion. $\mathcal{O}(n)$ items must be stored on the stack.

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

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



## Accumulative Recursion I

- 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 = dots' n ""

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

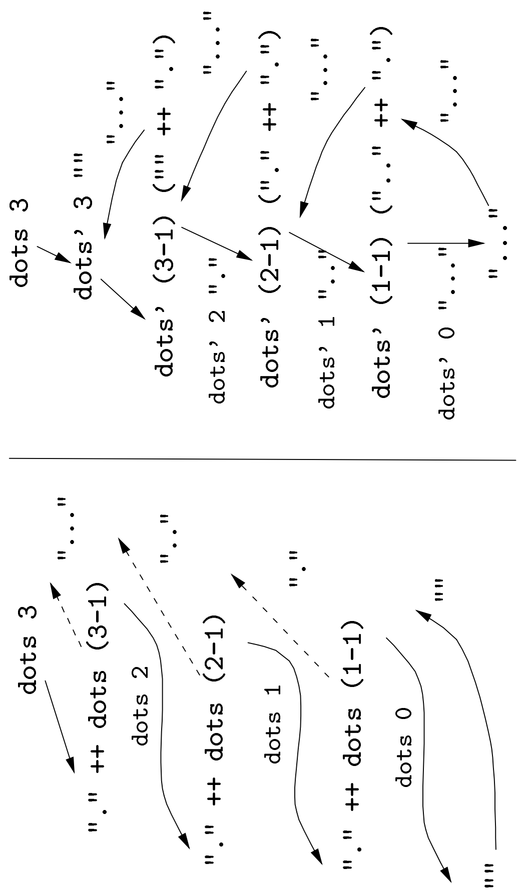


## Accumulative Recursion II

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

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

## Stack vs. Accumulative Recursion I




**Slide 5–4**

## Stack vs. Accumulative Recursion II

- 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.

**Slide 5–5**

## Stack Recursion Over Lists I

- 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:

```
f [ ]    = final_val
f (x:xs) = combine (do x) (f xs)
```

**Slide 5–6**

## Stack Recursion Over Lists II

```
f [ ]    = final_val
f (x:xs) = combine (do x) (f xs)
```

Example `sumlist`:

```
sumlist ::  [Int] -> Int
sumlist [] = 0
sumlist (x:xs) = x + sumlist xs

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

Example `double`:

```
double ::  [Int] -> [Int]
double [] = []
double (x:xs) = 2*x :  double xs

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

**Slide 5–7**

## Accumulative Rec. Over Lists I

- `main` calls `aux`, the function that does the actual work. `main` passes along $\boxed{\texttt{init\_val}}$, the value used to initiate the accumulator.
- $\boxed{\texttt{do}}$ is the operation we want to apply to every element of the list.
- $\boxed{\texttt{combine}}$ 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 5–8**

## Accumulative Rec. Over Lists II

```
main xs = aux xs [init_val]

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

Example `sumlist`:

```
sumlist xs = sumlist' xs 0

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

init_val=0; do x = x;
combine="+"
```

**Slide 5–9**

## Accumulative Rec. Over Lists III

```
main xs = aux xs [init_val]

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

Example `maxlist`:

```
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"
```

**Slide 5–10**

## The `reverse` Function I

In English:

"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:

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

In Gofer:

```
reverse ::  [Int] -> [Int]
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

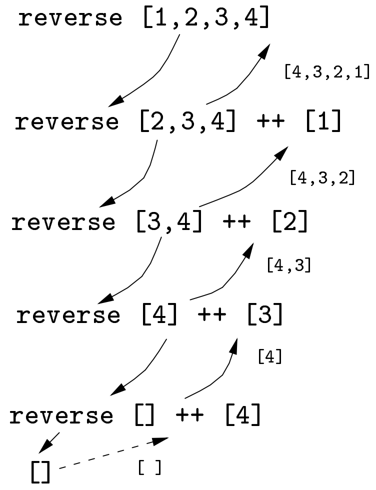
**Slide 5–11**

## The `reverse` Function II

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

**Slide 5–12**

## The `reverse` Function III




- Each list append `A ++ B` takes $\mathcal{O}(\texttt{length}\ A)$ time.
- There are $\mathcal{O}(n)$ applications of `reverse`, where $n$ is the length of the list. Each application invokes `append` on a list of length $\mathcal{O}(n)$. Total time $= \mathcal{O}(n^2)$.

**Slide 5–13**

## The `reverse` Function IV

- 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:

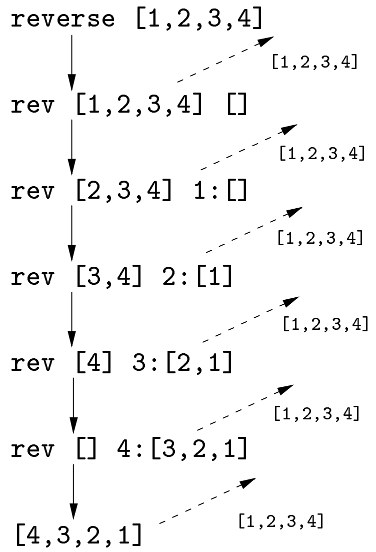
```
reverse [1,2] ⇒
   reverse' [1,2] [] ⇒
      reverse' [2] (1:[]) ⇒
         reverse' [] (2:[1]) ⇒ [2,1]
```

In Gofer:

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

**Slide 5–14**

## The `reverse` Function V




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

**Slide 5–15**

## The Offside Rule

- When does one function definition end and the next one begin?

```
square x =  x * x
            +2
cube x = ···
```

- Textual layout determines when definitions begin and end.
- The first character after the "=" opens up a $\boxed{\text{box}}$ which holds the right hand side of the equation:

```
square x = | x * x
           | +2
```

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

```
square x = | x * x
           | +2

cube x = ...
```

**Slide 5–16**

## Summary

- B&W reading list:
  1. Recursion: 104–105, 109–110. (Ignore induction for now.)
- 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.

**Slide 5–17**

## Homework I

- `occurs x xs` returns the number of times the item `x` occurs in the list `xs`.

1. Write a stack recursive definition of `occurs`.
2. Write an accumulative recursive definition of `occurs`.
3. Try the two definitions with a large list as input. How many cells/reductions do they use?

Template:

```
occurs ::  Int -> [Int] -> Int
```

Examples:

```
?  occurs 1 [3,1,4,5,1,1,2,1]
   3
```

**Slide 5–18**