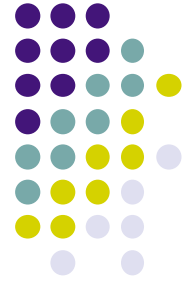


Summary and a bigger example



- We summarize this lesson in a few sentences
 - A recursive function is equivalent to a loop if it is **tail recursive**
 - To write functions in this way, we need to find an **accumulator**
 - We find the accumulator starting from an **invariant** using the **principle of communicating vases**
 - This is called **invariant programming** and it is the only reasonable way to program loops
 - Invariant programming is useful in **all programming paradigms**
- Now let's tackle a bigger example!

A bigger example: calculating X^N



- Let's use invariant programming to define a function {Pow X N} that calculates X^N ($N \geq 0$)
- Let's start with a naive definition of x^n :
$$x^0 = 1$$
$$x^n = x * x^{n-1} \text{ when } n > 0$$
- This gives a first program for {Pow X N} :

```
fun {Pow1 X N}
  if N==0 then 1
  else X*{Pow1 X N-1} end
end
```
- This function is highly inefficient in both time and space! **Why?** (there are two reasons)

Using a better definition of X^N



- Here is another definition of x^n :

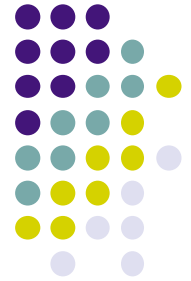
$$x^0 = 1$$

$$x^n = x * x^{n-1} \text{ when } n > 0 \text{ and } n \text{ is odd}$$

$$x^n = y^2 \text{ when } n > 0 \text{ and } n \text{ is even and } y = x^{n/2}$$

- This definition uses many fewer multiplications than the naive definition
 - And just like with the naive definition, we can use this definition to write a program
- Both definitions are also **specifications**
 - They are **purely mathematical** (no program code)

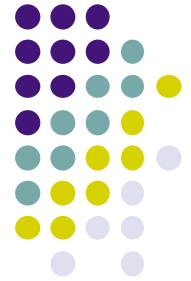
Second program for X^N



```
fun {Pow2 X N}  
  if N==0 then 1  
  elseif N mod 2 == 1 then  
    X*{Pow2 X (N-1)}  
  else Y in  
    Y={Pow2 X (N div 2)}  
    Y*Y  
  end  
end
```

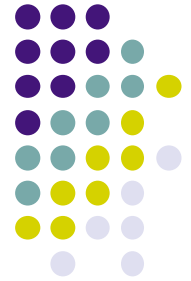
This definition is better than the first, but it is still not tail recursive!

Calculating X^N with invariant programming



- We can do better than Pow2
 - We can write a tail-recursive program: a true loop
- We need an invariant
 - The invariant is the key to a good program
 - One part of the invariant will accumulate the result and another part of the invariant will disappear
 - What can we accumulate?

Reasoning on the invariant



- Here is an invariant: *(x and n constant; y, i, and a vary)*
 $x^n = y^i * a$
- We represent this invariant compactly as a triple:
 (y, i, a)
- Initially: $(y, i, a) = (x, n, 1)$
- Let us decrease i while keeping the invariant true
- There are two ways to decrease i :
 - $(y, i, a) \Rightarrow (y*y, i/2, a)$ (when i is even)
 - $(y, i, a) \Rightarrow (y, i-1, y*a)$ (when i is odd)
- When $i=0$ then the answer is a

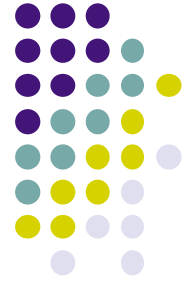
Third program for X^N



```
fun {Pow3 X N}  
  fun {PowLoop Y I A}  
    if I==0 then A  
    elseif I mod 2 == 0 then  
      {PowLoop Y*Y (I div 2) A}  
    else {PowLoop Y (I-1) Y*A} end  
  end  
in  
  {PowLoop X N 1}  
end
```

This program is a true loop
(it is tail-recursive) and it
uses very few multiplications

Invariants and goals



- Changing one part of the invariant forces the rest to change as well, because **the invariant must remain true**
 - The invariant's truth *drives* the program forward
- Programming a loop means finding a good invariant
 - Once a good invariant is found, coding is easy
 - Learn to think in terms of invariants!
- Using invariants is a form of **goal-oriented programming**
 - We will see another example of goal-oriented programming when we program with trees in lesson 5