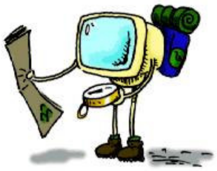
Programming Languages

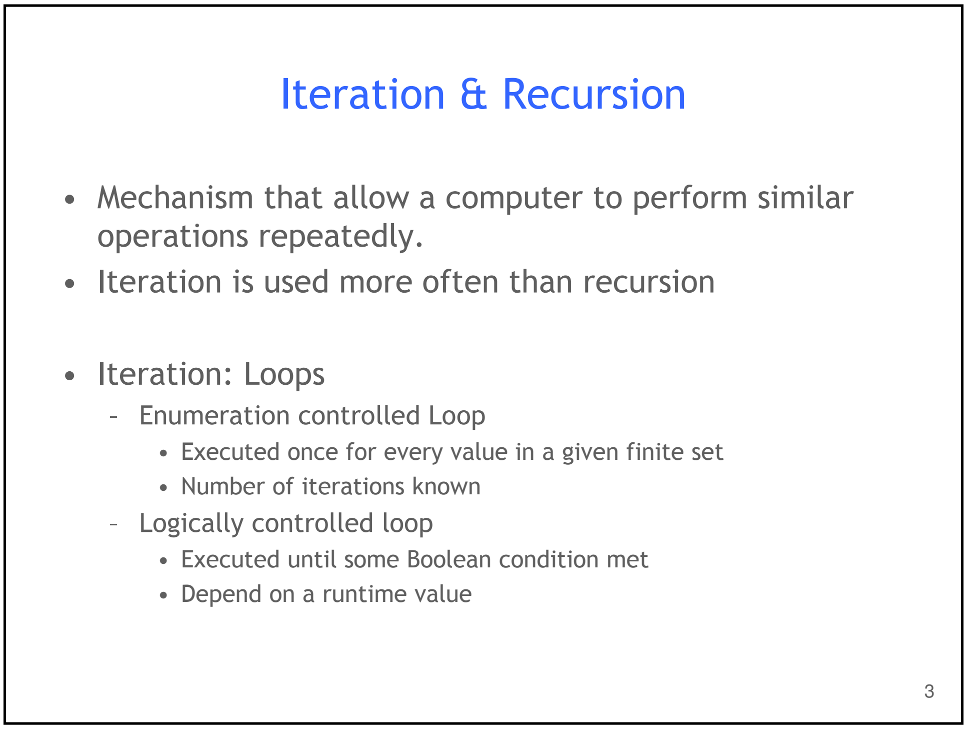
Tevfik Koşar

Lecture - XII
February 23rd, 2006

Roadmap

- Iteration
  - Loops
  - Iterator Objects
- Recursion
  - Tail Recursion
- Evaluation Order

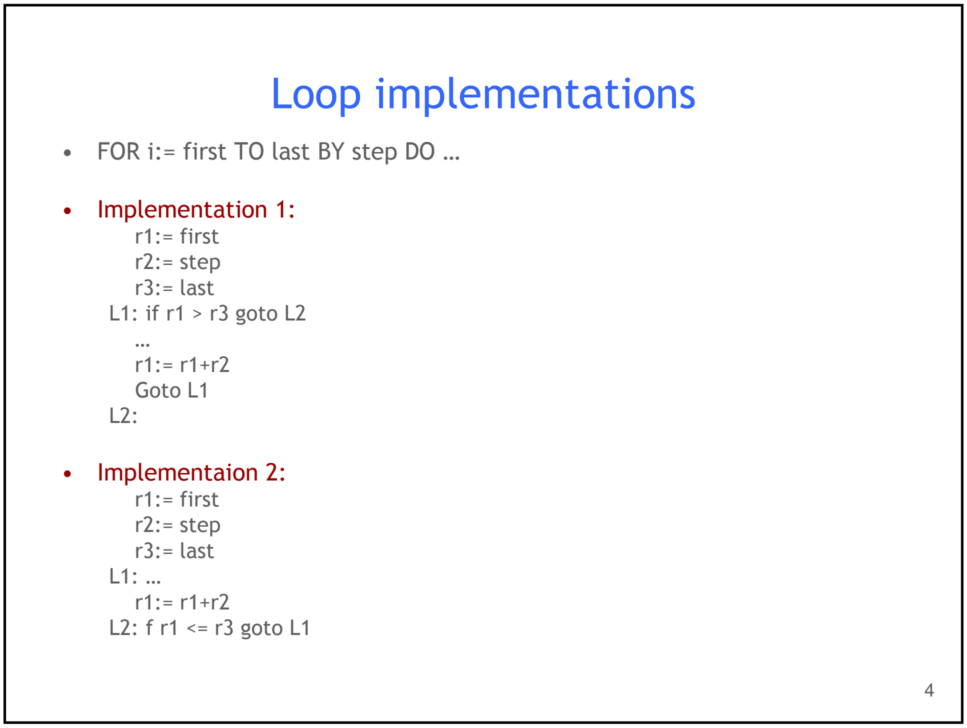




Iteration & Recursion

- Mechanism that allow a computer to perform similar operations repeatedly.
- Iteration is used more often than recursion

- Iteration: Loops
  - Enumeration controlled Loop
    - Executed once for every value in a given finite set
    - Number of iterations known
  - Logically controlled loop
    - Executed until some Boolean condition met
    - Depend on a runtime value



Loop implementations

- FOR i:= first TO last BY step DO …

- Implementation 1:

```
   r1:= first
   r2:= step
   r3:= last
L1: if r1 > r3 goto L2
   …
   r1:= r1+r2
   Goto L1
L2:
```

- Implementaion 2:

```
   r1:= first
   r2:= step
   r3:= last
L1: …
   r1:= r1+r2
L2: f r1 <= r3 goto L1
```

Iterator Objects

- Instead of a simple arithmetic sequence, they allow us to iterate on any well-defined set.

- eg. pointers

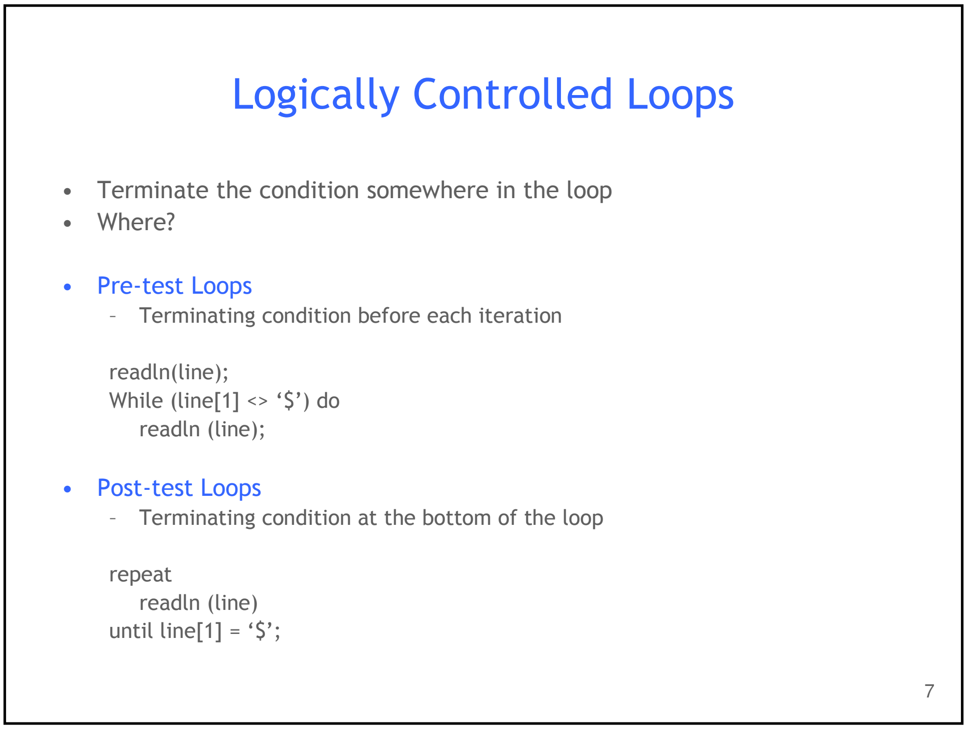
```
For (Iterator<Integer> it = myTree.iterator(); it.hasNext();){
      Integer I = it.next();
      System.out.println(i);
}
```

Iterating without Iterators

```
tree_node *my_tree;
tree_iter TI;
…

for (ti_create(mytree, &ti); !ti_done(ti)){
   tree_node *n = ti_val(ti);
   …
}

ti_delete(&ti);
```



Logically Controlled Loops

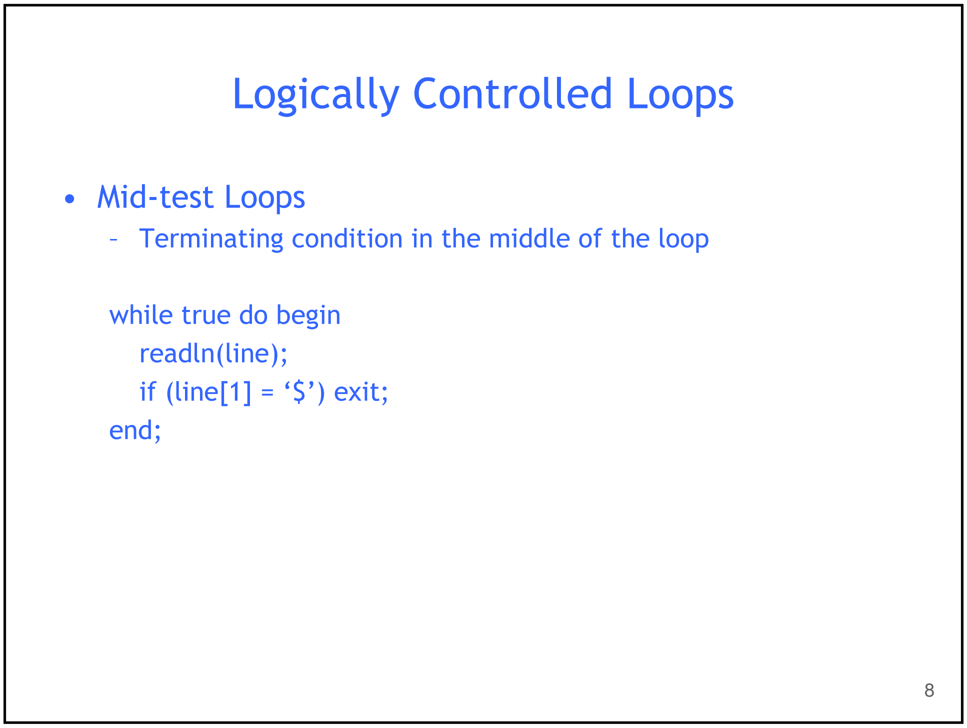
- Terminate the condition somewhere in the loop
- Where?

- Pre-test Loops
  - Terminating condition before each iteration

```
readln(line);
While (line[1] <> ‘$’) do
    readln (line);
```

- Post-test Loops
  - Terminating condition at the bottom of the loop

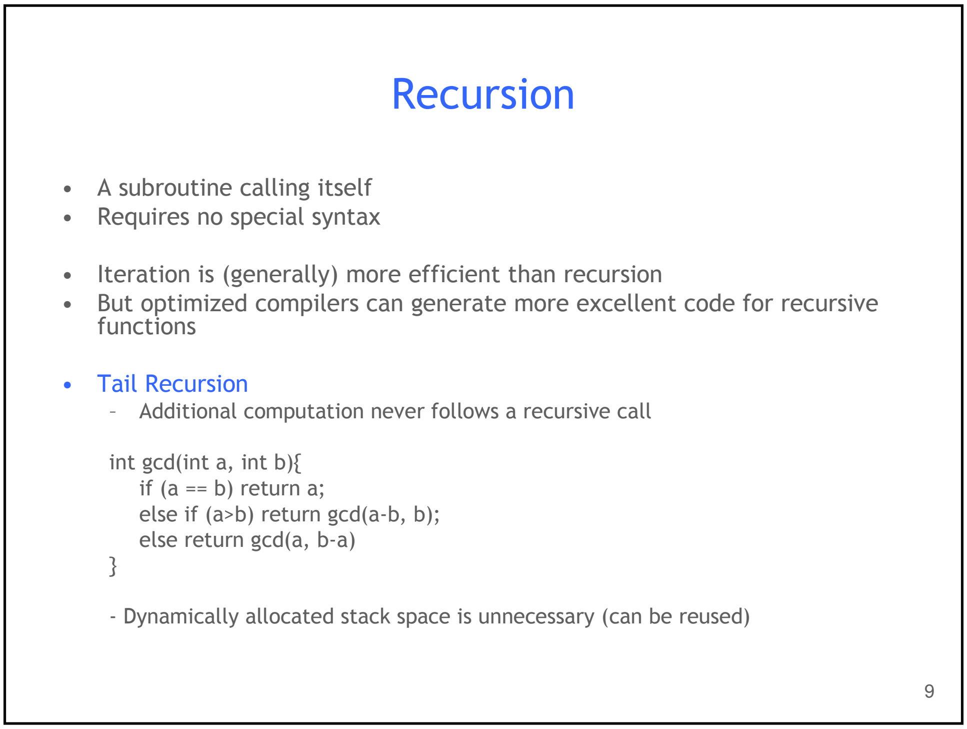
```
repeat
    readln (line)
until line[1] = ‘$’;
```



Logically Controlled Loops

- Mid-test Loops
  - Terminating condition in the middle of the loop

```
while true do begin
    readln(line);
    if (line[1] = ‘$’) exit;
end;
```



Recursion

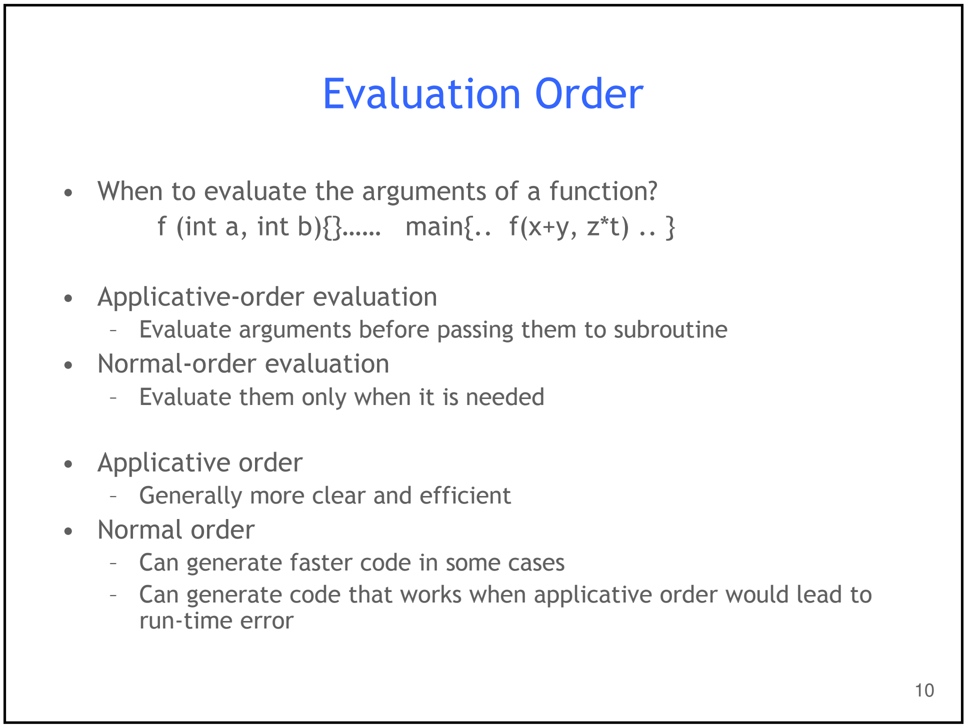
- A subroutine calling itself
- Requires no special syntax

- Iteration is (generally) more efficient than recursion
- But optimized compilers can generate more excellent code for recursive functions

- Tail Recursion
  - Additional computation never follows a recursive call

```
int gcd(int a, int b){
    if (a == b) return a;
    else if (a>b) return gcd(a-b, b);
    else return gcd(a, b-a)
}
```

- Dynamically allocated stack space is unnecessary (can be reused)



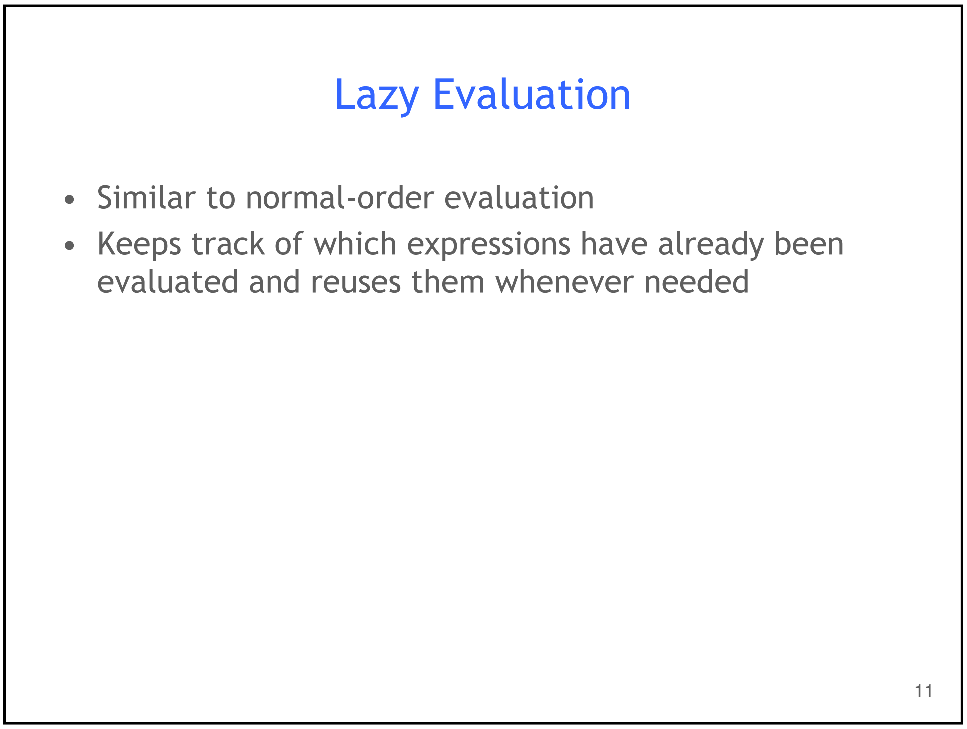
Evaluation Order

- When to evaluate the arguments of a function?

  f (int a, int b){}……   main{.. f(x+y, z*t) .. }

- Applicative-order evaluation
  - Evaluate arguments before passing them to subroutine
- Normal-order evaluation
  - Evaluate them only when it is needed

- Applicative order
  - Generally more clear and efficient
- Normal order
  - Can generate faster code in some cases
  - Can generate code that works when applicative order would lead to run-time error



Lazy Evaluation

- Similar to normal-order evaluation
- Keeps track of which expressions have already been evaluated and reuses them whenever needed