

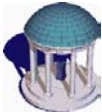
The University of North Carolina at Chapel Hill

COMP 144 Programming Language Concepts
Spring 2003

Iteration and Recursion

Stotts, Hernandez-Campos

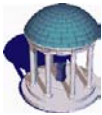
1



Control Flow Mechanisms

- Sequencing
 - Textual order, Precedence in Expression
- Selection
- Iteration
- Procedural abstraction
- Recursion
- Concurrency
- Nondeterminacy

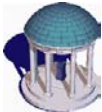
2



Iteration and Recursion

- These two control flow mechanisms allow a computer to perform the same set of operations repeatedly
- They make computers useful
 - Go beyond the power of deterministic finite automata
- *Imperative languages* mainly rely on iterations
- *Functional languages* make more use of recursion

3



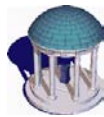
Iteration

- Iteration usually takes the form of *loops*
- There are two principal varieties
 - *Enumeration-controlled* loops
 - » E.g.

```
for (int i = 0; i <= 10; i++) {  
    ...  
}
```
 - *Logically controlled* loops
 - » E.g.

```
int i = 0;  
while (i <= 10) {  
    ...  
    i++;  
}
```

4



Iteration

Enumeration-controlled loops

- Enumeration-controlled loops

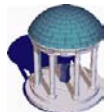
- Index variable
- Step size and bounds
- Body of the loop

- Fortran I, II and IV

```
do 10 i = 1, 10, 2
  ...
10: continue
```

- The value of *i* is tested at the end of the loop
 - » When `continue` is reached
- Implementation is very fast
 - » This statement is very close to assembly code

5



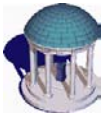
Iteration

Enumeration-controlled loops

- Problems:

- Loop boundaries must be integer
 - » Expressions are not allowed
- The index variable can change within the body of the loop
- Goto statements may jump in and out of the loop
- The value of *i* after the termination of the loop is implementation dependent
- The test of the loop takes place at the end, so the body is executed at least once
 - » Even if the lower bound is larger than the upper bound!

6



Iteration

- Loop should check for empty bounds
- Code generation
 - Optimization

```
for i := first to last by step do
  ...
end
```

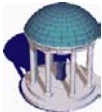
can be translated as

```
r1 := first
r2 := step
r3 := last
L1: if r1 > r3 goto L2
  ... -- loop body; use r1 for i
  r1 := r1 + r2
  goto L1
L2:
```

A slightly better if less straightforward translation is

```
r1 := first
r2 := step
r3 := last
goto L2
L1: ... -- loop body; use r1 for i
  r1 := r1 + r2
L2: if r1 <= r3 goto L1
```

7



Iteration

- Backward loops
 - Previous code assumed a positive step size

```
r1 := first
r2 := step
r3 := max([(last - first + step)/step], 0) -- iteration count
-- NB: this calculation may require several instructions.
-- It is guaranteed to result in a value within the
-- precision of the machine, but we have to be careful
-- to avoid overflow during its calculation.
if r3 <= 0 goto L2
L1: ... -- loop body; use r1 for i
  r1 := r1 + r2
  r3 := r3 - 1
  if r3 > 0 goto L1
  i := r1
L2:
```

8



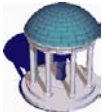
Iteration

Access to Index Outside the Loop

- The value of the index variable at the end of loop is undefined in several languages
 - E.g. Fortran, Pascal
- Compilers can *fix* this, but...
 - Generating slower code

```
r1 := 'a'
r2 := 'z'
if r1 > r2 goto L3    -- Code improver may remove this test,
                      -- since 'a' and 'z' are constants.
L1: ...                -- loop body; use r1 for i
  if r1 = r2 goto L2
  r1 := r1 + 1
  -- NB: Pascal step size is always 1 (or -1 if downto)
  goto L1
L2: i := r1
L3:
```

9

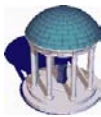


Iteration

Access to Index Outside the Loop

- The value of the index after the loop completes may not be valid
 - E.g.
var c: 'a'..'z';
...
for c:= 'a' to 'z' do begin
 ...
end;
(* what comes after 'z'? *)
- In summary, even the simplest type of loop requires a good design
 - You **will** use language with poorly designed statements!

10



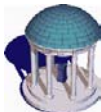
Iteration Iterators

- Iterators generalize enumeration-controlled loops
 - In the previous examples, the iteration was always over the elements of an arithmetic sequence
- Iterators are used to enumerate the elements of any well-defined set
 - *E.g.* In Clu,

```
for i in from_to_by(first, last, step) do
  ...
end
```

- Notice the similarity with Python's `range`

11



Iteration Iterators

- Clu allows any set-like abstract data type to provide an iterator
 - *E.g.* integer iterator

```
from_to_by = iter (from, to, by : int) yields (int)
  i : int := from
  if by > 0 then
    while i <= to do
      yield i
      i += by
    end
  else
    while i >= to do
      yield i
      i += by
    end
  end
end
end from_to_by
```

12

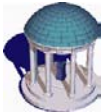


Iteration Iterators

- E.g. Binary tree iterator in Clu

```
bin_tree = cluster is ..., tree_gen, print, ... % export list
node = record [left, right : bin_tree]
rep = variant [some : node, empty : null]
...
tree_gen = iter (k : int) yields (cvt)
  if k = 0 then yield rep$make_empty(nil)
    % just the empty tree
  else
    for i : int in from_to (1, k) do
      for l : bin_tree in tree_gen (i-1) do
        for r : bin_tree in tree_gen (k-i) do
          yield rep$make_some(node${l, r})
        end
      end
    end
  end
end tree_gen
...
end bin_tree
...
for t : bin_tree in bin_tree$tree_gen (n) do
  bin_tree$print (t)
end
```

13



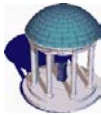
Iterators in Perl

- *Foreach* iterates over the elements of a list

```
@colors = ("red", "green" "#FF8ACC", 'lightred');
foreach $elt (@elements) {
  print $elt, ", ";
}
print "are the colors we have\n";

$reds = 0;
@colors = ("red", "green" "#FF8ACC", 127, 'puce');
foreach (@elements) { # alternate form using $_
  if /red/ { $reds++; } # match on $_ by default
}
print "We have $reds reddish colors \n";
```

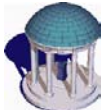
14



Iterations Iterators

- Python will support generators/iterators in the future
 - <http://www.python.org/doc/current/ref/yield.html>
 - <http://python.sourceforge.net/peps/pep-0255.html>
- Iterators can also be based on object-oriented design patterns
 - Java's Iterator interface
 - » <http://java.sun.com/docs/books/tutorial/collections/interfaces/collection.html>
 - Notice that the loop statement is a logically-controlled one, but it is used for an enumeration-controlled task
- Enumeration-controlled loops evolved significantly since FORTRAN's original `for`

15



Iteration Logically-Controlled Loops

- They have fewer semantic subtleties
 - The programmer has to be more explicit
- There are some design options
 - Pre-test
 - » http://java.sun.com/docs/books/jls/second_edition/html/statements_doc.html#237277
 - Post-test
 - » http://java.sun.com/docs/books/jls/second_edition/html/statements_doc.html#6045
 - Midtest
 - » C/C++/Java *idiom*:

```
for (;;) { ... if (condition) break ... }
```

16



Recursion

- Recursion requires no special syntax
- Recursion and logically-controlled iteration are equally powerful
- Example
 - Compute the greatest common divisor
 - It can be defined as a recurrence: for a, b positive integers

$$\text{gcd}(a, b) \equiv \begin{cases} a & \text{if } a = b \\ \text{gcd}(a - b, b) & \text{if } a > b \\ \text{gcd}(a, b - a) & \text{if } a < b \end{cases}$$

17



Recursion

- Implementation using recursion is direct

$$\text{gcd}(a, b) \equiv \begin{cases} a & \text{if } a = b \\ \text{gcd}(a - b, b) & \text{if } a > b \\ \text{gcd}(a, b - a) & \text{if } a < b \end{cases}$$

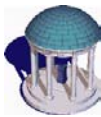
```
int gcd (int a, int b) {  
    /* assume a, b > 0 */  
    if (a == b) return a;  
    else if (a > b) return gcd (a-b, b);  
    else return gcd (a, b-a);  
}
```

Recursion

```
int gcd (int a, int b) {  
    /* assume a, b > 0 */  
    start:  
    if (a == b) return a;  
    else if (a > b) {  
        a = a - b; goto start;  
    } else {  
        b = b - a; goto start;  
    }  
}
```

Iteration

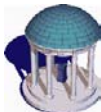
18



Recursion

- We will discuss other control flow issues in the ML lectures
 - Applicative-order evaluation vs. normal-order evaluation
 - Impatient evaluation vs. lazy evaluation

19



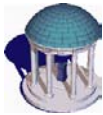
Nondeterminacy

- Nondeterministic constructs make choices between alternatives deliberately unspecified
- This mechanism is specially useful in concurrent programs
 - Message-based concurrent languages
 - We will discuss concurrent programming later in this class

```
process client:
  loop
    toss coin
    if heads, send read request to server
                wait for response
    if tails, send write request to server
                wait for response
```

```
process server:
  loop
    receive read request
    reply with data
  OR
    receive write request
    update data and reply
```

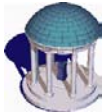
20



Nondeterminacy

- This is a very practical matter
 - Event-driven programming is related to nondeterminacy
 - » See events and listeners in Java
 - » <http://java.sun.com/docs/books/tutorial/uiswing/overview/event.html>
 - Non-blocking IO is related to nondeterminacy
 - » See the latest addition to Java (1.4): Channels and Selectors
 - » <http://java.sun.com/j2se/1.4/docs/guide/nio/index.html>

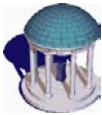
21



Nondeterminacy

- Nondeterminacy is often the behavior of a specific computational architecture (client/server)
- However, some languages have specific constructs to allow expression of it
- Ada “select” statement
- CSP “guarded command” and “alternative”
- Occam

22



Ada Select

```
task body RESOURCE is
  BUSY : BOOLEAN := FALSE;
begin
  loop
    select
      when not BUSY =>
        accept SEIZE do
          BUSY := TRUE;
        end;
      or
        accept RELEASE do
          BUSY := FALSE;
        end;
      or
        terminate;
    end select;
  end loop;
end RESOURCE;
```

23



CSP Alternative Command

CSP was an important notation by Tony Hoare in the early days of experimenting with concurrency

CSP's Alternative Command:

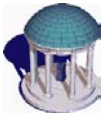
[GC₁ [] GC₂ [] ... [] GC_n]

where GC_i means guarded command i

Cases:

- If all guards fail, the result is an error.
- If one guard succeeds, it executes its command.
- If more than one guard succeeds, one of the commands (whose guard was true) is nondeterministically chosen to execute.
- If none succeed, but not all fail, wait.

24



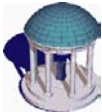
CSP Alternative Examples

```
[
  x >= y -> max := x
[]
  y >= x -> max := y
]

*[ I > 0 -> fact := fact * I; I := I-1 ]

[
  x < y -> z := 27;
[] x > k -> z := 45 * x;
[] k = 14*y -> y := x - z;
[] true -> x := x + 1;
]
```

25



A Taste of Formalism

We often think of programs as computing functions
(denotational semantics)

A program is a mapping... input to output

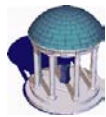
Text of a program is P

“meaning” (function, computation) of the text is [P]

[P] : int -> real *(just one example)*

This function takes an integer and produces a real

26



A Taste of Formalism

Programs as functions:

$$[P] : \text{int} \times \text{real} \rightarrow \text{real}$$

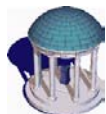
One program that matches this signature

$$\{ (4, 12.4), (2, 6.2), (3, 9.3) \}$$

Domain: $\{ 2, 3, 4 \}$

Range: $\{ 6.2, 9.3, 12.4 \}$

27



A Taste of Formalism

Nondeterminism

We think of programs as computing

relations

*Instead of a given input being mapped to a single output,
we allow it to be mapped to several possible outputs*

$$[P] : \{ (2, 6.2), (2, 4.2), (4, 12.4), (4, 8.4), (6, 18.6) \}$$

Domain: $\{ 2, 4, 6 \}$

Range: $\{ 4.2, 6.2, 8.4, 12.4, 18.6 \}$

28



A Taste of Formalism

Function

2 → 6.2

4 → 12.4

6 → 18.6

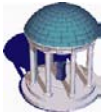
Relation

2 → 6.2
2 → 4.2

4 → 12.4
4 → 8.4

6 → 18.6

29



Reading Assignment

- Scott's chapter 6
 - Section 6.5, except
 - » Combination loops
 - » Generators in Icon
 - » Enumerating without Iterators
 - Section 6.6
 - » First page and a half
 - » We will discuss the rest in the context of ML (instead of Lisp)

30