
CSc 520

Principles of Programming Languages

31 : Control Structures — Iterators

Christian Collberg

collberg+520@gmail.com

Department of Computer Science
University of Arizona

Copyright © 2008 Christian Collberg

Iterators

- FOR-loops are typically used to iterate over some range of enumerable values.
- Iterators are used to iterate over an **abstraction**, such as the elements of a list, the nodes of a tree, the edges of a graph, etc.
- For example,

```
for n := tree_nodes_in_inorder(T) do  
    print n  
end
```

Iterators in Java

Iterators in Java

- In object-oriented languages it is typical to create an **enumeration object** `i` which contains the current state of the iteration:

```
LinkedList<String> L = new LinkedList<String>();  
L.add("Bebe");  
L.add("Wendy");  
L.add("Nelly");
```

```
Iterator<String> i = L.iterator();  
while (i.hasNext()) {  
    String c = i.next();  
    System.out.println(c);  
}
```

- This is not as clean as in languages with built-in support for iterators.

Java 1.5 extended for-loop

- As of Java 1.5, the for-loop has been augmented so you can say

```
LinkedList<String> L = new LinkedList<String>();
L.add("Bebe");
L.add("Wendy");
L.add("Nelly");
for (String c : L)
    System.out.println(c);
```

- However, this is just syntactic sugar for calls to the Iterator class.

Java 1.5 extended for-loop

- You can tell that this is just syntactic sugar by looking at the bytecode the compiler generates (use `javap -c Iter`):

```
29: aload_1
30: invokevirtual #8;          // LinkedList.iterator:
33: astore_2
34: aload_2
35: invokeinterface #9,1;      // java/util/Iterator.hasNext
40: ifeq 63
43: aload_2
44: invokeinterface #10,1;     // java/util/Iterator.next
49: checkcast #11;            // java/lang/String
52: astore_3
53: getstatic #12;             // java/lang/System.out
56: aload_3
57: invokevirtual #13;         // java/io/PrintStream.println
```

Ruby iterators

Blocks

- Let's write a simple Ruby for loop to search through an array looking for a particular value:

```
$flock = ["huey", "dewey", "louie"]
```

```
def isDuck?(name)
  for i in 0...$flock.length
    if $flock[i] == name then
      return true
    end
  end
  return false
end
```

```
puts isDuck?("dewey"), isDuck?("donald")
```

Iterators

- Ruby's *iterators* are an easier way to do this.
- The `Array` class implements a method `find` that iterates through the array.

```
def isDuck?(name)
  $flock.find do |x|
    x == name
  end
end

puts isDuck?("dewey")
puts isDuck?("donald")
```

Yield

- A block is enclosed within `{ }` or `do . . . end`. Arguments to the block (there can be more than one) are given within `| . . . |`.
- A block is passed to a method by giving it after the list of “normal” parameters.
- The method invokes the block by using `yield`.
- `yield` can take an argument which the method passed back to the block.

Yield...

```
def triplets()  
  yield "huey"  
  yield "dewey"  
  yield "louie"  
end
```

```
triplets() {|d| puts d}
```

```
triplets() do |d|  
  puts d  
end
```

Factorial

- Here's the factorial function, as an iterator.

```
def fac(n)
  f = 1
  for i in 1..n
    f *= i
    yield f
  end
end

fac(5) {|f| puts f}
```

Passing arguments

- `yield` can pass more than one value to the block.

```
def fac(n)
  f = 1
  for i in 1..n
    f *= i
    yield i, f
  end
end
```

```
fac(5) do |i, x|
  puts "#{i}! = #{x}"
end
```

Nesting iterators

- Iterators can be nested.

```
fac(3) do |i,x|  
  fac(3) do |j,y|  
    puts "#{i}! * #{j}! = #{x*y}"  
  end  
end
```

Scope

- A local variable which is active when the block is started up, can be accessed (and modified) within the block.

```
def sumfac(n)
  y = 0
  fac(n) do |i, x|
    y = y + x
  end
  return y
end

puts sumfac(5)
```

Implementing Array#find

- We can implement our own `find` method:

```
def find(arr)
  for i in 0..arr.length
    if yield arr[i] then return true end
  end
  return false
end
```

```
puts find($flock) {|x| x=="dewey"}
puts find($flock) {|x| x=="donald"}
```

Array#collect

- collect applies the block to every element of an array, creating a new array. This is similar to Haskell's map.

```
$flock = ["huey", "dewey", "louie"]  
$flock.each {|x| puts x}  
  
puts $flock.collect {|x| x.length}  
puts $flock.collect do |x|  
  "junior woodchuck, General " + x  
end
```

Array#inject

- `inject(init)` is similar to Haskell's `foldl`.
- `inject()` without an argument is like Haskell's `foldl`, i.e. it uses the first element of the array as the starting value.

```
x = $flock.inject("") do |elmt, total|  
  total = elmt + " " + total  
end  
puts x
```

```
x = $flock.inject() do |elmt, total|  
  total = elmt + " " + total  
end  
puts x
```

Icon Generators

Icon Generators

Procedures are really generators; they can return 0, 1, or a sequence of results. There are three cases

fail The procedure fails and generates no value.

return e The procedure generates one value, e.

suspend e The procedure generates the value e, and makes itself ready to possibly generate more values.

```
procedure To(i, j)
  while i <= j do {
    suspend i
    i+:= 1
  }
end
```

Example

```
procedure To(i,j)
  while i <= j do {
    suspend i
    i+:= 1
  }
end

procedure main()
  every k := To(1,3) do
    write(k)
  end
```

simple.icn

```
procedure P()  
    suspend 3  
    suspend 4  
    suspend 5  
end
```

```
procedure main()  
    every k := P() do  
        write(k)  
end
```

simple.icn...

```
> setenv TRACE 100
```

```
> simple
```

```
      :      main()  
simple.icn :      8  | P()  
simple.icn :      2  | P suspended 3  
3  
simple.icn :      9  | P resumed  
simple.icn :      3  | P suspended 4  
4  
simple.icn :      9  | P resumed  
simple.icn :      4  | P suspended 5  
5  
simple.icn :      9  | P resumed  
simple.icn :      5  | P failed  
simple.icn :     10  | main failed
```

Iterators in CLU

CLU-Style Iterators

- Iterators were pioneered by CLU, a (dead) class-based language from MIT.

```
setsum = proc(s:intset) returns(int)
  sum : int := 0
  for e:int in intset$elmts(s) do
    sum := sum + e
  end
  return sum
end setsum
```

CLU-style Iterators...

- Procedure `set sum` computes the sum of the elements in a set of integers.
- `set sum` iterates over an instance of the abstract type `intset` using the `intset$elems` iterator.
- Each time around the loop, `intset$elems` yields a new element, suspends itself, and returns control to the loop body.

CLU-style Iterators...

```
intset = cluster is create, elmts, ...  
  rep = array[int]  
  elmts = iter(s:cvt) yields(int)  
    i : int := rep$low(s)  
    while i <= rep$high(s) do  
      yield (s[i])  
      i = i + 1  
    end  
  end elmts  
end intset
```

CLU-style Iterators...

- A CLU `cluster` is a typed module; a C++ class, but without inheritance.
- CLU makes a clear distinction between the abstract type (the cluster as seen from the outside), and its representation (the cluster from the inside). The **rep** clause defines the relationship between the two.

CLU-style Iterators...

```
elmts = iter(s:cvt) yields(int)
  i : int := rep$low(s)
  while i <= rep$high(s) do
    yield (s[i])
    i = i + 1
  end
end elmts
```

CLU-style Iterators...

- `s:cvt` says that the operation converts its argument from the abstract to the representation type.
- `rep$low` and `rep$high` are the bounds of the array representation.
- `yield` returns the next element of the set, and then suspends the iterator until the next iteration.
- Iterators may be nested and recursive.

CLU-style Iterators...

```
array = cluster [t: type] is ...
    elmts = iter(s:array[t]) yields(t)
        for i:int in int$from_to(
            array[t]$low(a),
            array[t]$high(a)) do
            yield (a[i])
        end
    end elmts
end array
elmts = iter(s:cvt) yields(int)
    for i:int in array$elmts(s) do
        yield (i)
    end
end elmts
```

CLU-style Iterators...

- Iterators may invoke other iterators.
- CLU supports constrained generic clusters (like Ada's generic packages, only better).

CLU Iterators — Example A

- Here's an example of a CLU iterator that generates all the integers in a range:

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

CLU Iterators — Example A...

```
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
```

CLU Iterators — Example B

- Here's an example of a CLU iterator that generates all the binary trees of n nodes.

```
for t: bin_tree in bin_tree$tree_gen(n) do
  bin_tree$print(t)
end
```

CLU Iterators — Example B...

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

Implementing Iterators

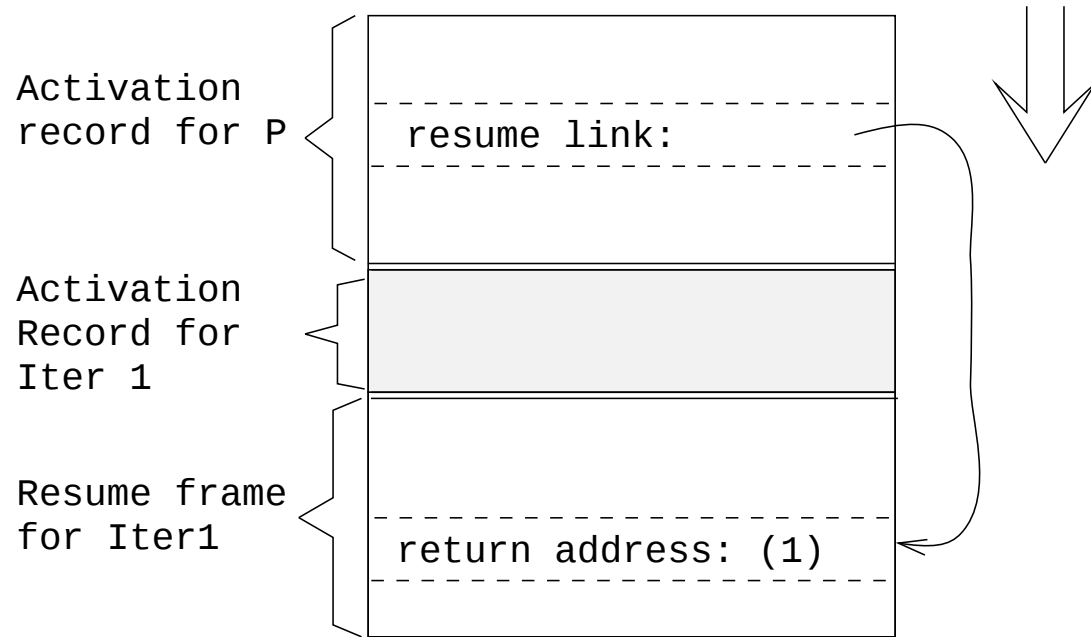
Iterator Implementation

```
Iter1 = iter ( ... )
    ... yield x
    (1) ...
end
end Iter1
P = proc ( ... )
    for i in Iter1(...) do
        S
    end
end P
```

Iterator Implementation

- Calling an iterator is the same as calling a procedure. Arguments are transferred, an activation record is constructed, etc.
- Returning from an iterator is also the same as returning from a procedure call.

Iterator Implementation...



Iterator Implementation...

- When an iterator yields an item, its activation record remains on the stack. A new activation record (called a **resume frame**) is added to the stack.
- The resume frame contains information on how to resume the iterator. The **return address**-entry in the resume frame contains the address in the iterator body where execution should continue when the iterator is resumed.

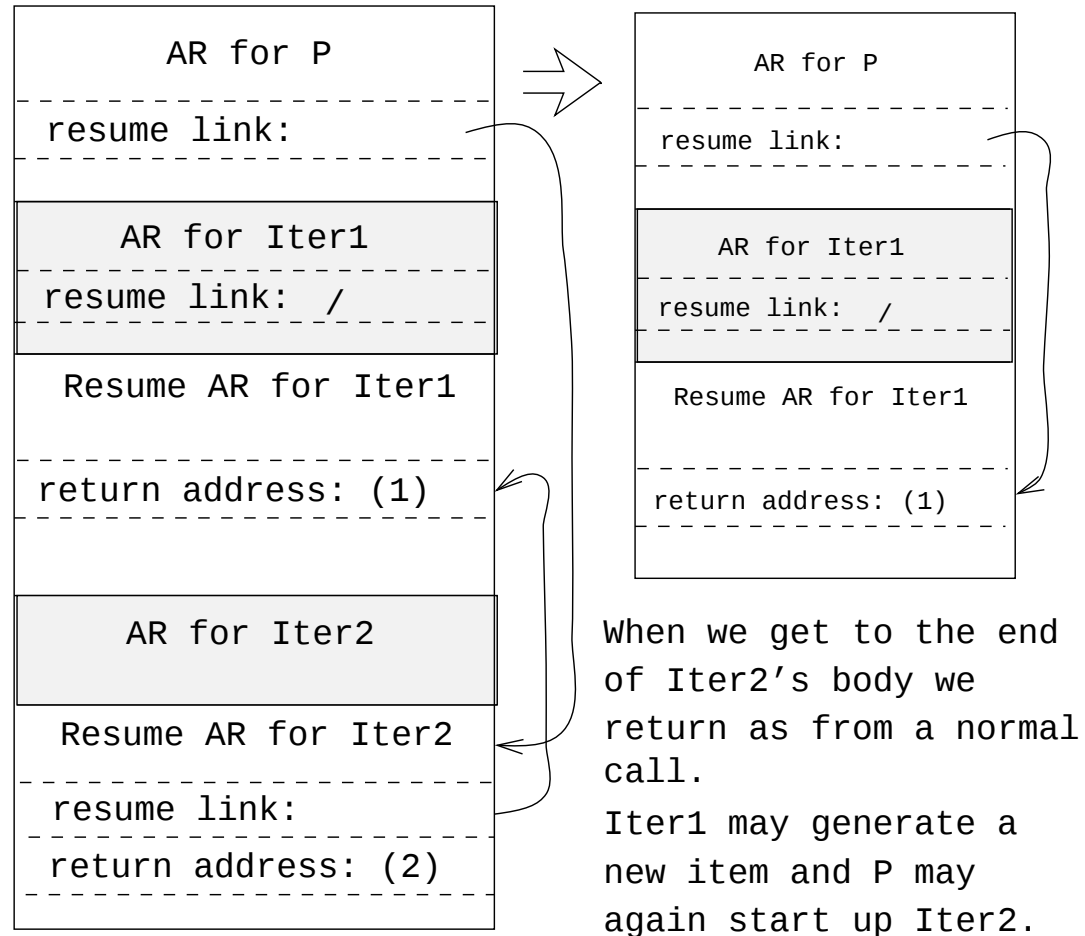
Nested Iterators

```
for i in Iter1(...) do
  for j in Iter2(...) do
    S
  end
end
```

Nested Iterators...

- Since iterators may be nested, a procedure may have several resume-frames on the stack.
- A new resume frame is inserted **first** in the procedure's **iterator chain**.
- At the end of the **for**-loop body we resume the **first** iterator on the iterator chain:
 1. The first resume frame is unlinked.
 2. We jump to the address contained in the removed frame's return address entry.

Nested Iterators...



Simpler Iterator Implementation

```
Iter = iter ( ... )  
    while ... do  
        yield x  
    end  
end  
  
begin  
    for i in Iter(...) do  
        print(i);  
    end  
end
```



Simpler Iterator Implementation...

```
PROCEDURE Iter (  
    Success, Fail : LABEL;  
    VAR Resume : LABEL; VAR Result : T);  
BEGIN  
    WHILE ... DO  
        ResumeLabel:  
        Result := x;  
        Resume := ADDR(ResumeLabel);  
        GOTO Success  
    END;  
    GOTO Fail;  
END
```

Simpler Iterator Implementation...

```
VAR Result : T;
VAR Resume : LABEL;
BEGIN
    Iter(ADDR(SuccesLabel), ADDR(FailLabel),
        Resume, Result);
    SuccessLabel:
    WRITE Result;
    GOTO Resume;
    FailLabel:
END;
```

Summary

Readings and References

1. Read Scott, pp. 278–284, 135CD-136CD.
2. Russell R. Atkinson, Barbara H. Liskov, and Robert W. Scheifler: *Aspects of Implementing CLU*, Proceedings ACM National Conference, pp. 123–129, Dec, 1978.
3. Murer, Omohundro, Szyperski: *Sather Iters: Object-Oriented Iteration Abstraction*:
<ftp://ftp.icsi.berkeley.edu/pub/techreports/1993/tr-93-045.ps.gz>
4. Todd A. Proebsting: *Simple Translation of Goal-Directed Evaluation*, PLDI'97, pp. 1–6. This paper describes an efficient implementation of Icon iterators.

Summary

- Sather (a mini-Eiffel) has adopted an iterator concept similar to CLU's, but tailored to OO languages.
- Iterators function (and can be implemented as) coroutines. Smart compilers should, however, take care to implement “simple” iterators in a more direct way (See the Sather paper).
- Inline expansion of iterators may of course be helpful, but the same caveats as for expansion of procedures apply: code explosion, cache overflow, extra compilation dependencies.