Lecture 4

Building Control Abstractions with Coroutines

iterators, tables, comprehensions, coroutines, lazy iterators, composing iterators

Ras Bodik
with help from Mangpo and Ali

Hack Your Language!

CS164: Introduction to Programming Languages and Compilers, Spring 2013
UC Berkeley
Administrativia

Let us know right away if you don’t have a team

Optional (but recommended) homework

- It will help you with PA1
- See the web page that shows lecture notes
Today

- iterators (the `for` construct)
- comprehensions
- coroutines
- lazy iterators
- composing lazy iterators
- producer/consumer patterns
For loops and other iterators
Iterators

Whenever a language includes collections or allows you to build one, we also want constructs for iterating over them.

**Example:** d3 selections (sets of DOM nodes)

The `each` operator in

```javascript
aSelection.each(aFunction)
```

is an iterator (implemented as a function)
Let’s design a for iterator (behavior)

Desired behavior: say want to iterate from 1 to 10:

```python
for x in iter(10) { print x }
```

Q1: Is iter a keyword in the language? No, a function.


```javascript
function iter(n) {
    def i = 0
    function () {
        if (i < n) { i = i + 1; i }
        else { null } ~for terminates
    }
}
```
Let’s design a for iterator (generality)

Q3: In general, what constructs to permit in __?

```python
for x in __ { print x }
```

A: Any expression that returns an iterator function.

- the syntax of for thus is: `for ID in E { S }`
  ```python
def myIter = iter(10)
```
- these are all legal programs:
  ```python
  for x in myIter { S }
  for x in myIterArray[2] { S }
  for x in myIterFactoryFactory()() { S }  // 😊
  ```
Let’s design a for iterator (scoping)

Q4: What is the scope x?

\[ \text{for } \mathbf{x} \text{ in } E \{ S \} \]

Q5: In what environment should E be evaluated? In particular, should the environment include x?

E should be evaluated in e, the environment of for.

S should be evaluated in e extended with the binding for x.
Implementing the **for** iterator

We are done with the design of behavior (semantics). Now to implementation. We’ll desugar it, of course.

```
for ID in E { S }
```

---

```
{ // a block to introduce new scope
  def $t1 = E               // t1 a temp var
  def ID =$t1()
  while (ID != null) {
    S
    ID =$t1()
  }
}
```
Side note: the block scope

A new scope can be introduced by desugaring, too:

```
{ S } --> ( function(){ S } )()
```

This trick is used in JS programs to restrict symbol visibility, i.e., to implement a simple module construct.
Iterator factory for tables

Assume we are using the table (dict) as array:

def t = {}; t[0] = 1; t[1] = 2
for x in asArray(t) { print x }
def asArray(tbl) {
    def i = 0
    // your exercise: fill in the rest

```python
    function() {
        if t[i] in the array t
            yes : return tbl[i]
        no : return null
    }
```
}
Summary of key concepts

• Iterators and loops are useful for collections and also just to repeat some statement
• We want iterators that work in a modular way, i.e., any library can provide a “iterator factory”
• When desugaring for x in E { S }, S turns into a closure that accesses x as a nonlocal variable
Test yourself

Optional homework, posted on the course web page
(understand surprises behind Python comprehensions)
Hint for the optional homework

Why should a desugar rule not touch the body?

\[
\text{for } id \text{ in } E:
\]

\[
\text{body}
\]
should not modify the body.

If you are the compiler, you want to translate \textit{for} without regard for what’s in the body. That is, the desugar rule should not have to peek into the subtree body. Otherwise there will be many special cases based on what’s in \textit{body}.

A simple, modular compiler desugars \textit{body} recursively
Comprehensions
Comprehensions

A map operation over anything that is iterable.

```
toUpperCase(v) for v in elements(["a","b")
```

```
->
[“A”, “B”]
```

General syntax:

```
[E1 for ID in E2]
```

Can E1, E2 be comprehension expressions?
Comprehensions

Desugaring rule (case specific to this example):

\[
\text{[toUpperCase}(v) \text{ for } v \text{ in elements(list)]}
\]

--->

\[
$1 = []
\]

\[
\text{for } v \text{ in elements(list) } \{ \text{append}($1, \text{toUpperCase}(v)) \} 
\]

\[
$1
\]
Homework: write a general desugar rule

make sure it works work on nested comprehensions

mat = [[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9],
     ]
print([[row[i] for row in mat]
       for i in [0, 1, 2]
     ])

--> [[1, 4, 7], [2, 5, 8], [3, 6, 9]]

"To avoid apprehension when nesting list comprehensions, read from right to left"
Our abstraction stack is growing nicely

comprehensions
for + iterators
if + while
lambda
Lazy iterators
Print all permutations of a list

def permgen(a,n=len(a)) {
    if (n <= 1) {
        print(a)
    } else {
        for i in iter(n) {
            a[n],a[i] = a[i],a[n]
            permgen(a,n-1)
            a[n],a[i] = a[i],a[n]
        }
    }
}

permgen(["a","b","c"])
Now let’s try to wrap permgen in an iterator

We want to be able to write

```python
for p in permIterator(list) {
    print p
    if (condition(p))
        return p  // find first p with some property
}
```

The loop may iterate only over some permutations, so let’s not compute and store all $O(2^n)$ of them in a list. Let’s compute them **lazily**, as needed by the loop.
First attempt at wrapping permgen in iterator

```python
def permIterator(lst) {
    def permgen(a,n=len(a)) {
        if (n = 1) {
            return a  // print(a)
        } else {
            for i in iter(n) {
                a[n],a[i] = a[i],a[n]
                permgen(a,n-1)
                a[n],a[i] = a[i],a[n]
            }
        }
    }
    lambda () { permgen(lst) }  // the iterator
}
```
What is our stumbling block?

The call stack in `for p in permIterator (lst)\{S(p)\}` when `permgen` attempts to pass a permutation to `for`:

```
inside while loop
   iteror
      permgen(n)
      ...
      permgen(1)
```

Why can’t `permgen` pass the permutation to `iterator`?
- it would need to return all the way to top of recursion
- this would make to lose all context
- here, context = value of i for each recursion level
Idea for workarounds?

Rewrite permgen to be iterative.

Unfortunately, trading recursion for a loop will force us maintain the context (a copy of i for each list element). The code will be uglier.

Reverse the master-slave relationship.

At the moment, for is the master. It calls the iterator (slave). Critically, its master decides when to stop iterating, conveniently without having to communicate this decision to the slave. --- Once permgen is the master, we pass to it the body of for as a closure. This body is the slave who decides when to stop iterating (it’s when condition(p) holds). This decision needs to be communicated to the master permgen. So we need implement a termination protocol between master and slave.
We need something like a goto

**Idea:** Jump from permgen to the while loop and back, preserving permgen context on its call stack

Two execution contexts, each with own stack:

```
while call stack
inside while loop
iter-lambda
“call” permgen
```

```
permgen call stack
permgen(n/\lambda)
...
permgen(\theta)
“return” to while
```
Coroutines
Coroutines == “cooperating threads”

Cooperating =

– one thread of control (one PC)
– coroutines themselves decide when control is transferred between them
  • as opposed to an OS scheduler deciding when to preempt the running thread and transfer control (as in timeslicing)
– transfer done with a yield statement

Many flavors of coroutines exist.
Asymmetric Coroutines

Asymmetric: notion of master vs. slave

- symmetric c/r can be implemented on top of symmetric

Benefits of asymmetric coroutines:

- easier to understand for the programmer because from the master the transfer looks like an ordinary call

- easier to implement (you’ll do it in PA2)
Asymmetric Coroutines

Three constructs:

- `coroutine(body)` master creates cortn
- `resume(co, arg)` call into cortn co
- `yield(arg)` return to master

Body is a closure
Example

def co = create_coroutine(
    lambda():{
        print(1)
        yield
        print(2)
        yield
        print(3)
    }  # implicit yield
)

resume(co) --> 1
resume(co) --> 2
resume(co) --> 3
resume(co) --> error (resuming to a terminated coroutine)
Example

def co = create_coroutine(lambda():
    yield 1
    yield 2
    yield 3
)

print(resume(co))  --> 1
print(resume(co))  --> 2
print(resume(co))  --> 3
resume(co)  -->
def permgen(a, n=len(a)) {
    if (n <= 1) {
        yield(a)  /* print(a) */
    } else {
        for i=1 to n {
            a[n],a[i] = a[i],a[n]
            permgen(a,n-1)
            a[n],a[i] = a[i],a[n]
        }
    }
}

def permIterator(lst) {
    def co = coroutine(
        function(l) { permgen(l); null })
    function () { resume(co, lst) }
}
What can we do with coroutines

define control abstractions impossible with functions:

- lazy iterators
- push or pull producer-consumer patterns
- backtracking
- regexes
- exceptions
Iterate over multiple collections simultaneously
Problem: Merge two binary search trees

You are given two binary search trees. Print the “merge” of the trees, traversing each tree only once.

We know how to print values of one tree:

```python
def preorder(node) {
    if (node) {
        preorder(node.left)
        print(node.key)
        preorder(node.right)
    }
}
```

But how do you traverse two trees at once?
Preorder tree iterator

First step: Create a preorder iterator based on c/r.

def preorder(node)
    if (node) {
        preorder(node.left)
        yield(node.key)
        preorder(node.right)
    }
    null
}
def preorder_iterator(tree) {
    def co = coroutine(lambda(t){ preorder(t) })
    lambda () { resume(co, tree) }
}
Now we do “merge sort” over trees

def merge(t1, t2):
    def it1 = preorder_iterator(t1)
    def it2 = preorder_iterator(t2)
    def v1 = it1()
    def v2 = it2()
    while (v1 || v2):
        if (v1 != null and (v2==null or v1<v2)):
            print(v1); v1 = it1()
        else:
            print(v2); v2 = it2()
Exercise for you, part 1

Wrap \text{merge}(t_1,t_2)$ in an iterator so that you can do

\begin{verbatim}
for v in mergeTreeIterator(tree1,tree2) {
    process(v)
}
\end{verbatim}

\begin{verbatim}
function mergeTreeIterator(tree1,tree2) {

}
\end{verbatim}
Exercise, part 2

Write an iterator for merging of three trees

```python
for v in merge3TreeIterator(tr1, tr2, tr3)
```

Build the iterator on top of mergeTreeIterator
Consumer-Producer Pattern
Create a dataflow on streams

Process the values from merge(t1,t2)

We can apply operations:

```python
for v in toUppercaseF(merge(tree1,tree2)) { process(v) }
```

How to create “filters” like toUpperCaseF?
A filter element of the pipeline

def filter(ant):
    def co = coroutine(function() {
        while (True) {
            --resume antecessor to obtain value
            def x=ant()
            -- yield transformed value
            yield(f(x))
        }
    })
    lambda() { resume(co,0) }
}
consumer(filter(filter(filter(producer()))))
How to implement such pipelines

Producer-consumer pattern: often a pipeline structure

producer \rightarrow filter \rightarrow consumer

All we need to say in code is

consumer(filter(producer()))

Producer-driven (push) or consumer-driven (pull)

This decides who initiates resume(). In pull, the consumer resumes to producer who yields datum to consumer.

Each producer, consumer, filter is a coroutine

Who initiates resume is the main coroutine.
In for x in producer, the main coroutine is the for loop.
Summary

Coroutines allow powerful control abstractions
  iterators but also backtracking, which we’ll cover soon

You will implement coroutines in PA2
  we’ll describe the implementation in L5
What you need to know

• Iterators
• Programming with coroutines
• Write push and pull producer-consumer patterns
HW3

• Will prepare you for the project
Glossary
Reading

Required:

- Chapter on coroutines from the Lua textbook

Recommended:

- Python generators are coroutines

Fun:

- More applications of coroutines are in *Revisiting Coroutines*
Acknowledgements

Our course language, including its coroutines, are modeled after Lua, a neat extensible language.

Many examples in this lecture come from Programming in Lua, a great book. Read the 1st edition on the web but consider buying the 2nd edition.

http://www.lua.org/pil/

Coroutine examples are from Revisiting Coroutines.
Backtracking
Problem: Regex matching

We are given a (an abstract syntax tree) of a regex. The goal is to decide if the regex matches a string.

Pattern ("abc" | "de") . "x" can be defined as follows:

\[
patt = \text{seq}(\text{alt}(\text{prim("abc")}, \text{prim("de")}), \text{prim("x")})
\]

which effectively encodes the pattern’s AST.

\text{seq}, \text{alt}, \text{prim} are coroutines.
Regex matching with coroutines

-- matching a string literal (primitive goal)
def prim(str) {
    lambda(S,pos) {
        def len = len(str)
        if (sub(S,pos,pos+len-1)==str) {
            yield(pos+len)
        }
    }
}

-- alternative patterns (disjunction)
def alt(patt1,patt2) {
    lambda(S,pos) { patt1(S,pos); patt2(S,pos) }
}

-- sequence of sub-patterns (conjunction)
def seq(patt1,patt2) {
    lambda(S,pos) {
        def btpoint=coroutine.wrap(function(){ patt1(S,pos) })
        for npos in btpoint { patt2(S,npos) }
    }
}
And now the main match routine

```python
def match(S, patt):
    def m = coroutine.wrap(lambda(): { patt(S, 0) })
    for (pos in m):
        if (pos == len(S)):
            return true
    return false

match("de", alt(prim("abc"), prim("de")))
--> true
```