Hack Your Language!

CSE401 Winter 2016
Introduction to Compiler Construction

Lecture 4: Iterators and Coroutines

Comprehensions
Lazy Iterators
Coroutines
Announcements

Sign up on Piazza if you haven’t

Makeup lecture on Friday 2:30pm-3:50pm
- Same room (here!)
- Will be audio recorded

Section tomorrow

Permanent OHs posted on website
Today

- Comprehensions
- Composing iterators
- Lazy iterators
- Intro to coroutines
Reading

Required:

Chapter on coroutines from the Lua textbook (http://www.lua.org/pil/)

Recommended:

Python generators are coroutines, actually

Fun:

More applications of coroutines are in Revisiting Coroutines
Our abstraction stack is growing nicely

comprehensions
↓ for + iterators
↓ if + while
↓ functions

CORE LANGUAGE

desugaring
Comprehensions
Comprehensions

A map operation over anything that is iterable.

\[
\text{[toUpperCase(v) for v in elements(["a","b"])]} \\
\rightarrow \\
\text{["A","B"]}
\]

General syntax:

\[[E1 \text{ for } ID \text{ in } E2]\]

Can E1, E2 be comprehension expressions?
Where is variable ID visible? In E1, E2 or both?
Comprehensions

Desugaring this example:

```
[toUpperCase(v) for v in elements(list)]
```
Your homework: write a general desugar rule

Must work on nested comprehensions

```
mat = [[1, 2, 3],  // 2D matrix
      [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"
Compositions of iterators
Iterators can be composed

A simple example: convert characters to uppercase

```python
def lst = [“a”, “b”, “c”]
for c in toUpCsIt(asArray(lst)) {
    print c
}
```

Exercise 1. Write `toUpCsIt` function

```python
function toUpCsIt(it) {
    function () {
        def c = it()
        if (c) { toUpperCase(c) }
        else { null }
    }
}
```

(see note on next slide)
Modularity via high-order functions

Instead, use map (functional programming); a list comprehension; or the **pipes-and-filters pattern**. In either case, no code is specific to ToUpperCase:

1. `map(toUpperCase, lst)`
2. `[toUpperCase(c) for c in lst]`
3. `consumer(filter(toUpperCase, producer()))`
Lazy iterators
A motivating example

Find the best first move in Scrabble given some time:

```plaintext
s = ['a', 'f', ...] // 7 letter tiles
for p in permgen(s) {
    for s in subsets(p) {
        if (legalWord(s, wordDict)) {
            // check score of s
            // exit when out if time
        }
    }
}
```
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"])

[a, b, c], 3
[c, b, a], 2
[b, c, a], 1
[c, b, a], 1
"b c a"
"c b a"
Now let’s try to wrap permgen in an iterator

We want to be able to write this code

```python
for p in permIterator(list) {
    if (condition(p))
        print p    // print a subset of permutations
}
```
Don’t need to iterate over all permutations

We may want to print just the first legal word

```python
def s = legalWord(permIterator(ltrs), myDict)
def word = s()
if (word) print word
```

legalWord 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 caller of the permutation iterator
An incorrect attempt at permgen iterator

def permIterator(lst) {
    def permgen(a,n=len(a)) {
        if (n == 1) {
            return a       // was 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]
            }
        }
    }

    function () { 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
iterator
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 force it to lose all context
- context = the value of `i` for each recursion level
Solution and lessons

Rewriting permgen to be resumable

Replacing recursion with a loop forces us to maintain the context (a distinct copy of i for each level of recursion).

The code is significantly harder to write and read.
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-function
“call” permgen
```

```
permgen call stack
permgen(n)
...
permgen(1)
“return” to while
```
Coroutines
Coroutines == cooperating “threads”

Cooperating =

- one thread of control (one Program Counter)
- 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)
  - hence also known as “green threads”
- transfer done with a yield statement

many flavors of coroutines exist

We will cover Lua’s asymmetric coroutines
Asymmetric Coroutines

Asymmetric: notion of master vs. slave

- symmetric coros. can be implemented on top of asymmetric

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:

- `co=create_coroutine(body)` create a coroutine
- `co` is a handle
- `resume(co, arg)` call/resume a coroutine
- `yield(arg)` return to master, who can resume

Body is a closure
Example (no values passed)

```javascript
var co = create_coroutine(
    function(){
        print(1)
        yield
        print(2)
        yield
        print(3)
    }
)
resume(co) -->
resume(co) -->
resume(co) -->
resume(co) -->
```

Body of coroutine (a closure)
Example (yield passes values to master)

```javascript
var co = create_coroutine(function(){
    yield(1)
    yield(2)
    yield(3)
})

print(resume(co)) -->
print(resume(co)) -->
print(resume(co)) -->
print(resume(co)) -->
resume(co) -->
```
Example (pass values to initial yield)

```javascript
var co = create_coroutine(function(x){
    print(x)
    yield()
})

resume(co, 1) -->
resume(co) -->
```
Test yourself

var co = create_coroutine(function(x){
    print("1", x)
    print("2", yield())
})

resume(co, "hello") -->
resume(co, "world") -->
**Iterator factory for permgen**

var permgen(a, n=len(a)) {
  if (n <= 1) { yield(a) } /* used to be 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]
    }
  }
}

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

Define control abstractions impossible with functions:

- lazy iterators
- push or pull producer-consumer patterns
- backtracking
- regexes
- exceptions

We will see some of these in lecture and PA
Stackful vs. stackless coroutines
Python generators

Python generators are coroutines with a limitation:

yield must occur in the body of the coroutine

That is, the call stack must be empty
Consumer-Producer Pattern
Create a dataflow on streams

Process the values from permgen

We can apply operations:

\[
\text{for } v \text{ in toUpperCaseF(permgen(...)) } \{ \text{process}(v) \}
\]

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

```javascript
var filter(ant, f) {
    var co = coroutine(function() {
        while (true) {
            -- resume antecessor to obtain value
            var x = ant();
            -- yield transformed value
            yield(f(x));
        }
    })
    function() { resume(co, 0) }
}

f1 = function(x) { ... }
f2 = function(x) { ... }
consumer(filter(filter(producer(), f1), f2))
```
How to implement such pipelines

Producer-consumer pattern: often a pipeline structure

producer → filter → 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 of 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 next time
What you need to know

• Iterators
• Programming with coroutines
• Write push and pull producer-consumer patterns
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 3rd edition.

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

Coroutine examples are from Revisiting Coroutines.