Python generators

p. 185

counter2

```
def nats():
    n = 0
    while True:
        yield n
        n += 1

g = nats()

print(next(g) + next(g) + next(g))
```

An infinite loop with side effects

loop

```
(let ([n 0])
  (define (loop)
    (displayln  n)
    (set! n (add1 n))
    (loop))
  (loop))
```

- nothing surprising, but things clearly happen before the loop finishes

Wrap the loop in (generator ...)

counter2

```
(define nats
  (generator ()
      (let ([n 0])
        (define (loop)
          (yield  n)
          (set! n (add1 n))
          (loop))
        (loop))))
```

- replace `displayln` with `yield`
- g can be suspended and restarted
- trace the control flow in the debugger

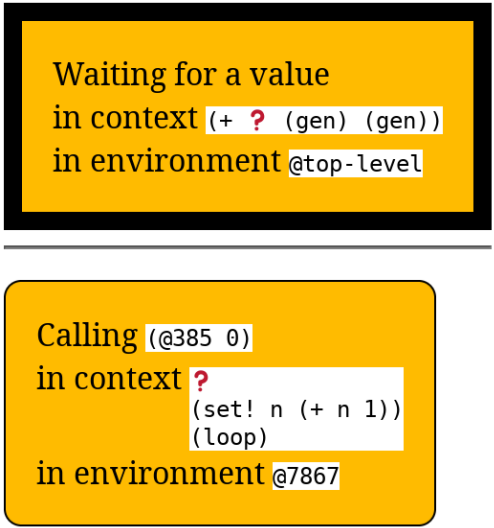
(sortof) Translating to SMoL

```
(deffun (yield n) n)

(deffun (gen)
  (defvar n 0)
  (deffun (loop)
    (yield n)
    (set! n (+ n 1))
    (loop))
  (loop))

(+ (gen) (gen) (gen))
```

stacker

- Starting Loop stacker
- Calling Yield stacker

Stack of contexts



- We can think about the bottom (generator) stack as independent
- in this case especially since it never returns

What is `yield`

p. 189

Unlike our fake yield in `smol`, `yield` should

- store the generator's stack,
- return a value to the other stack

Generators have their own stack I

- break tail call optimization, so we can see the stack

loop2

```
(let ([n 0])
  (define (loop)
    (displayln  n)
    (set! n (add1 n))
    (cons (loop) empty))
  (loop))
```

Generators have their own stack II

- every time we re-enter `nats`, we can see the previous stack levels

counter3

```
(define nats
  (generator ()
      (let ([n 0])
        (define (nat-loop)
          (yield  n)
          (set! n (add1 n))
          (cons (nat-loop) empty))
        (nat-loop))))
```

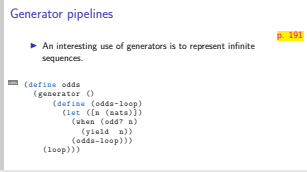
Generator pipelines

p. 191

- An interesting use of generators is to represent infinite sequences.

odds

```
(define odds
  (generator ()
      (define (odds-loop)
        (let ([n (nats)])
          (when (odd? n)
            (yield  n))
          (odds-loop)))
    (loop)))
```



1. This is translated into racket from the books python example, mainly because it lets us see the independent stacks of the two generators

Generator pipelines II

- Disable TCO, trace the stack in the DrRacket Debugger

odds2

```
(define odds
  (generator ()
      (define (odd-loop)
        (let ([n (nats)])
          (when (odd? n)
            (yield  n))
          (cons (odd-loop) empty)))
    (odd-loop)))
```

Continuations

- Consider the context `(+ ? (nat) (nat))`
- The ? is something like a formal-parameter, and the whole context is something like a function.
- in racket these contexts are called continuations, and `let/cc` is one primitive to work with them.
- `(let/cc id body)` binds the current continuation to `id`, and it can be called like a function in `body`.

Continuations

- Consider the context `(+ ? (nat) (nat))`
- The `?` is something like a formal-parameter, and the whole context is something like a function.
- in racket these contexts are called continuations, and `let/cc` is one primitive to work with them.
- `(let/cc id body)` binds the current continuation to `id`, and it can be called like a function in `body`.

1. In fact closures can be used simulate continuations, but it requires a particular style of writing code called *continuation passing style*
2. Continuations are a common implementation technique for interpreters, but less common as a language feature

let/cc examples

p. 209

- Continuations add generalized short circuit evaluation

let/cc

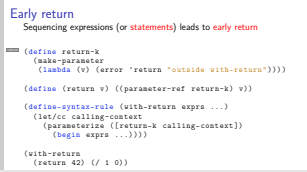
```
;; (test ? 3)
(test (let/cc k 3) 3)
;; (test ? 3)
(test (let/cc k (k 3)) 3)
;;  (test (+ 1 ?) 4 )
(test (+ 1 (let/cc k (k 3))) 4)
;;  (test ? 3)
(test (let/cc k (+ 2 (k 3))) 3)
;; (test (+ 1 ?) 4)
(test (+ 1 (let/cc k (+ 2 (k 3)))) 4)
```

Early return

Sequencing expressions (or statements) leads to early return

return

```
(define return-k
  (make-parameter
   (lambda (v) (error 'return "outside with-return"))))

(define (return v) ((parameter-ref return-k) v))

(define-syntax-rule (with-return exprs ...)
  (let/cc calling-context
    (parameterize ([return-k calling-context])
      (begin exprs ...))))

(with-return
  (return 42) (/ 1 0))
```



1. From the point of view of the type system, continutations are single parameter functions

Exception handling

- Close related to early return is exception handling

throw1

```
(define exception (make-parameter identity))

(define (throw msg) ((parameter-ref exception) msg))

(define-syntax-rule
  (try expr ...  (catch (id) recovers ...))
  (let ([recovery (lambda (id) recovers ...)])
    (let/cc esc
      (parameterize
          ([exception
            (lambda (x) (esc (recovery x)))])
        (begin expr ...)))))
```

Using the exception handler

throw1

```
(try
 (throw "abort!")
 (/ 1 0)
 (display "done")
 (catch (x)
   (display (string-append "caught " x))))
```

Nested try-catch blocks

throw2

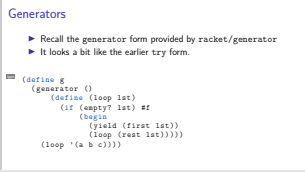
```
(try
 (try
  (throw "abort 1\n") (display "unreached 1")
  (catch (x) (display (string-append "1:" x))))
 (throw "abort 2\n") (display "unreached 2")
 (catch (x) (display (string-append "2:" x))))
```

Generators

- Recall the `generator` form provided by `racket/generator`
- It looks a bit like the earlier `try` form.

lgen

```
(define g
  (generator ()
      (define (loop lst)
        (if (empty? lst) #f
            (begin
              (yield (first lst))
              (loop (rest lst)))))
    (loop '(a b c))))
```



1. The generators here are based on those discussed in Chapter 14 of PLAI2 http://cs.brown.edu/courses/cs173/2012/book/Control_Operations.html
2. The approach here relies on parameters (dynamic scope), rather than on macros (as the version in PLAI).

Building Generators

Roughly speaking, generators require two control flow features:

- early return, which we just did, and
- resuming execution, which is more exotic as a language feature

Checkpoints

```
(define printer
  (with-checkpoint
    (display "first\n")
        (checkpoint!)
        (display
           "second\n")))
```

We want that execution restarts at the last `(checkpoint!)` reached.

```
(printer)
(printer)
(printer)
```

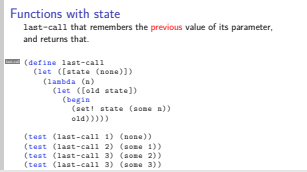
```
first
second
second
second
```

Functions with state

`last-call` that remembers the previous value of its parameter, and returns that.

last-call

```
(define last-call
  (let ([state (none)])
    (lambda (n)
      (let ([old state])
        (begin
          (set! state (some n))
          old)))))

(test (last-call 1) (none))
(test (last-call 2) (some 1))
(test (last-call 3) (some 2))
(test (last-call 3) (some 3))
```



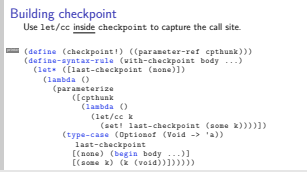
1. We could combine boxes with closures for this, but since we don't need the pass-by-reference features of boxes, we will use the usually-forbidden `set!` instead
2. The "tricky" bit is the use of `let` to define a variable to preserve the state in. This variable is visible only inside the `define`. This "let-over-lambda" pattern should be fairly familiar by now.
3. Note also the use of the plait `Option` type. This could be avoided in plain `racket` or `typed/racket`

Building checkpoint

Use `let/cc` inside `checkpoint` to capture the call site.

printer

```
(define (checkpoint!) ((parameter-ref cpthunk)))
(define-syntax-rule (with-checkpoint body ...)
  (let* ([last-checkpoint (none)])
    (lambda ()
      (parameterize
          ([cpthunk
            (lambda ()
              (let/cc k
                (set! last-checkpoint (some k))))])
        (type-case (Optionof (Void -> 'a))
           last-checkpoint
          [(none) (begin body ...)]
          [(some k) (k (void))])))))
```



1. Now that we know how to store store things for future invocations of a function, we can use a combination of ‘let‘ and ‘set¡ to store a continuation.
2. We might loosely call the place where `checkpoint!` is invoked the call site

Generators

- two uses of `let/cc`

generator

```
(let/cc dyn-k ;; generator call site
  (parameterize ([yield-param
                  (lambda (v)
                    (let/cc gen-k ;; yield call site
                      (begin
                        (set! last-checkpoint
                              (some gen-k))
                        (dyn-k v))))])
    (type-case (Optionof ('a -> 'b)) last-checkpoint
        [(none) (let ([arg v]) (begin exprs ...))]
        [(some k) (k v)])))
```

Using the generator 1/2

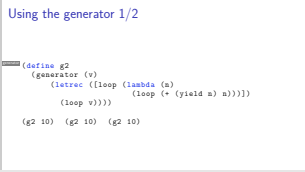
generator

```
(define g1
  (generator (v)
      (letrec ([loop (lambda (n)
                       (begin
                         (yield n)
                         (loop (+ n 1))))])
        (loop v))))

(g1 10)  (g1 10)  (g1 10)
```

Using the generator 1/2

generator

```
(define g2
  (generator (v)
      (letrec ([loop (lambda (n)
                       (loop (+ (yield n) n)))])
        (loop v))))

(g2 10)  (g2 10)  (g2 10)
```



1. The identifier names are different, but my generator solution is based on the macro based solution from [Chapter 14 of PLAI](http://cs.brown.edu/courses/cs173/2012/book/Control_Operations.html). Thever