PL: Lecture #29

Tuesday, December 3rd

Playing with Continuations extra

> ☞ ***PLAI §19***

So far we've seen a number of "tricks" that can be done with continuations. The simplest was aborting a computation — here's an implementation of functions with a `return` that can be used to exit the function early:

```
(define-syntax (fun stx)
  (syntax-case stx ()
    [(_ name (x ...) body ...)
     (with-syntax ([return (datum->syntax #'name 'return)])
       #'(define (name x ...) (let/cc return body ...)))]))

;; try it:
(fun mult (list)
  (define (loop list)
    (cond [(null? list) 1]
          [(zero? (first list)) (return 0)] ; early return
          [else (* (first list) (loop (rest list)))]))
  (loop list))
(mult '(1 2 3 0 x))
```

[Side note: This is a cheap demonstration. If we rewrite the loop tail-recursively, then aborting it is simple — just return 0 instead of continuing the loop. And that's not a coincidence, aborting from a tail-calling loop is easy, and CPS makes such aborts possible by making only tail calls.]

But such uses of continuations are simple because they're used only to "jump out" of some (dynamic) context. More exotic uses of continuations rely on the ability to jump into a previously captured continuation. In fact, our `web-read` implementation does just that (and more). The main difference is that in the former case the continuation is used exactly once — either explicitly by using it, or implicitly by returning a value (without aborting). If a continuation can be used after the corresponding computation is over, then why not use it over and over again… For example, we can try an infinite loop by capturing a continuation and later use it as a jump target:

```
(define (foo)
  (define loop (let/cc k k))    ; captured only for the context
  (printf "Meh.\n")
  (loop 'something))            ; need to give it some argument
```

This almost works — we get two printouts so clearly the jump was successful. The problem is that the captured `loop` continuation is the one that expects a value to bind to `loop` itself, so the second attempted call has `'something` as the value of `loop`, obviously, leading to an error. This can be used as a hint for a solution — simply pass the continuation to itself:

```
(define (foo)
  (define loop (let/cc k k))
  (printf "Meh.\n")
  (loop loop))                  ; keep the value of `loop'
```

Another way around this problem is to capture the continuation that is just *after* the binding — but we can't do that (try it…). Instead, we can use side-effects:

```
(define (foo)
  (define loop (box #f))
  (let/cc k (set-box! loop k))  ; cont. of the outer expression
  (printf "Meh.\n")
  ((unbox loop) 'something))
```

> *Note: the `'something` value goes to the continuation which makes it the result of the `(let/cc ...)` expression — which means that it's never actually used now.*

This might seem like a solution that is not as "clean", since it uses mutation — but note that the problem that we're solving stems from a continuation that exposes the mutation that Racket performs when it creates a new binding.

Here's an example of a loop that does something a little more interesting in a goto-like kind of way:

```
(define (foo)
  (define n (box 0))
  (define loop (box #f))
  (let/cc k (set-box! loop k))
  (set-box! n (add1 (unbox n)))
  (printf "n = ~s\n" (unbox n))
  ((unbox loop)))
```

> *Note: in this example the continuation is called without any inputs. How is this possible? As we've seen, the `'something` value in the last example is the never-used result of the `let/cc` expression. In this case, the continuation is called with no input, which means that the `let/cc` expression evaluates to … nothing! This is not just some `void` value, but no value at all. The complete story is that in Racket expressions can evaluate to multiple values, and in this case, it's no values at all.*

Given such examples it's no wonder that continuations tend to have a reputation for being "similar to goto in their power". This reputation has some vague highlevel justification in that both features can produce obscure "spaghetti code" — but in practice they are very different. On one hand continuations are more limited: unlike `goto`, you can only jump to a continuation that you have already "visited". On the other hand, jumping to a continuation is doing much more than jumping to a goto label, the latter changes the next instruction to execute (the "program counter" register), but the former changes current computation context (in low level terms, both the PC register and the stack). (See also the `setjmp()` and `longjmp()` functions in C, or the `context` related functions (`getcontext()`, `setcontext()`, `swapcontext()`).)

To demonstrate how different continuations are from plain gotos, we'll start with a variation of the above loop — instead of performing the loop we just store it in a global box, and we return the counter value instead of printing it:

```
(define loop (box #f))

(define (foo)
  (define n (box 0))
  (let/cc k (set-box! loop k))
  (set-box! n (add1 (unbox n)))
  (unbox n))
```

Now, the first time we call (foo), we get 1 as expected, and then we can call (unbox loop) to re-invoke the continuation and get the following numbers:

```
> (foo)
1
> ((unbox loop))
2
```

```
> ((unbox loop))
3
```

[Interesting experiment: try doing the same, but use (list (foo)) as the first interaction, and the same ((unbox loop)) later.]

The difference between this use and a `goto` is now evident: we're not just just jumping to a label — we're jumping back into a computation that returns the next number. In fact, the continuation can include a context that is outside of `foo`, for example, we can invoke the continuation from a different function, and `loop` can be used to search for a specific number:

```
(define (bar)
  (let ([x (foo)])
    (unless (> x 10) ((unbox loop)))
    x))
```

and now (bar) returns 11. The loop is now essentially going over the obvious part of `foo` but also over parts of `bar`. Here's an example that makes it even more obvious:

```
(define (bar)
  (let* ([x (foo)]
         [y (* x 2)])
    (unless (> x 10) ((unbox loop)))
    y))
```

Since the `y` binding becomes part of the loop. Our `foo` can be considered as a kind of a producer for natural numbers that can be used to find a specific number, invoking the `loop` continuation to try the next number when the last one isn't the one we want.

The Ambiguous Operator: `amb` extra

Our `foo` is actually a very limited version of something that is known as "McCarthy's Ambiguous Operator", usually named `amb`. This operator is used to perform a kind of a backtrack-able choice among several values.

To develop our `foo` into such an `amb`, we begin by renaming `foo` as `amb` and `loop` as `fail`, and instead of returning natural numbers in sequence we'll have it take a list of values and return values from this list. Also, we will use mutable variables instead of boxes to reduce clutter (a feature that we've mostly ignored so far). The resulting code is:

```
(define fail #f)

(define (amb choices)
  (let/cc k (set! fail k))
  (let ([choice (first choices)])
    (set! choices (rest choices))
    choice))
```

Of course, we also need to check that we actually have values to return:

```
(define fail #f)

(define (amb choices)
  (let/cc k (set! fail k))
  (if (pair? choices)
    (let ([choice (first choices)])
      (set! choices (rest choices))
      choice)
    (error "no more choices!")))
```

The resulting `amb` can be used in a similar way to the earlier `foo`:

```
(define (bar)
  (let* ([x (amb '(5 10 15 20))]
         [y (* x 2)])
    (unless (> x 10) (fail))
    y))
(bar)
```

This is somewhat useful, but searching through a simple list of values is not too exciting. Specifically, we can have only one search at a time. Making it possible to have multiple searches is not too hard: instead of a single failure continuation, store a stack of them, where each new `amb` pushes a new one on it.

We define `failures` as this stack and push a new failure continuation in each `amb`. `fail` becomes a function that simply invokes the most recent failure continuation, if one exists.

```
(define failures null)

(define (fail)
  (if (pair? failures)
    ((first failures))
    (error "no more choices!")))

(define (amb choices)
  (let/cc k (set! failures (cons k failures)))
  (if (pair? choices)
    (let ([choice (first choices)])
      (set! choices (rest choices))
      choice)
    (error "no more choices!")))
```

This is close, but there's still something missing. When we run out of options from the `choices` list, we shouldn't just throw an error — instead, we should invoke the previous failure continuation, if there is one. In other words, we want to use `fail`, but before we do, we need to pop up the top-most failure continuation since it is the one that we are currently dealing with:

```
(define failures null)

(define (fail)
  (if (pair? failures)
    ((first failures))
    (error "no more choices!")))

(define (amb choices)
  (let/cc k (set! failures (cons k failures)))
  (if (pair? choices)
    (let ([choice (first choices)])
      (set! choices (rest choices))
      choice)
    (begin (set! failures (rest failures))
           (fail))))

(define (assert condition)
  (unless condition (fail)))
```

Note the addition of a tiny `assert` utility, something that is commonly done with `amb`. We can now play with this code as before:

```
(let* ([x (amb '(5 10 15 20))]
       [y (* x 2)])
```

```
    (unless (> x 10) (fail))
    y)
```

But of course the new feature is more impressive, for example, find two numbers that sum up to 6 and the first is the square of the second:

```
  (let ([a (amb '(1 2 3 4 5 6 7 8 9 10))]
        [b (amb '(1 2 3 4 5 6 7 8 9 10))])
    (assert (= 6 (+ a b)))
    (assert (= a (* b b)))
    (list a b))
```

Find a Pythagorean triplet:

```
  (let ([a (amb '(1 2 3 4 5 6))]
        [b (amb '(1 2 3 4 5 6))]
        [c (amb '(1 2 3 4 5 6))])
    (assert (= (* a a) (+ (* b b) (* c c))))
    (list a b c))
```

Specifying the list of integers is tedious, but easily abstracted into a function:

```
  (let* ([int6 (lambda () (amb '(1 2 3 4 5 6)))]
         [a (int6)]
         [b (int6)]
         [c (int6)])
    (assert (= (* a a) (+ (* b b) (* c c))))
    (list a b c))
```

A more impressive demonstration is finding a solution to tests known as "Self-referential Aptitude Test", for example, **here's one such test** (by Christian Schulte and Gert Smolka) — it's a 10-question multiple choice test:

1. The first question whose answer is b is question (a) 2; (b) 3; (c) 4; (d) 5; (e) 6.
2. The only two consecutive questions with identical answers are questions (a) 2 and 3; (b) 3 and 4; (c) 4 and 5; (d) 5 and 6; (e) 6 and 7.
3. The answer to this question is the same as the answer to question (a) 1; (b) 2; (c) 4; (d) 7; (e) 6.
4. The number of questions with the answer a is (a) 0; (b) 1; (c) 2; (d) 3; (e) 4.
5. The answer to this question is the same as the answer to question (a) 10; (b) 9; (c) 8; (d) 7; (e) 6.
6. The number of questions with answer a equals the number of questions with answer (a) b; (b) c; (c) d; (d) e; (e) none of the above.
7. Alphabetically, the answer to this question and the answer to the following question are (a) 4 apart; (b) 3 apart; (c) 2 apart; (d) 1 apart; (e) the same.
8. The number of questions whose answers are vowels is (a) 2; (b) 3; (c) 4; (d) 5; (e) 6.
9. The number of questions whose answer is a consonant is (a) a prime; (b) a factorial; (c) a square; (d) a cube; (e) divisible by 5.
10. The answer to this question is (a) a; (b) b; (c) c; (d) d; (e) e.

and the solution is pretty much a straightforward translation:

```
  (define (self-test)
    (define (choose-letter) (amb '(a b c d e)))
    (define q1  (choose-letter))
    (define q2  (choose-letter))
    (define q3  (choose-letter))
    (define q4  (choose-letter))
    (define q5  (choose-letter))
    (define q6  (choose-letter))
    (define q7  (choose-letter))
    (define q8  (choose-letter))
```

```
(define q9  (choose-letter))
(define q10 (choose-letter))
;; 1. The first question whose answer is b is question (a) 2;
;;    (b) 3; (c) 4; (d) 5; (e) 6.
(assert (eq? q1 (cond [(eq? q2 'b) 'a]
                      [(eq? q3 'b) 'b]
                      [(eq? q4 'b) 'c]
                      [(eq? q5 'b) 'd]
                      [(eq? q6 'b) 'e]
                      [else (assert #f)])))
;; 2. The only two consecutive questions with identical answers
;;    are questions (a) 2 and 3; (b) 3 and 4; (c) 4 and 5; (d) 5
;;    and 6; (e) 6 and 7.
(define all (list q1 q2 q3 q4 q5 q6 q7 q8 q9 q10))
(define (count-same-consecutive l)
  (define (loop x l n)
    (if (null? l)
      n
      (loop (first l) (rest l)
            (if (eq? x (first l)) (add1 n) n))))
  (loop (first l) (rest l) 0))
(assert (eq? q2 (cond [(eq? q2 q3) 'a]
                      [(eq? q3 q4) 'b]
                      [(eq? q4 q5) 'c]
                      [(eq? q5 q6) 'd]
                      [(eq? q6 q7) 'e]
                      [else (assert #f)])))
(assert (= 1 (count-same-consecutive all))) ; exactly one
;; 3. The answer to this question is the same as the answer to
;;    question (a) 1; (b) 2; (c) 4; (d) 7; (e) 6.
(assert (eq? q3 (cond [(eq? q3 q1) 'a]
                      [(eq? q3 q2) 'b]
                      [(eq? q3 q4) 'c]
                      [(eq? q3 q7) 'd]
                      [(eq? q3 q6) 'e]
                      [else (assert #f)])))
;; 4. The number of questions with the answer a is (a) 0; (b) 1;
;;    (c) 2; (d) 3; (e) 4.
(define (count x l)
  (define (loop l n)
    (if (null? l)
      n
      (loop (rest l) (if (eq? x (first l)) (add1 n) n))))
  (loop l 0))
(define num-of-a (count 'a all))
(define num-of-b (count 'b all))
(define num-of-c (count 'c all))
(define num-of-d (count 'd all))
(define num-of-e (count 'e all))
(assert (eq? q4 (case num-of-a
                  [(0) 'a]
                  [(1) 'b]
                  [(2) 'c]
                  [(3) 'd]
                  [(4) 'e]
                  [else (assert #f)])))
;; 5. The answer to this question is the same as the answer to
;;    question (a) 10; (b) 9; (c) 8; (d) 7; (e) 6.
```

```
(assert (eq? q5 (cond [(eq? q5 q10) 'a]
                      [(eq? q5 q9) 'b]
                      [(eq? q5 q8) 'c]
                      [(eq? q5 q7) 'd]
                      [(eq? q5 q6) 'e]
                      [else (assert #f)])))
;; 6. The number of questions with answer a equals the number of
;;    questions with answer (a) b; (b) c; (c) d; (d) e; (e) none
;;    of the above.
(assert (eq? q6 (cond [(= num-of-a num-of-b) 'a]
                      [(= num-of-a num-of-c) 'b]
                      [(= num-of-a num-of-d) 'c]
                      [(= num-of-a num-of-e) 'd]
                      [else 'e])))
;; 7. Alphabetically, the answer to this question and the answer
;;    to the following question are (a) 4 apart; (b) 3 apart; (c)
;;    2 apart; (d) 1 apart; (e) the same.
(define (choice->integer x)
  (case x [(a) 1] [(b) 2] [(c) 3] [(d) 4] [(e) 5]))
(define (distance x y)
  (if (eq? x y)
    0
    (abs (- (choice->integer x) (choice->integer y)))))
(assert (eq? q7 (case (distance q7 q8)
                  [(4) 'a]
                  [(3) 'b]
                  [(2) 'c]
                  [(1) 'd]
                  [(0) 'e]
                  [else (assert #f)])))
;; 8. The number of questions whose answers are vowels is (a) 2;
;;    (b) 3; (c) 4; (d) 5; (e) 6.
(assert (eq? q8 (case (+ num-of-a num-of-e)
                  [(2) 'a]
                  [(3) 'b]
                  [(4) 'c]
                  [(5) 'd]
                  [(6) 'e]
                  [else (assert #f)])))
;; 9. The number of questions whose answer is a consonant is (a) a
;;    prime; (b) a factorial; (c) a square; (d) a cube; (e)
;;    divisible by 5.
(assert (eq? q9 (case (+ num-of-b num-of-c num-of-d)
                  [(2 3 5 7) 'a]
                  [(1 2 6)   'b]
                  [(0 1 4 9) 'c]
                  [(0 1 8)   'd]
                  [(0 5 10)  'e]
                  [else (assert #f)])))
;; 10. The answer to this question is (a) a; (b) b; (c) c; (d) d;
;;     (e) e.
(assert (eq? q10 q10)) ; (note: does nothing...)
;; The solution should be: (c d e b e e d c b a)
all)
```

Note that the solution is simple because of the freedom we get with continuations: the search is not a sophisticated one, but we’re free to introduce ambiguity points anywhere that fits, and mix assertions with other code without worrying about control flow (as you do in an implementation that uses explicit loops). On the other hand, it is not too efficient since it uses a naive search strategy. (This could be improved somewhat by deferring ambiguous points, for example, don't assign q7, q8, q9, and q10 before the first question; but much of the cost comes from the strategy for implementing continuation in Racket, which makes capturing continuations a relatively expensive operation.)

When we started out with the modified loop, we had a representation of an arbitrary natural number — but with the move to lists of choices we lost the ability to deal with such infinite choices. Getting it back is simple: delay the evaluation of the `amb` expressions. We can do that by switching to a list of thunks instead. The change in the code is in the result: just return the result of calling `choice` instead of returning it directly. We can then rename `amb` to `amb/thunks` and reimplement `amb` as a macro that wraps all of its sub-forms in thunks.

```
(define (amb/thunks choices)
  (let/cc k (set! failures (cons k failures)))
  (if (pair? choices)
    (let ([choice (first choices)])
      (set! choices (rest choices))
      (choice))                   ;*** call the choice thunk
    (begin (set! failures (rest failures))
           (fail))))

(define-syntax-rule (amb E ...)
  (amb/thunks (list (lambda () E) ...)))
```

With this, we can implement code that computes choices rather than having them listed:

```
(define (integers-between n m)
  (assert (<= n m))
  (amb n (integers-between (add1 n) m)))
```

or even ones that are infinite:

```
(define (integers-from n)
  (amb n (integers-from (add1 n))))
```

As with any infinite sequence, there are traps to avoid. In this case, trying to write code that can find any Pythagorean triplet as:

```
(collect 7
  (let ([a (integers-from 1)]
        [b (integers-from 1)]
        [c (integers-from 1)])
    (assert (= (* a a) (+ (* b b) (* c c))))
    (list a b c)))
```

will not work. The problem is that the search loop will keep incrementing `c`, and therefore will not find any solution. The search can work if only the top-most choice is infinite:

```
(collect 7
  (let* ([a (integers-from 1)]
         [b (integers-between 1 a)]
         [c (integers-between 1 a)])
    (assert (= (* a a) (+ (* b b) (* c c))))
    (list a b c)))
```

The complete code follows:

```
;; The ambiguous operator and related utilities

#lang racket
```

```
(define failures null)

(define (fail)
  (if (pair? failures)
    ((first failures))
    (error "no more choices!")))

(define (amb/thunks choices)
  (let/cc k (set! failures (cons k failures)))
  (if (pair? choices)
    (let ([choice (first choices)])
      (set! choices (rest choices))
      (choice))
    (begin (set! failures (rest failures))
           (fail))))

(define-syntax-rule (amb E ...)
  (amb/thunks (list (lambda () E) ...)))

(define (assert condition)
  (unless condition (fail)))

(define (integers-between n m)
  (assert (<= n m))
  (amb n (integers-between (add1 n) m)))

(define (integers-from n)
  (amb n (integers-from (add1 n))))

(define (collect/thunk n thunk)
  (define results null)
  (let/cc too-few
    (set! failures (list too-few))
    (define result (thunk))
    (set! results (cons result results))
    (set! n (sub1 n))
    (unless (zero? n) (fail)))
  (set! failures null)
  (reverse results))

(define-syntax collect
  (syntax-rules ()
    ;; collect N results
    [(_ N E) (collect/thunk N (lambda () E))]
    ;; collect all results
    [(_ E) (collect/thunk -1 (lambda () E))]))
```

As a bonus, the code includes a `collect` tool that can be used to collect a number of results — it uses `fail` to iterate until a sufficient number of values is collected. A simple version is:

```
(define (collect/thunk n thunk)
  (define results null)
  (define result (thunk))
  (set! results (cons result results))
  (set! n (sub1 n))
  (unless (zero? n) (fail))
  (reverse results))
```

(Question: why does this code use mutation to collect the results?)

But since this might run into a premature failure, the actual version in the code installs its own failure continuation that simply aborts the collection loop. To try it out:

```
(collect (* (integers-between 1 3) (integers-between 1 5)))
```

Generators (and Producers) extra

Another popular facility that is related to continuations is generators. The idea is to split code into separate “producers” and “consumers”, where the computation is interleaved between the two. This simplifies some notoriously difficult problems. It is also a twist on the idea of co-routines, where two functions transfer control back and forth as needed. (Co-routines can be developed further into a “cooperative threading” system, but we will not cover that here.)

A classical example that we have mentioned previously is the “same fringe” problem. One of the easy solutions that we talked about was to run two processes that spit out the tree leaves, and a third process that grabs both outputs as they come and compares them. Using a lazy language allowed a very similar solution, where the two processes are essentially represented as two lazy lists. But with continuations we can find a solution that works in a strict language too, and in fact, one that is very close to the two processes metaphor.

The fact that continuations can support such a solution shouldn’t be surprising: as with the kind of server-client interactions that we’ve seen with the web language, and as with the `amb` tricks, the main theme is the same — the idea of suspending computation. (Intuitively, this also explains why a lazy language is related: it is essentially making all computations suspendable in a sense.)

To implement generators, we begin with a simple code that we want to eventually use:

```
(define (producer)
  (yield 1)
  (yield 2)
  (yield 3))
```

where `yield` is expected to behave similarly to a `return` — it should make the function return 1 when called, and then somehow return 2 and 3 on subsequent calls. To make it easier to develop, we’ll make `yield` an argument to the producer:

```
(define (producer yield)
  (yield 1)
  (yield 2)
  (yield 3))
```

To use this producer, we need to find a proper value to call it with. Sending it an identity, (lambda (x) x), is clearly not going to work: it will make all `yield`s executed on the first call, returning the last value. Instead, we need some way to abort the computation on the first `yield`. This, of course, can be done with a continuation, which we should send as the value of the `yield` argument. And indeed,

```
> (let/cc k (producer k))
1
```

returns `1` as we want. But if we use this expression again, we get more `1`s as results:

```
> (let/cc k (producer k))
1
> (let/cc k (producer k))
1
```

The problem is obvious: our producer starts every time from scratch, always sending the first value to the given continuation. Instead, we need to make it somehow save where it stopped — its own continuation — and on subsequent calls it should resume from that point. We start with adding a `resume` continuation to save our position into:

```
(define (producer yield)
  (define resume #f)
  (if (not resume)   ; we just started, so no resume yet
    (begin (yield 1)
           (yield 2)
           (yield 3))
    (resume 'blah))) ; we have a resume, use it
```

Next, we need to make it so that each use of `yield` will save its continuation as the place to resume from:

```
(define (producer yield)
  (define resume #f)
  (if (not resume)
    (begin (let/cc k (set! resume k) (yield 1))
           (let/cc k (set! resume k) (yield 2))
           (let/cc k (set! resume k) (yield 3)))
    (resume 'blah)))
```

But this is still broken in an obvious way: every time we invoke this function, we define a new local `resume` which is always `#f`, leaving us with the same behavior. We need `resume` to persist across calls — which we can get by "pulling it out" using a `let`:

```
(define producer
  (let ([resume #f])
    (lambda (yield)
      (if (not resume)
        (begin (let/cc k (set! resume k) (yield 1))
               (let/cc k (set! resume k) (yield 2))
               (let/cc k (set! resume k) (yield 3)))
        (resume 'blah)))))
```

And this actually works:

```
> (let/cc k (producer k))
1
> (let/cc k (producer k))
2
> (let/cc k (producer k))
3
```

(Tracing how it works is a good exercise.)

Before we continue, we'll clean things up a little. First, to make it easier to get values from the producer, we can write a little helper:

```
(define (get producer)
  (let/cc k (producer k)))
```

Next, we can define a local helper inside the producer to improve it in a similar way by making up a `yield` that wraps the `raw-yield` input continuation (also flip the condition):

```
(define producer
  (let ([resume #f])
    (lambda (raw-yield)
      (define (yield value)
        (let/cc k (set! resume k) (raw-yield value)))
      (if resume
        (resume 'blah)
        (begin (yield 1)
               (yield 2)
               (yield 3))))))
```

And we can further abstract out the general producer code from the specific 1-2-3 producer that we started with. The complete code is now:

```
(define (make-producer producer)
  (let ([resume #f])
    (lambda (raw-yield)
      (define (yield value)
        (let/cc k (set! resume k) (raw-yield value)))
      (if resume
        (resume 'blah)
        (producer yield)))))

(define (get producer)
  (let/cc k (producer k)))

(define producer
  (make-producer (lambda (yield)
                   (yield 1)
                   (yield 2)
                   (yield 3))))
```

When we now evaluate (get producer) three times, we get back the three values in the correct order. But there is a subtle bug here, first try this (after re-running!):

```
> (list (get producer) (get producer))
```

Seems that this is stuck in an infinite loop. To see where the problem is, re-run to reset the producer, and then we can see the following interaction:

```
> (* 10 (get producer))
10
> (* 100 (get producer))
20
> (* 12345 (get producer))
30
```

This looks weird… Here’s a more clarifying example:

```
> (list (get producer))
'(1)
> (get producer)
'(2)
> (get producer)
'(3)
```

Can you see what’s wrong now? It seems that all three invocations of the producer use the same continuation — the first one, specifically, the `(list <*>)` continuation. This also explains why we run into an infinite loop with `(list (get producer) (get producer))` — the first continuation is:

```
(list <*> (get producer))
```

so when we get the first `1` result we plug it in and proceed to evaluate the second `(get producer)`, but that re-invokes the *first* continuation again, getting into an infinite loop. We need to look closely at our `make-producer` to see the problem:

```
(define (make-producer producer)
  (let ([resume #f])
    (lambda (raw-yield)
      (define (yield value)
        (let/cc k (set! resume k) (raw-yield value)))
      (if resume
        (resume 'blah)
        (producer yield)))))
```

When `(make-producer (lambda (yield) ...))` is first called, `resume` is initialized to `#f`, and the result is the `(lambda (raw-yield) ...)`, which is bound to the global `producer`. Next, we call this function, and since `resume` is `#f`, we apply the `producer` on our `yield` — which is a closure that has a reference to the `raw-yield` that we received — the continuation that was used in this first call. The problem is that on subsequent calls `resume` will contain a continuation which it is called, but this will jump back to that first closure with the original `raw-yield`, so instead of returning to the current calling context, we re-return to the first context — the same first continuation. The code can be structured slightly to make this a little more obvious: push the `yield` definition into the only place it is used (the first call):

```
(define (make-producer producer)
  (let ([resume #f])
    (lambda (raw-yield)
      (if resume
        (resume 'blah)
        (let ([yield (lambda (value)
                       (let/cc k
                         (set! resume k)
                         (raw-yield value)))])
          (producer yield))))))
```

`yield` is not used elsewhere, so this code has exactly the same meaning as the previous version. You can see now that when the producer is first used, it gets a `raw-yield` continuation which is kept in a newly made closure — and even though the following calls have *different* continuations, we keep invoking the first one. These calls get new continuations as their `raw-yield` input, but they ignore them. It just happened that the when we evaluated `(get producer)` three times on the REPL, all calls had essentially the same continuation (the `P` part of the REPL), so it seemed like things are working fine.

To fix this, we must avoid calling the same initial `raw-yield` every time: we must change it with each call so it is the right one. We can do this with another mutation — introduce another state variable that will refer to the correct `raw-yield`, and update it on every call to the producer. Here’s one way to do this:

```
(define (make-producer producer)
  (let ([resume #f]
        [return-to-caller #f])
    (lambda (raw-yield)
      (set! return-to-caller raw-yield)
      (if resume
        (resume 'blah)
        (let ([yield (lambda (value)
                       (let/cc k
                         (set! resume k)
                         (return-to-caller value)))])
          (producer yield))))))
```

Using this, we get well-behaved results:

```
> (list (get producer))
'(1)
> (* 8 (get producer))
16
> (get producer)
3
```

or (again, after restarting the producer by re-running the code):

```
> (list (get producer) (get producer) (get producer))
'(1 2 3)
```

> *Side-note: a different way to achieve this is to realize that when we invoke `resume`, we're calling the continuation that was captured by the `let/cc` expression. Currently, we're sending just `'blah` to that continuation, but we could send `raw-yield` there instead. With that, we can make that continuation be the target of setting the `return-to-caller` state variable. (This is how PLAI solves this problem.)*

```
(define (make-producer producer)
  (let ([resume #f])
    (lambda (raw-yield)
      (define return-to-caller raw-yield)
      (define (yield value)
        (set! return-to-caller
              (let/cc k
                (set! resume k)
                (return-to-caller value))))
      (if resume
        (resume raw-yield)
        (producer yield)))))
```

Continuing with our previous code, and getting the `yield` back into a a more convenient definition form, we have this complete code:

```
;; An implementation of producer functions

#lang racket

(define (make-producer producer)
  (let ([resume #f]
        [return-to-caller #f])
    (lambda (raw-yield)
      (define (yield value)
        (let/cc k (set! resume k) (return-to-caller value)))
      (set! return-to-caller raw-yield)
      (if resume
        (resume 'blah)
        (producer yield)))))

(define (get producer)
  (let/cc k (producer k)))

(define producer
  (make-producer (lambda (yield)
                   (yield 1)
                   (yield 2)
                   (yield 3))))
```

There is still a small problem with this code:

```
> (list (get producer) (get producer) (get producer))
'(1 2 3)
> (get producer)
;; infinite loop
```

Tracking this problem is another good exercise, and finding a solution is easy. (For example, throwing an error when the producer is exhausted, or returning `'done`, or returning the return value of the producer function.)

Delimited Continuations extra

While the continuations that we have seen are a useful tool, they are often “too global” — they capture the complete computation context. But in many cases we don’t want that, instead, we want to capture a specific context. In fact, this is exactly why producer code got complicated: we needed to keep capturing the `return-to-caller` continuation to make it possible to return to the correct context rather than re-invoking the initial (and wrong) context.

Additional work on continuations resulted in a feature that is known as “delimited continuations”. These kind of continuations are more convenient in that they don’t capture the complete context — just a potion of it up to a specific point. To see how this works, we’ll restart with a relatively simple producer definition:

```
(define producer
  (let ()
    (define (cont)
      (let/cc ret
        (define (yield value)
          (let/cc k (set! cont k) (ret value)))
        (yield 1)
        (yield 2)
        (yield 3)
        4))
    (define (generator) (cont))
    generator))
```

This producer is essentially the same as one that we’ve seen before: it seems to work in that it returns the desired values for every call:

```
> (producer)
1
> (producer)
2
> (producer)
3
```

But fails in that it always returns to the initial context:

```
> (list (producer))
'(1)
> (+ 100 (producer))
'(2)
> (* "bogus" (producer))
'(3)
```

Fixing this will lead us down the same path we’ve just been through: the problem is that `generator` is essentially an indirection “trampoline” function that goes to whatever `cont` currently holds, and except for the initial value of `cont` the other values are continuations that are captured inside `yield`, meaning that the calls are all using the same `ret` continuation that was grabbed once, at the beginning. To fix it, we will need to re-capture a return continuation on every use of `yield`, which we can do by modifying the `ret` binding, giving us a working version:

```
(define producer
  (let ()
    (define (cont)
      (let/cc ret
        (define (yield value)
          (let/cc k
            (set! cont (lambda () (let/cc r (set! ret r) (k))))
            (ret value)))
        (yield 1)
        (yield 2)
        (yield 3)
        4))
    (define (generator) (cont))
    generator))
```

This pattern of grabbing the current continuation and then jumping to another — `(let/cc k (set! cont k) (ret value))` — is pretty common, enough that there is a specific construct that does something similar: `control`. Translating the `let/cc` form to it produces:

```
(control k (set! cont ...) value)
```

A notable difference here is that we don't use a `ret` continuation. Instead, another feature of the `control` form is that the value returns to a specific point back in the current computation context that is marked with a `prompt`. (Note that the `control` and `prompt` bindings are not included in the default `racket` language, we need to get them from a library: `(require racket/control)`.) The fully translated code simply uses this `prompt` in place of the outer capture of the `ret` continuation:

```
(define producer
  (let ()
    (define (cont)
      (prompt
        (define (yield value)
          (control k
            (set! cont ???)
            value))
        (yield 1)
        (yield 2)
        (yield 3)
        4))
    (define (generator) (cont))
    generator))
```

We also need to translate the `(lambda () (let/cc r (set! ret r) (k)))` expression — but there is no `ret` to modify. Instead, we get the same effect by another use of `prompt` which is essentially modifying the implicitly used return continuation:

```
(define producer
  (let ()
    (define (cont)
      (prompt
        (define (yield value)
          (control k
            (set! cont (lambda () (prompt (k))))
            value))
        (yield 1)
        (yield 2)
        (yield 3)
        4))
    (define (generator) (cont))
    generator))
```

This looks like the previous version, but there's an obvious advantage: since there is no `ret` binding that we need to maintain, we can pull out the `yield` definition to a more convenient place:

```
(define producer
  (let ()
    (define (yield value)
      (control k
        (set! cont (lambda () (prompt (k))))
        value))
    (define (cont)
      (prompt
        (yield 1)
        (yield 2)
        (yield 3)
        4))
    (define (generator) (cont))
    generator))
```

Note that this is an important change, since the producer machinery can now be abstracted into a `make-producer` function, as we've done before:

```
(define (make-producer producer)
  (define (yield value)
    (control k
      (set! cont (lambda () (prompt (k))))
      value))
  (define (cont) (prompt (producer yield)))
  (define (generator) (cont))
  generator)

(define producer
  (make-producer (lambda (yield)
                   (yield 1)
                   (yield 2)
                   (yield 3)
                   4)))
```

This is, again, a common pattern in such looping constructs — where the continuation of the loop keeps modifying the prompt as we do in the thunk assigned to `cont`. There are two other operators that are similar to `control` and `prompt`, which re-instate the point to return to automatically. Confusingly, they have completely different name: `shift` and `reset`. In the case of our code, we simply do the straightforward translation, and drop the extra wrapping step inside the value assigned to `cont` since that is done automatically. The resulting definition becomes even shorter now:

```
(define (make-producer producer)
  (define (yield value) (shift k (set! cont k) value))
  (define (cont) (reset (producer yield)))
  (define (generator) (cont))
  generator)
```

(Question: which set of forms is the more powerful one?)

It even looks like this code works reasonably well when the producer is exhausted:

```
> (list (producer) (producer) (producer) (producer) (producer))
'(1 2 3 4 4)
```

But the problem is still there, except a but more subtle. We can see it if we add a side-effect:

```
(define producer
  (make-producer (lambda (yield)
                   (yield 1)
                   (yield 2)
                   (yield 3)
                   (printf "Hey!\n")
                   4)))
```

and now we get:

```
> (list (producer) (producer) (producer) (producer) (producer))
Hey!
Hey!
'(1 2 3 4 4)
```

This can be solved in the same way as we've discussed earlier — for example, grab the result value of the producer (which means that we get the value only after it's exhausted), then repeat returning that value. A particularly easy way to do this is to set `cont` to a thunk that returns the value — since the resulting `generator` function simply invokes it, we get the desired behavior of returning the last value on further calls:

```
(define (make-producer producer)
  (define (yield value) (shift k (set! cont k) value))
  (define (cont)
    (reset (let ([retval (producer yield)])
             ;; we get here when the producer is done
             (set! cont (lambda () retval))
             retval)))
  (define (generator) (cont))
  generator)

(define producer
  (make-producer (lambda (yield)
                   (yield 1)
                   (yield 2)
                   (yield 3)
                   (printf "Hey!\n")
                   4)))
```

and now we get the improved behavior:

```
> (list (producer) (producer) (producer) (producer) (producer))
Hey!
'(1 2 3 4 4)
```

Continuation Conclusions

Continuations are often viewed as a feature that is too complicated to understand and/or are hard to implement. As a result, very few languages provide general first-class continuations. Yet, they are an extremely useful tool since they enable implementing new kinds of control operators as user-written libraries. The "user" part is important here: if you want to implement producers (or a convenient `web-read`, or an ambiguous operator, or any number of other uses) in a language that doesn't have continuations your options are very limited. You can ask for the new feature and wait for the language implementors to provide it, or you can CPS the relevant code (and the latter option is possible only if you have complete control over the whole code source to transform). With continuations, as we have seen, it is not only possible to build such libraries, the resulting functionality is as if the language has the desired feature already built-in. For example, Racket comes with a generator library that is very similar to Python generators — but in contrast to Python, it is implemented completely in user code. (In fact, the implementation is very close to the delimited continuations version that we've seen last.)

Obviously, in cases where you don't have continuations and you need them (or rather when you need some functionality that is implementable via continuations), you will likely resort to the CPS approach, in some limited version. For example, the **Racket documentation search page** allows input to be typed while the search is happening.

This is a feature that by itself is not available in JavaScript — it is as if there are two threads running (one for the search and one to handle input), where JS is single-threaded on principle. This was implemented by making the search code side-effect free, then CPS-ing the code, then mimic threads by running the search for a bit, then storing its (manually built) continuation, handling possible new input, then resuming the search via this continuation. An approach that solves a similar problem using a very different approach is node.js — a JavaScript-based server where all IO is achieved via functions that receive callback functions, resulting in a style of code that is essentially writing CPSed code. For example, it is similar in principle to write code like:

```
;; copy "foo" to "tmp", read a line, delete "tmp", log the line
(copy-file "foo" "tmp"
  (lambda ()
    (read-line "tmp"
      (lambda (line)
        (delete-file "tmp"
          (lambda ()
            (log-line line
              (lambda ()
                (printf "All done.\n")))))))))
```

or a concrete node.js example — to swap two files, you could write:

```
function swap(path1, path2, callback) {
  fs.rename(path1, "temp-name",
    function() {
      fs.rename(path2, path1,
        function() {
          fs.rename("temp-name", path2, callback);
        });
    });
}
```

and if you want to follow the convention of providing a “convenient” synchronous version, you would also add:

```
function swapSync(path1, path2) {
  fs.renameSync(path1, "temp-name");
  fs.renameSync(path2, path1);
  fs.renameSync("temp-name", path2);
}
```

As we have seen in the web server application example, this style of programming tends to be “infectious”, where a function that deals with these callback-based functions will itself consume a callback —

```
;; abstract the above as a function
(define (safe-log-line in-file callback)
  (copy-file in-file "tmp"
    (lambda ()
      ... (log-line line callback))))
```

You should be able to see now what is happening here, without even mentioning the word “continuation” in the docs… See also this **Node vs Apache** video and read this **extended JS rant**. Quote:

> *No one ever for a second thought that a programmer would write actual code like that. And then Node came along and all of the sudden here we are pretending to be compiler back-ends. Where did we go wrong?*

> *(Actually, JavaScript has gotten better with promises, and then even better with `async`/`await` — but these are new, so it is actually common to find libraries that provide two such versions and a third promise-based one, and even a fourth one, using `async`. See for example the `replace-in-file` package on NPM.)*

Finally, as mentioned a few times, there has been extensive research into many aspects of continuations. Different CPS approaches, different implementation strategies, **a zoo-full of control operators**, assigning types to continuation-related functions and connections between continuations and types, even connections between CPS and certain proof techniques. Some research is still going on, though not as hot as it was — but more importantly, many modern languages "discover" the utility of having continuations, sometimes in some more limited forms (eg, Python and JavaScript generators), and sometimes in full form (eg, Ruby's `callcc`).

Sidenote: JavaScript: Continuations, Promises, and Async/Await

JavaScript went through a long road that is directly related to continuations. The motivating language feature that started all of this is the early decision to make the language single-threaded. This feature does make code easier to handle WRT side effects of all kinds. For example, consider:

```
let x = 0;
x = x + 1;
console.log(`A. x = ${x}`);
x = x + 1;
console.log(`B. x = ${x}`);
```

In JS, this code is guaranteed to show exactly the two output lines, with the expected values of 1 and 2. This is not guaranteed in any language that has threads (or any form of parallelism), since other threads might kick in at any point, possibly producing more output or mutating the value of `x`.

For several years JS was used only as a lightweight scripting language for simple tasks on web pages, and life was simple with the single threaded-ness nature of the language. Note that the environment *around* the language (= the browser) was very early quite thread-heavy, with different threads used for text layout, rendering, loading, user interactions, etc. But then came Node and JS was used for more traditional uses, with a web server being one of the first use cases. This meand that there was a a need for some way to handle "multiple threads" — it's impractical to wait for any server-side interaction to be done before serving the next.

This was addressed by designing Node as a program that makes heavy use of callback functions and IO events. In cases where you really want a linear sequence of operations you can use `*Sync` functions, such as the one we've seen above:

```
function swapSync(path1, path2) {
  fs.renameSync(path1, "temp-name");
  fs.renameSync(path2, path1);
  fs.renameSync("temp-name", path2);
}
```

But you can also use the non-`Sync` version which allows to switch between different active jobs — translating the above to (ignoring errors for simplicity):

```
function swap(path1, path2) {
  fs.rename(path1, "temp-name", () => {
    fs.rename(path2, path1, () => {
      fs.rename("temp-name", path2);
    });
  });
}
```

This could be better in some cases — for example, if the filesystem is a remote-mounted directory, each rename will likely take noticeable time. Using callbacks means that after firing each rename request the whole process is free to do any other work, and later resume the execution with the next operation when the last request is done.

But there is a major problem with this code: we can be more efficient since the main process can do anything while waiting for the three operations, but if you want to run this function to swap two files then you likely have some code that needs to run after executing it — code that expects the name swapping to have happened:

```
swap("foo", "bar");
... more code ...
```

The problem is that the function call returns *immediately*, and just registers the first rename operation. There might be a small chance (*very* unlikely) for the first rename to have happened, but even if it happens fast, the callback is *guaranteed* to not be called (again, since the core language is single-threaded), making this code completely broken.

This means that for such uses we *must* switch to using callbacks in the main function too:

```
function swap(path1, path2, cb) {
  fs.rename(path1, "temp-name", () => {
    fs.rename(path2, path1, () => {
      fs.rename("temp-name", path2, () =>
        cb()); // or just use cb directly
    });
  });
}
```

And the uses too:

```
swap("foo", "bar", () => {
  ... more code ...
});
```

This is the familiar manual CPS-ing of code that was for many years something that all JS programmes needed to use.

As we've seen, the problem here is that writing such code is difficult, and even more so working with such code when it needs to be extended, debugged, etc. Eventually, the JS world settled on using "promises" to simplify all of this: the idea is that instead of building "towers of callbacks", we abstract them as promises that are threaded using a `.then` property for the chain of operations.

```
function swapPromise(path1, path2) {
  return fsPromises.rename(path1, "temp-name")
           .then(() => fsPromises.rename(path2, path1))
           .then(() => fsPromises.rename("temp-name", path2));
}
```

Note that here too, the changes in the function body are reflated in how the whole function is used: the body uses promises, and therefore the whole function returns a promise.

This is a little better in that we don't suffer from nested code towers, but it's still not too convenient. The last step in this evolution was the introduction of `async`/`await` — language features that declare a promise-ified function which can wait for promises to be executed in a more-or-less plain looking code:

```
async function swap(path1, path2) {
  await fsPromises.rename(path1, "temp-name");
  await fsPromises.rename(path2, path1);
  await fsPromises.rename("temp-name", path2);
}
```

(The implementation of this feature is interesting: it relies on using generator functions, and using the fact that you can send values *into* a generator when you resume it — and all of this is, roughly speaking, used to implement a threading system.)

This looks almost like normal code, but note that while doing so we lose the simplification of a single-threaded language. Uses of `await` are equivalent to function calls that can switch the context to other code which might introduce unexpected side-effects. This is not really getting back to a proper multi-threaded language like Racket: it's rather a language with "cooperative threads". You still have the advantage that if you *don't* use `await`, then code executes sequentially and uninterrupted.

Whether it is useful or not to complicate these cases just to be able to write non-async code is a question…