PL: Lecture #28

Tuesday, December 3rd

Highlevel Overview on Continuations

Very roughly speaking, the transformation we made turns a function call like

```
(...stuff... (f ...args...) ...more-stuff...)
```

into

```
(f/k ...args...
     (lambda (<*>)
       (...stuff... <*> ...more-stuff...)))
```

This is the essence of the solution to the statelessness problem: to remember where we left off, we conveniently flip the expression inside-out so that its context is all stored in its continuation. One thing to note is that we did this only for functions that had some kind of web interaction, either directly or indirectly (since in the indirect case they still need to carry around the continuation).

If we wanted to make this process a completely mechanical one, then we wouldn't have been able to make this distinction. After all, a function like `map` is perfectly fine as it is, unless it happens to be used with a continuation-carrying function — and that's something that we know only at runtime. We would therefore need to transform *all* function calls as above, which in turn means that all functions would need to get an extra continuation argument.

Here are a few things to note about such fully-transformed code:

- All function calls in such code are tail calls. There is no single call with some context around it that is left for the time when the call is done. This is the exact property that makes it useful for a stateless interaction: such contexts are bad since a web interaction will mean that the context is discarded and lost. (In our pretend system, this is done by throwing an error.) Having no non-tail context means that capturing the continuation argument is sufficient, and no context gets lost.
- An implication of this, when you consider how the language is implemented, is that there is no need to have anything "on the stack" to execute fully transformed code. (If you'd use the stepper on such code, there would be no accumulation of context.) So is this some radical new way of computing without a stack? Not really: if you think about it, continuation arguments hold the exact same information that is traditionally put on the stack. (There is therefore an interesting relationship between continuations and runtime stacks, and in fact, one way of making it possible to grab continuations without doing such a transformation is to capture the current stack.)
- The evaluation order is fixed. Obviously, if Racket guarantees a left-to-right evaluation, then the order is always fixed — but in the fully transformed code there are no function calls where this makes any difference. If Racket were to change, the transformed code would still retain the order it uses. More specifically, when we do the transformation, we control the order of evaluation by choosing how to proceed at every point. For example, if we have:

  ```
  (...stuff... (f1 ...args...) (f2 ...args...) ...more-stuff...)
  ```

  then it's up to use to choose whether to pull `f1` first, or maybe we'd want to start with `f2`.

  But there's more: the resulting code is independent of the evaluation strategy of the language. Even if the language is lazy, the transformed code is still executing things in the same order. (Alternatively, we could convert things so that the resulting computation corresponds to a lazy evaluation strategy even in a strict language.)
- In other words, the converted code is completely sequential. The conversion process requires choosing left-to-right or delaying some evaluations (or all), but the resulting code is free from any of these and has

exactly one specific (sequential) order. You can therefore see how this kind of transformation is something that a compiler would want to do, since the resulting sequential code is easier for execution on a sequential base (like machine code, or C code). Another way to see this is that we have explicit names for each and every intermediate result — so the converted code would have a direct mapping between identifiers and machine registers (unlike “plain” code where some of these are implicit and compilation needs to make up names).

- The transformation is a *global* one. Not only do we have to transform the first top-level expression that makes up the web application, we also need to convert every function that is mentioned in the code, and in functions that those functions mentioned, etc. Even worse, the converted code is very different from the original version, since everything is shuffled around — in a way that matches the sequential execution, but it’s very hard to even see the original intention through all of these explicit continuations and the new intermediate result names.

  The upshot of this is that it’s not really something that we need to do manually, but instead we’d like it to be done automatically for us, by the compiler of the language.

What we did here is the tedious way of getting continuations: we basically implemented them by massaging our code, turning it inside-out into code with the right shape. The problem with this is that the resulting code is no longer similar to what we had originally written, which makes it more difficult to debug and to maintain. We therefore would like to have this done in some automatic way, ideally in a way that means that we can leave our plain original code as is.

An Automatic Continuation Converter

> ☞ ***PLAI §18***

The converted code that we produced manually above is said to be written in “Continuation Passing Style”, or CPS. What we’re looking for is for a way to generate such code automatically — a way to “CPS” a given source code. When you think about it, this process is essentially a source to source function which should be bolted onto the compiler or evaluator. In fact, if we want to do this in Racket, then this description makes it sound a lot like a macro — and indeed it could be implemented as such.

> *Note that “CPS” has two related but distinct meanings here: you could have code that is written “in CPS style”, which means that it handles explicit continuations. Uses of this term usually refer to using continuation functions in some places in the code, not for fully transformed code. The other meaning is used for fully-CPS-ed code, which is almost never written directly. In addition, “CPS” is often used as a verb — either the manual process of refactoring code into passing some continuations explicitly (in the first case), or the automatic process of fully converting code (in the second one).*

Before we get to the actual implementation, consider how we did the translation — there was a difference in how we handled plain top-level expressions and library functions. In addition, we had some more discounts in the manual process — one such discount was that we didn’t treat all value expressions as possible computations that require conversion. For example, in a function application, we took the function sub-expression as a simple value and left it as is, but for an automatic translation we need to convert that expression too since it might itself be a more complicated expression.

Instead of these special cases and shortcuts, we’ll do something more uniform: we will translate *every* expression into a function. This function will accept a receiver (= a continuation) and will pass it the value of the expression. This will be done for *all* expressions, even simple ones like plain numbers, for example, we will translate the `5` expression into (lambda (k) (k 5)), and the same goes for other constants and plain identifiers. Since we’re specifying a transformation here, we will treat it as a kind of a meta function and use a `CPS[x]` to make it easy to talk about:

```
CPS[5]
-->
(lambda (k) (k 5)) ; same for other numbers and constants

CPS[x]
-->
(lambda (k) (k x)) ; same as above for identifiers too
```

When we convert a primitive function application, we still do the usual thing, which is now easier to see as a general rule — using `CPS[?]` as the meta function that does the transformation:

```
CPS[(+ E1 E2)]
-->
(lambda (k)          ; everything turns to cont.-consuming functions
  (CPS[E1]           ; the CPS of E1 -- it expects a cont. argument
   (lambda (v1)      ; send this cont. to CPS[E1], so v1 is its value
     (CPS[E2]        ; same for E2 -- expects a cont.
      (lambda (v2)   ; and again, v2 becomes the value of E2
        (k (+ v1 v2))))))) ; finally return the sum to our own cont.
```

In the above, you can see that (CPS[E] (lambda (v) …)) can be read as “evaluate `E` and bind the result to `v`”. (But note that the CPS conversion is not doing any evaluation, it just reorders code to determine how it gets evaluated when it later runs — so “compute” might be a better term to use here.) With this in mind, we can deal with other function applications: evaluate the function form, evaluate the argument form, then apply the first value on the second value, and finally wrap everything with a (lambda (k) …) and return the result to this continuation:

```
CPS[(E1 E2)]
-->
(lambda (k)
  (CPS[E1]           ; bind the result of evaluating E1
   (lambda (v1)      ; to v1
     (CPS[E2]        ; and the result of evaluating E2
      (lambda (v2)   ; to v2
        (k (v1 v2))))))) ; apply and return the result
```

But this is the rule that we should use for primitive non-continuation functions only — it’s similar to what we did with `+` (except that we skipped evaluating `+` since it’s known). Instead, we’re dealing here with functions that are defined in the “web language” (in the code that is being converted), and as we’ve seen, these functions get a `k` argument which they use to return the result to. That was the whole point: pass `k` on to functions, and have them return the value directly to the `k` context. So the last part of the above should be fixed:

```
CPS[(E1 E2)]
-->
(lambda (k)
  (CPS[E1]           ; bind the result of evaluating E1
   (lambda (v1)      ; to v1
     (CPS[E2]        ; and the result of evaluating E2
      (lambda (v2)   ; to v2
        (v1 v2 k)))))) ; apply and have it return the result to k
```

There’s a flip side to this transformation — whenever a function is created with a `lambda` form, we need to add a `k` argument to it, and make it return its value to it. Then, we need to “lift” the whole function as usual, using the same transformation we used for other values in the above. We’ll use `k` for the latter continuation argument, and `cont` for the former:

```
CPS[(lambda (arg) E)]
-->
```

```
(lambda (k) ; this is the usual
  (k        ; lifting of values
   (lambda (arg cont) ; the translated function has a cont. input
     (CPS[E] cont)))) ; the translated body returns its result to it
```

It is interesting to note the two continuations in the translated result: the first one (using `k`) is the continuation for the function value, and the second one (using `cont`) is the continuation used when the function is applied. Comparing this to our evaluators — we can say that the first roughly corresponds to evaluating a function form to get a closure, and the second corresponds to evaluating the body of a function when it's applied, which means that `cont` is the dynamic continuation that matches the dynamic context in which the function is executed. Inspecting the CPS-ed form of the identity function is unsurprising: it simply passes its first argument (the "real" one) into the continuation since that's how we return values in this system:

```
CPS[(lambda (x) x)]
-->
(lambda (k)
  (k
   (lambda (x cont)
     (CPS[x] cont))))
-->
(lambda (k)
  (k
   (lambda (x cont)
     ((lambda (k) (k x)) cont))))
--> ; reduce the redundant function application
(lambda (k)
  (k
   (lambda (x cont)
     (cont x))))
```

Note the reduction of a trivial application — doing this systematic conversion leads to many of them.

We now get to the transformation of the form that is the main reason we started with all of this — `web-read`. This transformation is simple, it just passes along the continuation to `web-read/k`:

```
CPS[(web-read E)]
-->
(lambda (k)
  (CPS[E]           ; evaluate the prompt expression
   (lambda (prompt) ; and bind it to prompt
     (web-read/k prompt k)))) ; use the prompt and the current cont.
```

We also need to deal with `web-display` — we changed the function calling protocol by adding a continuation argument, but `web-display` is defined outside of the CPS-ed language so it doesn't have that argument. Another way of fixing it could be to move its definition into the language, but then we'll still need to have a special treatment for the `error` that it uses.

```
CPS[(web-display E)]
-->
(lambda (k)
  (CPS[E]          ; evaluate the expression
   (lambda (val)   ; and bind it to val
     (web-display val))))
```

As you can see, all of these transformations are simple rewrites. We can use a simple `syntax-rules` macro to implement this transformation, essentially creating a DSL by translating code into plain Racket. Note that in the specification above we've implicitly used some parts of the input as keywords — `lambda`, `+`, `web-read`, and `define` — this is reflected in the macro code. The order of the rules is important, for example, we need to match first on (web-read E) and then on the more generic (E1 E2), and we ensure that the last default lifting of values has a simple expression by matching on (x …) before that.

```
(define-syntax CPS
  (syntax-rules (+ lambda web-read web-display) ;*** keywords
    [(CPS (+ E1 E2))
     (lambda (k)
       ((CPS E1)
        (lambda (v1)
          ((CPS E2)
           (lambda (v2)
             (k (+ v1 v2)))))))]
    [(CPS (web-read E))
     (lambda (k)
       ((CPS E)
        (lambda (val)
          (web-read/k val k))))]
    [(CPS (web-display E))
     (lambda (k)                ; could be:
       ((CPS E)                 ;     (lambda (k)
        (lambda (val)           ;       ((CPS E) web-display))
          (web-display val))))] ; but keep it looking uniform
    [(CPS (lambda (arg) E))
     (lambda (k)
       (k (lambda (arg cont)
            ((CPS E)
             cont))))]
    [(CPS (E1 E2))
     (lambda (k)
       ((CPS E1)
        (lambda (v1)
          ((CPS E2)
           (lambda (v2)
             (v1 v2 k))))))]
    ;; the following pattern ensures that the last rule is used only
    ;; with simple values and identifiers
    [(CPS (x ...))
     ---syntax-error---]
    [(CPS V)
     (lambda (k) (k V))]))
```

The transformation that this code implements is one of the oldest CPS transformations — it is called the Fischer Call by Value CPS transformation, and is due Michael Fischer. There has been much more research into such transformations — the Fischer translation, while easy to understand due to its uniformity, introduces significant overhead in the form of many new functions in its result. Some of these are easy to optimize — for example, things like ((lambda (k) (k v)) E) could be optimized to just (E v) assuming a left-to-right evaluation order or proving that E has no side-effects (and Racket performs this optimization and several others), but some of the overhead is not easily optimized. There have been several other CPS transformations, in an attempt to avoid such overhead.

Finally, trying to run code using this macro can be a little awkward. We need to explicitly wrap all values in definitions by a `CPS`, and we need to invoke top-level expressions with a particular continuation — `web-display` in our context. We can do all of that with a convenience macro that will transform a number of definitions followed by an optional expression.

> *Note the use of `begin` — usually, it is intended for sequential execution, but it is also used in macro result expressions when we need a macro to produce multiple expressions (since the result of a macro must be a single S-expression) — this is why it's used here, not for sequencing side-effects.*

```
(define-syntax CPS-code
  (syntax-rules (define)
    [(CPS-code (define (id arg) E) more ...)
     ;; simple translation to `lambda'
     (CPS-code (define id (lambda (arg) E)) more ...)]
    [(CPS-code (define id E) more ...)
     (begin (define id ((CPS E) (lambda (x) x)))
            (CPS-code more ...))]
    [(CPS-code last-expr)
     ((CPS last-expr) web-display)]
    [(CPS-code) ; happens when there is no plain expr at
     (begin)])) ; the end so do nothing in this case
```

The interesting thing that this macro does is set up a proper continuation for definitions and top-level expressions. In the latter case, it passes `web-display` as the continuation, and in the former case, it passes the identity function as the continuation — which is used to "lower" the lifted value from its continuation form into a plain value. Using the identity function as a continuation is not really correct: it means that if evaluating the expression to be bound performs some web interaction, then the definition will be aborted, leaving the identifier unbound. The way to solve this is by arranging for the definition operation to be done in the continuation, for example, we can get closer to this using an explicit mutation step:

```
    [(CPS-code (define id E) more ...)
     (begin (define id #f)
            ((CPS E) (lambda (x) (set! id x)))
            (CPS-code more ...))]
```

But there are two main problems with this: first, the rest of the code — `(CPS-code more ...)` — should also be done in the continuation, which will defeat the global definitions. We could try to use the continuation to get the scope:

```
    [(CPS-code (define id E) more ...)
     ((CPS E) (lambda (id) (CPS-code more ...)))]
```

but that breaks recursive definitions. In any case, the second problem is that this is not accurate even if we solved the above: we really need to have parts of the Racket definition mechanism exposed to make it work. So we'll settle with the simple version as an approximation. It works fine if we use definitions only for functions, and invoke them in toplevel expressions.

For reference, the complete code at this point follows.

```
;; Simulation of web interactions with a CPS converter (not an
;; interpreter)

#lang racket

(define error raise-user-error)

(define (nothing-to-do ignored)
  (error 'nothing-to-do "No computation to resume."))

(define resumer (box nothing-to-do))

(define (web-display n)
  (set-box! resumer nothing-to-do)
  (error 'web-display "~s" n))

(define (web-read/k prompt k)
  (set-box! resumer k)
  (error 'web-read
         "enter (submit N) to continue the following\n  ~a:"
         prompt))

(define (submit n)
  ;; to avoid mistakes, we clear out `resumer' before invoking it
  (let ([k (unbox resumer)])
    (set-box! resumer nothing-to-do)
    (k n)))

(define-syntax CPS
  (syntax-rules (+ lambda web-read web-display) ;*** keywords
    [(CPS (+ E1 E2))
     (lambda (k)
       ((CPS E1)
        (lambda (v1)
          ((CPS E2)
           (lambda (v2)
             (k (+ v1 v2)))))))]
    [(CPS (web-read E))
     (lambda (k)
       ((CPS E)
        (lambda (val)
          (web-read/k val k))))]
    [(CPS (web-display E))
     (lambda (k)
       ((CPS E)
        (lambda (val)
          (web-display val))))]
    [(CPS (lambda (arg) E))
     (lambda (k)
       (k (lambda (arg cont)
            ((CPS E)
             cont))))]
    [(CPS (E1 E2))
     (lambda (k)
       ((CPS E1)
        (lambda (v1)
          ((CPS E2)
           (lambda (v2)
             (v1 v2 k))))))]
    ;; the following pattern ensures that the last rule is used only
    ;; with simple values and identifiers
    [(CPS (x ...))
     ---syntax-error---]
    [(CPS V) ; <-- only numbers, other literals, and identifiers
     (lambda (k)
       (k V))]))

(define-syntax CPS-code
  (syntax-rules (define)
    [(CPS-code (define (id arg) E) more ...)
     ;; simple translation to `lambda'
     (CPS-code (define id (lambda (arg) E)) more ...)]
    [(CPS-code (define id E) more ...)
     (begin (define id ((CPS E) (lambda (x) x)))
            (CPS-code more ...))]
    [(CPS-code last-expr)
     ((CPS last-expr) web-display)]
    [(CPS-code) ; happens when there is no plain expr at
     (begin)])) ; the end so do nothing in this case
```

Here is a quick example of using this:

```
(CPS-code
  (web-display (+ (web-read "First number")
                  (web-read "Second number"))))
```

Note that this code uses `web-display`, which is not really needed since `CPS-code` would use it as the top-level continuation. (Can you see why it works the same either way?) So this is even closer to a plain program:

```
(CPS-code (+ (web-read "First number")
             (web-read "Second number")))
```

A slightly more complicated example:

```
(CPS-code
  (define (add n)
    (lambda (m)
      (+ m n)))
  (define (read-and-add n)
    ((add n) (web-read "Another number")))
  (read-and-add (web-read "A number")))
```

Using this for the other examples is not possible with the current state of the translation macro. These example will require extending the CPS transformation with functions of any arity, multiple expressions in a body, and it recognize additional primitive functions. None of these is difficult, it will just make it more verbose.

Continuations as a Language Feature

> *This is conceptually between ☞ **PLAI §18** and ☞ **PLAI §19***

In the list of CPS transformation rules there were two rules that deserve additional attention in how they deal with their continuation.

First, note the rule for `web-display`:

```
[(CPS (web-display E))
 (lambda (k)
   ((CPS E)
    (lambda (val)
      (web-display val))))]
```

— it simply ignores its continuation. This means that whenever `web-display` is used, the rest of the computation is simply discarded, which seems wrong — it's the kind of problem that we've encountered several times when we discussed the transforming web application code. Of course, this doesn't matter much for our implementation of `web-display` since it aborts the computation anyway using `error` — but what if we did that intentionally? We would get a kind of an "abort now" construct: we can implement this as a new `abort` form that does just that:

```
(define-syntax CPS
  (syntax-rules (...same... abort) ;*** new keyword
    ...
    [(CPS (abort E))
     (lambda (k)
       ((CPS E) (lambda (x) x)))] ; ignore `k'
    ...))
```

You could try that — (CPS-code (+ 1 2)) produces 3 as “web output”, but (CPS-code (+ 1 (abort 2))) simply returns 2. In fact, it doesn’t matter how complicated the code is — as soon as it encounters an `abort` the whole computation is discarded and we immediately get its result, for example, try this:

```
(CPS-code
  (define (add n)
    (lambda (m)
      (+ m n)))
  (define (read-and-add n)
    ((abort 999) ((add n) (web-read "Another number"))))
  (read-and-add (web-read "A number")))
```

it reads the first number and then it immediately returns 999. This seems like a potentially useful feature, except that it’s a little too “untamed” — it aborts the program completely, getting all the way back to the top-level with a result. (It’s actually quite similar to throwing an exception, only without a way to catch it.) It would be more useful to somehow control the part of the computation that gets aborted instead.

That leads to the second exceptional form in our translator: `web-read`. If you look closely at all of our transformation rules, you’ll see that the continuation argument is never made accessible to user code — the `k` argument is always generated by the macro (and inaccessible to user code due to the hygienic macro system). The continuation is only passed as the extra argument to user functions, but in the rule that adds this argument:

```
    [(CPS (lambda (arg) E))
     (lambda (k)
       (k (lambda (arg cont)
            ((CPS E)
             cont))))]
```

the new `cont` argument is introduced by the macro so it is inaccessible as well. The only place where the `k` argument is actually used is in the `web-read` rule, where it is sent to the resulting `web-read/k` call. (This makes sense, since web reading is how we mimic web interaction, and therefore it is the only reason for CPS-ing our code.) However, in our fake web framework this function is a given built-in, so the continuation is still not accessible for user code.

What if we pass the continuation argument to a user function in a way that *intentionally* exposes it? We can achieve this by writing a function that is similar to `web-read/k`, except that it will somehow pass the continuation to user code. A simple way to do that is to have the new function take a function value as its primary input, and call this function with the continuation (which is still received as the implicit second argument):

```
(define (call-k f k)
  (f k))
```

This is close, but it fails because it doesn’t follow our translated function calling protocol, where every function receives two inputs — the original argument and the continuation. Because of this, we need to call `f` with a second continuation value, which is `k` as well:

```
(define (call-k f k)
  (f k k))
```

But we also fail to follow the calling protocol by passing `k` as is: it is a continuation value, which in our CPS system is a one-argument function. (In fact, this is another indication that continuations are not accessible to user code — they don’t follow the same function calling protocol.) It is best to think about continuations as *meta* values that are not part of the user language just yet. To make it usable, we need to wrap it so we get the usual two-argument function which user code can call:

```
(define (call-k f k)
  (f (lambda (val cont) (k val)) k))
```

This explicit wrapping is related to the fact that continuations are a kind of meta-level value — and the wrapping is needed to "lower" it to the user's world. (This is sometimes called "reification": a meta value is *reified* as a user value.)

Using this new definition, we can write code that can access its own continuation as a plain value. Here is a simple example that grabs the top-level continuation and labels it `abort`, then uses it in the same way we've used the above `abort`:

```
> (CPS-code (call-k (lambda (abort) (+ 1 (abort 2)))))
web-display: 2
```

But we can grab any continuation we want, not just the top-level one:

```
(CPS-code (+ 100 (call-k (lambda (abort) (+ 1 (abort 2))))))
web-display: 102
```

Side note: how come we didn't need a new CPS translation rule for this function? There is no need for one, since `call-k` is already written in a way that follows our calling convention, and no translation rule is needed. In fact, no such rule is needed for `web-read` too — except for changing the call to `web-read/k`, it does exactly the same thing that a function call does, so we can simply rename `web-read/k` as `web-read` and drop the rule. (Note that the rewritten function call will have a (CPS web-read) — but CPS-ing an identifier results in the identifier itself.) The same holds for `web-display` — we just need to make it adhere to the calling convention and add a `k` input which is ignored. One minor complication is that `web-display` is also used as a continuation value for a top-level expression in `CPS-code` — so we need to wrap it there.

The resulting code follows:

```
#lang racket

(define error raise-user-error)

(define (nothing-to-do ignored)
  (error 'nothing-to-do "No computation to resume."))

(define resumer (box nothing-to-do))

(define (web-display n k) ; note that k is not used!
  (set-box! resumer nothing-to-do)
  (error 'web-display "~s" n))

(define (web-read prompt k)
  (set-box! resumer k)
  (error 'web-read
         "enter (submit N) to continue the following\n  ~a:"
         prompt))

(define (submit n)
  ;; to avoid mistakes, we clear out `resumer' before invoking it
  (let ([k (unbox resumer)])
    (set-box! resumer nothing-to-do)
    (k n)))

(define (call-k f k)
  (f (lambda (val cont) (k val)) k))

(define-syntax CPS
  (syntax-rules (+ lambda)
    [(CPS (+ E1 E2))
     (lambda (k)
       ((CPS E1)
        (lambda (v1)
          ((CPS E2)
           (lambda (v2)
             (k (+ v1 v2)))))))]
    [(CPS (lambda (arg) E))
     (lambda (k)
       (k (lambda (arg cont)
            ((CPS E)
             cont))))]
    [(CPS (E1 E2))
     (lambda (k)
       ((CPS E1)
        (lambda (v1)
          ((CPS E2)
           (lambda (v2)
             (v1 v2 k))))))]
    ;; the following pattern ensures that the last rule is used only
    ;; with simple values and identifiers
    [(CPS (x ...))
     ---syntax-error---]
    [(CPS V)
     (lambda (k)
       (k V))]))

(define-syntax CPS-code
  (syntax-rules (define)
    [(CPS-code (define (id arg) E) more ...)
     ;; simple translation to `lambda'
     (CPS-code (define id (lambda (arg) E)) more ...)]
    [(CPS-code (define id E) more ...)
     (begin (define id ((CPS E) (lambda (x) x)))
            (CPS-code more ...))]
    [(CPS-code last-expr)
     ((CPS last-expr) (lambda (val) (web-display val 'whatever)))]
    [(CPS-code) ; happens when there is no plain expr at
     (begin)])) ; the end so do nothing in this case
```

Obviously, given `call-k` we could implement `web-read/k` in user code: `call-k` makes the current continuation available and going on from there is simple (it will require a little richer language, so we will do that in a minute). In fact, there is no real reason to stick to the fake web framework to play with continuations. (Note: since we don't throw an error to display the results, we can also allow multiple non-definition expressions in `CPS-code`.)

```
 ;; A language that is CPS-transformed (not an interpreter)

 #lang racket

 (define (call-k f k)
   (f (lambda (val cont) (k val)) k))

 (define-syntax CPS
   (syntax-rules (+ lambda)
     [(CPS (+ E1 E2))
      (lambda (k)
        ((CPS E1)
         (lambda (v1)
           ((CPS E2)
            (lambda (v2)
              (k (+ v1 v2)))))))]
     [(CPS (lambda (arg) E))
      (lambda (k)
        (k (lambda (arg cont)
             ((CPS E)
              cont))))]
     [(CPS (E1 E2))
      (lambda (k)
        ((CPS E1)
         (lambda (v1)
           ((CPS E2)
            (lambda (v2)
              (v1 v2 k))))))]
     ;; the following pattern ensures that the last rule is used only
     ;; with simple values and identifiers
     [(CPS (x ...))
      ---syntax-error---]
     [(CPS V)
      (lambda (k)
        (k V))]))

 (define-syntax CPS-code
   (syntax-rules (define)
     [(CPS-code (define (id arg) E) more ...)
      ;; simple translation to `lambda'
      (CPS-code (define id (lambda (arg) E)) more ...)]
     [(CPS-code (define id E) more ...)
      (begin (define id ((CPS E) (lambda (x) x)))
             (CPS-code more ...))]
     [(CPS-code expr more ...)
      (begin ((CPS expr) (lambda (x) x))
             (CPS-code more ...))]
     [(CPS-code) (begin)])) ; done

 (CPS-code (call-k (lambda (abort) (+ 1 (abort 2))))
           (+ 100 (call-k (lambda (abort) (+ 1 (abort 2))))))
```

Continuations in Racket

As we have seen, CPS-ing code makes it possible to implement web applications with a convenient interface. This is fine in theory, but in practice it suffers from some problems. Some of these problems are technicalities: it relies on proper implementation of tail calls (since all calls are tail calls), and it represents the computation stack as a chain of closures and therefore prevents the usual optimizations. But there is one problem that is much more serious: it is a *global* transformation, and as such, it requires access to the complete program code. As an example, consider how `CPS-code` deals with definitions: it uses an identity function as the continuation, but that wasn't the proper way to do them, since it would break if computing the value performs some web interaction. A good solution would instead put the side-effect that `define` performs in the continuation — but this side effect is not even available for us when we work inside Racket.

Because of this, the proper way to make continuations available is for the language implementation itself to provide it. There are a few languages that do just that — and Scheme has pioneered this as part of the core requirements that the standard dictates: a Scheme implementation needs to provide `call-with-current-continuation`, which is the same tool as our `call-k`. Usually it is also provided with a shorter name, `call/cc`. Here are our two examples, re-done with Racket's built-in `call/cc`:

```
(call/cc (lambda (abort) (+ 1 (abort 2))))
(+ 100 (call/cc (lambda (abort) (+ 1 (abort 2)))))
```

[Side note: continuations as we see here are still provided only by a few “fringe” functional languages. However, they are slowly making their way into more mainstream languages — Ruby has these continuations too, and several other languages provide more limited variations, like generators in Python. On the other hand, Racket provides a much richer functionality: it has delimited continuations (which represents only a part of a computation context), and its continuations are also composable — a property that goes beyond what we see here.]

Racket also comes with a more convenient `let/cc` form, which exposes the “grab the current continuation” pattern more succinctly — it’s a simple macro definition:

```
(define-syntax-rule (let/cc k body ...)
  (call/cc (lambda (k) body ...)))
```

and the two examples become:

```
(let/cc abort (+ 1 (abort 2)))
(+ 100 (let/cc abort (+ 1 (abort 2))))
```

When it gets to choosing an implementation strategy, there are two common approaches: one is to do the CPS transformation at the compiler level, and another is to capture the actual runtime stack and wrap it in an applicable continuation objects. The former can lead to very efficient compilation of continuation-heavy programs, but the latter makes it easier to deal with foreign functions (consider higher order functions that are given as a library where you don’t have its source) and allows using the normal runtime stack that CPUs are using very efficiently. Racket implements continuations with the latter approach mainly for these reasons.

To see how these continuations expose some of the implementation details that we normally don’t have access to, consider grabbing the continuation of a definition expression:

```
> (define b (box #f))
> (define a (let/cc k (set-box! b k) 123))
> a
123
> ((unbox b) 1000)
> a
1000
```

> *Note that using a top-level (let/cc abort …code…) is not really aborting for a reason that is related to this: a true `abort` must capture the continuation before any computation begins. A natural place to do this is in the REPL implementation.*

Finally, we can use these to re-implement our fake web framework, using Racket’s continuations instead of performing our own transformation. The only thing that requires continuations is our `web-read` — and using the Racket facilities we can implement it as follows:

```
(define (web-read prompt) ; no `k' input
  (let/cc k ; instead, get it with `let/cc'
    ;; and the body is the same as it was
    (set-box! resumer k)
    (error 'web-read
           "enter (submit N) to continue the following\n  ~a:"
           prompt)))
```

Note that this kind of an implementation is no longer a “language” — it is implemented as a plain library now, demonstrating the flexibility that having continuations in our language enables. While this is still just our fake toy framework, it is the core way in which the Racket web server is implemented (see the “addition server” implementation above), using a hash table that maps freshly made URLs to stored continuations. The complete code follows:

```
;; Simulation of web interactions with Racket's built-in
;; continuation facility

#lang racket

(define error raise-user-error)

(define (nothing-to-do ignored)
  (error 'nothing-to-do "No computation to resume."))

(define resumer (box nothing-to-do))

(define (web-display n)
  (set-box! resumer nothing-to-do)
  (error 'web-display "~s" n))

(define (web-read prompt)
  (let/cc k
    (set-box! resumer k)
    (error 'web-read
           "enter (submit N) to continue the following\n  ~a:"
           prompt)))

(define (submit n)
  ;; to avoid mistakes, we clear out `resumer' before invoking it
  (let ([k (unbox resumer)])
    (set-box! resumer nothing-to-do)
    (k n)))
```

Using this, you can try out some of the earlier examples, which now become much simpler since there is no need to do any CPS-ing. For example, the code that required transforming `map` into a `map/k` can now use the plain `map` directly. In fact, that’s the exact code we started that example with — no changes needed:

```
(define (sum l) (foldl + 0 l))
(define (square n) (* n n))
(define (read-number prompt)
  (web-read (format "~a number" prompt)))
(web-display (sum (map (lambda (prompt)
                         (square (read-number prompt)))
                       '("First" "Second" "Third"))))
```

Note how `web-read` is executed directly — it is a plain library function.