Continuations

This note provides some additional background on continuations, and their use in controlling the flow of computation within a process.

Continuations approach control flow differently than standard Scheme code. Rather than writing procedures that return values to their caller, we write procedures that accept arguments (also procedures) called continuations and call those continuations on their return values. For example, we might rewrite

```
(define (add-to-3 x)
  (+ 3 x))
```

as

```
(define (add-to-3 x cont)
  (cont (+ 3 x)))
```

The new definition of `add-to-three` doesn't return anything directly; instead, it passes the value to whatever the caller provided as `cont`.

Register Machine Code

Continuations really aren't a new idea. You saw a similar idea in the discussion of register machines. When you program in the register machine code, your procedures probably end in something similar to

```
  (goto (reg continue))
```

When we decided to start using the `continue` register (to allow our procedures to return to any caller), we were effectively adding another parameter to the procedure; that parameter was the continuation and we called it by executing `(goto (reg continue))`.

Why Continuations

Why might you use continuation passing style? Continuations give you greater control over the control flow of your program. For example, you can pass several continuations to a procedure :

```
(define (divide a b success fail)
    (if (= b 0)
      (fail "divide-by-zero")
      (success (/ a b))))
```

Without continuations, evaluating `(divide 17 0)` would terminate the computation and ask the user if they wanted to start the debugger. With continuations, we might pass a procedure as `fail` that prints an error message and restarts the computation.

Another aspect of continuations is that they represent all of the future computation that will happen on the return value of a procedure. Since they're being passed as variables, we can do neat things like save them, pass them to other procedures, or even call them several times with different values.

In the evaluator, continuations will provide a useful means of handling errors. If we hit some error condition, we can remember the current continuation, ask the user to fix or handle the error, and then call the saved continuation to resume the computation.

Some Practice with Continuations

The procedures above are pretty simple. Let's write some more interesting procedures using continuations.

```
(define (factorial n c)
  (if (= n 0)
      (c 1)
      (factorial (- n 1) (lambda (x) (c (* n x))))))
```

Here's our old friend factorial. Let's look at the base case – if `n` is zero, we just call the continuation on 1 (ie, call the continuation on the value that we would have returned). In the recursive case, we call `factorial` on `(- n 1)` and pass it a new continuation. That new continuation will be called on $(n-1)!$ (remember, that's the contract for continuations; they're called on the return value of the procedure). So our `(lambda (x) ...)` will be called on $(n-1)!$. It multiplies that by `n` and then passes that value to `c`, the original continuation.

Now how would we use such a procedure? (Note that the use of continuation style is independent of whether we bury this inside the evaluator as in the second part of the project.) Well, we just need to call the procedure with a value for `n`, and a procedure that tells us what to do with the result. For example

```
(factorial 10 (lambda (x) x))
;Value: 3628800
```

would just return the final value. But instead we might be fancier about how we print out results:

```
(factorial 10
   (lambda (x)
     (newline)
     (display "Factorial result is: ")
     (display x)))
```

would result in the following appearing on our monitor:

```
Factorial result is: 3628800
;Value: #[unspecified-return-value]
```

Here's a version of sum-interval using continuations (this sums the numbers from a to b inclusive):

```
(define (sum-interval a b c)
  (if (= a b)
      (c a)
      (sum-interval (+ a 1)
                    b
                    (lambda (x) (c (+ a x))))))
```

In the recursive case here, we call sum-interval on the interval from $a + 1$ to b. The new continuation is a procedure that adds the sum over the interval from $a + 1$ to b to a and passes that value to the original continuation.