Continuations

The Ultimate Control Structure
Suppose expression E contains a subexpression S

The continuation of S in E consists of all of the steps needed to complete E after the completion of S.

At any point during a computation the *current continuation* is the continuation of whatever expression is currently executing. Note that the current continuation is constantly changing.
For example,
(define fact (lambda (n) (if (= 0 n) 1 (* n (fact (- n 1))))))

Consider expression E: (printf "5! = ~s" (fact 5))

The continuation in E of (fact 5) is the call to printf.
The continuation in E of (fact 4) is "multiply the result by 5, then call printf".
The continuation of (fact 3) is "multiply the result by 4, multiply the result of that by 5, then call printf."
Note that the continuations become increasingly complex as we proceed through the recursion.
Now consider fact-acc, the accumulator version of fact:

\[
\text{(define fact-acc (lambda (n acc)
                 (if (= 0 n) acc (fact-acc (- n 1) (* n acc)))))}
\]

Let E be the expression \((\text{printf \"5! = \~s\" (fact-acc 5 1)})\)

The continuation of \((\text{fact-acc 5 1})\) is the printf statement.

The continuation of \((\text{fact-acc 4 5})\) is the printf statement.
Note that in this last example the continuation doesn't change as we go back through the recursion. The difference is that the accumulator version is tail-recursive and the original version is not.

The continuation of a deep recursion becomes more complex as the recursion progresses. The continuation of a tail recursion remains constant as the recursion progresses.
We can illustrate this using Scheme expressions to describe the continuation, with \( \Box \) representing the current expression. The \( \Box \) is called a "context" for the continuation.
For example, consider the expression

\( S: (\ast (\ast 2 5) (+ 3 8)) \)

If we let \( E_1 \) be all of \( S \), then \( C_1 \), the current continuation, is \( \Box \): do the whole computation and return it.

If \( E_2 \) is \( (\ast 2 5) \) then \( C_2 \), the continuation of \( E_2 \), is \( (\ast \Box (+ 3 8)) \)

If \( E_3 \) is \( (+ 3 8) \), the continuation of \( E_3 \) is \( C_3: (\ast 10 \Box) \).
The current subexpression and its continuation make up the current *execution state* of the computation.

The sequence of execution states shows the *dynamics* of the computation.
Ex. (define fact (lambda (n) (if (= 0 n) 1 (* n (fact (- n 1))))))

Here are the dynamics of (fact 3)

<table>
<thead>
<tr>
<th>Direction</th>
<th>Expression</th>
<th>Continuation</th>
<th>Result of Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>(fact 3)</td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>IN</td>
<td>(fact 2)</td>
<td>(* 3 □)</td>
<td>?</td>
</tr>
<tr>
<td>IN</td>
<td>(fact 1)</td>
<td>(* 3 (*2 □))</td>
<td>?</td>
</tr>
<tr>
<td>IN</td>
<td>(fact 0)</td>
<td>(* 3 (* 2 (* 1 □)))</td>
<td>?</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact 0)</td>
<td>(* 3 (* 2 (* 1 1)))</td>
<td>1</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact 1)</td>
<td>(* 3 (* 2 1))</td>
<td>1</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact 2)</td>
<td>(* 3 2)</td>
<td>2</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact 3)</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
(define fact-acc (lambda (n acc) 
  (if (= 0 n) acc (fact-acc (- n 1) (* n acc))))))

Dynamics of (fact 3 1)

<table>
<thead>
<tr>
<th>Direction</th>
<th>Expression</th>
<th>Continuation</th>
<th>Result of Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>(fact-acc 3 1)</td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>IN</td>
<td>(fact-acc 2 3)</td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>IN</td>
<td>(fact-acc 1 6)</td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>IN</td>
<td>(fact-acc 0 6)</td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact-acc 0 6)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact-acc 1 6)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact-acc 2 3)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact-acc 3 1)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Direction</td>
<td>Expression</td>
<td>Continuation</td>
<td>Result of Expression</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------</td>
<td>--------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>IN</td>
<td>(fact-acc 3 1)</td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>IN</td>
<td>(fact-acc 2 3)</td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>IN</td>
<td>(fact-acc 1 6)</td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>IN</td>
<td>(fact-acc 0 6)</td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact-acc 0 6)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact-acc 1 6)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact-acc 2 3)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact-acc 3 1)</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Note that in a system that handles tail recursions properly, the last three lines in this table can be omitted, since once you know that the continuation is a constant we know the whole value the expression will return as soon as you know the value of the current expression.
Many deep recursions can be converted to tail recursions by using accumulating parameters or continuation-passing style:

deeep: (define fact (lambda (n)
   (if (= 0 n) 1 (* n (fact (- n 1))))))
tail:  (define fact-acc (lambda (n acc)
   (if (= 0 n) acc (fact-acc (- n 1) (* n acc)))))

or (define fact-k (lambda (n k)
   (if (= 0 n) (k 1)
       (fact-k (- n 1)
           (lambda (t) (k (* n t)))))))
deep: (define fib (lambda (n)
    (if (= 0 n) 0
        (if (= 1 n) 1
            (+ (fib (- n 1)) (fib (- n 2)))))
    (if (= 0 n) (k 0)
        (if (= 1 n) (k 1)
            (fib-k (- n 1)
                (lambda (s)
                    (fib-k (- n 2)
                        (lambda (t) (k (+ s t))))))))
    (lambda (s)
        (fib-k (- n 2)
            (lambda (t) (k (+ s t)))))))
What happens if we try to add another condition:
  (fib 0) is 0
  (fib 1) is 1
  If we ever try to compute (fib 2) just give up and say 'gak

(see the "Fib.rkt" file)
We could add an "escape continuation" to any cps style function. This lets us leave the function with a particular answer if a condition is met. For example, here is a version of rember that removes atom x from lat; if lat ever contains the atom 'bob it exits with the result 'no-bob
(define rember-k (lambda (x lat k esc)
    (cond
        [(null? lat) (k null)]
        [(eq? (car lat) x) (rember-k x (cdr lat)
                                   (lambda (t) (k t)) esc)]
        [(eq? (car lat) 'bob) (esc 'bob)]
        [else (rember-k x (cdr lat)
                       (lambda (t) (k (cons (car lat) t)))
                       esc)]))

The escape continuation esc is just
    (lambda (t) 'no-bob)
Here is the same idea applied to the function that counts atoms in a general list. If the list contains the atom 'bob anywhere this function escapes with the value (esc 'bob):

\[
\text{(define count-k (lambda (L k esc)}
\text{ (cond}
\text{ [ (null? L) (k 0)]}
\text{ [ (atom? (car L))}
\text{ (if (eq? (car L) 'bob) (esc 'bob)
\text{ (count-k (cdr L) (lambda (t) (k (+ 1 t))) esc))]
\text{ [else (count-k (car L)
\text{ (lambda (s) (count-k (cdr L)
\text{ (lambda (t) (k (+ s t)))
\text{ esc))
\text{ esc)]))])))
\text{)}}
\text{)}}
\]