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:

```
(define fact-acc (lambda (n acc)
    (if (= 0 n) acc (fact-acc (- n 1) (* n acc)))))
```

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

The continuation of (fact-acc 5 1) is the printf statement.

The continuation of (fact-acc 4 5) is the printf statement

Note that in this last example the continuation doesn't change as we go 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 □ representing the current expression. The □ is called a "context" for the continuation.

For example, consider the expression

S: (* (* 2 5) (+ 3 8))

If we let E1 be all of S, then C1, the current continuation, is □: do the whole computation and return it.

If E2 is (* 2 5) then C2, the continuation of E2, is (* □ (+ 3 8))

If E3 is (+ 3 8), the continuation of E3 is C3: (* 10 □).

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)

Direction	Expression	Continuation	Result of Expression
IN	(fact 3)	□	?
IN	(fact 2)	(* 3 □)	?
IN	(fact 1)	(* 3 (*2 □))	?
IN	(fact 0)	(* 3 (* 2 (* 1 □)))	?
OUT	(fact 0)	(* 3 (* 2 (* 1 1)))	1
OUT	(fact 1)	(* 3 (* 2 1))	1
OUT	(fact 2)	(* 3 2)	2
OUT	(fact 3)	6	6

```
(define fact-acc (lambda (n acc)
        (if (= 0 n) acc (fact-acc (- n 1) (* n acc)))))
```

Dynamics of (fact 3 1)

Direction	Expression	Continuation	Result of Expression
IN	(fact-acc 3 1)	□	?
IN	(fact-acc 2 3)	□	?
IN	(fact-acc 1 6)	□	?
IN	(fact-acc 0 6)	□	?
OUT	(fact-acc 0 6)	6	6
OUT	(fact-acc 1 6)	6	6
OUT	(fact-acc 2 3)	6	6
OUT	(fact-acc 3 1)	6	6

Direction	Expression	Continuation	Result of Expression
IN	(fact-acc 3 1)	□	?
IN	(fact-acc 2 3)	□	?
IN	(fact-acc 1 6)	□	?
IN	(fact-acc 0 6)	□	?
OUT	(fact-acc 0 6)	6	6
OUT	(fact-acc 1 6)	6	6
OUT	(fact-acc 2 3)	6	6
OUT	(fact-acc 3 1)	6	6

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.