Control Flow in Scheme

Normally, Scheme’s control flow is simple and recursive:

• The first argument is evaluated to get a function.

• Remaining arguments are evaluated to get actual parameters.

• Actual parameters are bound to the function’s formal parameters.

• The functions’ body is evaluated to obtain the value of the function call.

This approach routinely leads to deeply nested expression evaluation.
As an example, consider a simple function that multiplies a list of integers:

```
(define (*list L)
  (if (null? L)
      1
      (* (car L)(*list (cdr L)))
  )
)
```

The call `(*list '(1 2 3 4 5))` expands to

```
(* 1 (* 2 (* 3 (* 4 (* 5 1))))))
```

But,

what if we get clever and decide to improve this function by noting that if 0 appears anywhere in list L, the product must be 0?
Let’s try

(define (*list0 L)
  (cond
    ((null? L) 1)
    ((= 0 (car L)) 0)
    (else (* (car L)
                (*list0 (cdr L))))
  )
)

This helps a bit—we never go past a zero in \( L \), but we still unnecessarily do a sequence of pending multiplies, all of which must yield zero!

Can we escape from a sequence of nested calls once we know they’re unnecessary?
Exceptions

In languages like Java, a statement may throw an exception that’s caught by an enclosing exception handler. Code between the statement that throws the exception and the handler that catches it is abandoned.

Let’s solve the problem of avoiding multiplication of zero in Java, using its exception mechanism:

```java
class Node {
    int val;
    Node next;
}
class Zero extends Throwable {
}
```
int mult (Node L) {
    try {
        return multNode(L);
    } catch (Zero z) {
        return 0;
    }
}

int multNode(Node L)
    throws Zero {
    if (L == null)
        return 1;
    else if (L.val == 0)
        throw new Zero();
    else return
        L.val * multNode(L.next);
}

In this implementation, no multiplies by zero are ever done.
Continuations

In our Scheme implementation of *list, we’d like a way to delay doing any multiplies until we know no zeros appear in the list. One approach is to build a continuation—a function that represents the context in which a function’s return value will be used:

(define (*listC L con)
  (cond
    ((null? L) (con 1))
    ((= 0 (car L)) 0)
    (else
     (*listC (cdr L)
      (lambda (n)
       (* n (con (car L)))))
    ))
  )
)
The top-level call is
(*listC L (lambda (x) x))
For ordinary lists *listC expands to a series of multiplies, just like *list did.

(define (id x) x)
(*listC '(1 2 3) id) ⇒
(*listC '(2 3)
  (lambda (n) (* n (id 1)))) ≡
(*listC '(2 3)
  (lambda (n) (* n 1))) ⇒
(*listC '(3)
  (lambda (n) (* n (* 2 1)))) ≡
(*listC '(3)
  (lambda (n) (* n 2))) ⇒
(*listC ()
  (lambda (n) (* n (* 3 2)))) ≡
(*listC () (lambda (n) (* n 6)))
⇒ (* 1 6) ⇒ 6
But for a list with a zero in it, we get a different execution path:

\[ (*\text{listC} \ '(1\ 0\ 3)\ \text{id}) \Rightarrow \]
\[ (*\text{listC} \ '(0\ 3)\]
\[ \quad (\text{lambda} \ (n) \ (*\ n\ (\text{id}\ 1))) \Rightarrow 0 \]

No multiplies are done!
Another Example of Continuations

Let’s redo our list multiply example so that if a zero is seen in the list we return a function that computes the product of all the non-zero values and a parameter that is the “replacement value” for the unwanted zero value. The function gives the caller a chance to correct a probable error in the input data.

We create

(*list2 L) ≡

Product of all integers in L if no zero appears else
(lambda (n) (* n product-of-all-nonzeros-in-L))
(define (*list2 L) (*listE L id))

(define (*listE L con)
  (cond
    ((null? L) (con 1))
    ( (= 0 (car L))
      (lambda(n)
        (* (con n)
          (*listE (cdr L) id))))
    (else
     (*listE (cdr L)
      (lambda(m)
        (* m (con (car L)))))))
  )
)
In the following, we check to see if *list2 returns a number or a function. If a function is returned, we call it with 1, effectively removing 0 from the list

(let ( (V (*list2 L)) )
  (if (number? V)
      V
      (V 1)
  )
)
For ordinary lists $*\text{list}_2$ expands to a series of multiplies, just like $*\text{list}$ did.

$(*\text{listE} '(1 2 3) \text{id}) \Rightarrow$

$(*\text{listE} '(2 3)
  (\lambda (m) (* m \text{id} 1))) \equiv$

$(*\text{listE} '(2 3)
  (\lambda (m) (* m 1))) \Rightarrow$

$(*\text{listE} '(3)
  (\lambda (m) (* m (* 2 1)))) \equiv$

$(*\text{listE} '(3)
  (\lambda (m) (* m 2))) \Rightarrow$

$(*\text{listE} ()
  (\lambda (m) (* m (* 3 2)))) \equiv$

$(*\text{listE} () \text{(lambda} (n) (* n 6)))$

$\Rightarrow (* 1 6) \Rightarrow 6$
But for a list with a zero in it, we get a different execution path:

\[
(*\text{list}\text{E} \ (1 \ 0 \ 3) \ \text{id}) \Rightarrow
(*\text{list}\text{E} \ (0 \ 3)
(lambda \ (m) \ (* \ m \ (\text{id} \ 1)))) \Rightarrow
(lambda \ (n) \ (* \ (\text{con} \ n)
(*\text{list}\text{E} \ (3) \ \text{id}))) \equiv
(lambda \ (n) \ (* \ (* \ n \ 1)
(*\text{list}\text{E} \ (3) \ \text{id}))) \equiv
(lambda \ (n) \ (* \ (* \ n \ 1) \ 3))
\]

This function multiplies \( n \), the replacement value for 0, by 1 and 3, the non-zero values in the input list.
But note that only one zero value in the list is handled correctly!

Why?

(define (*listE L con)
  (cond
    ((null? L) (con 1))
    ((= 0 (car L))
      (lambda (n)
        (* (con n)
          (*listE (cdr L) id))))
    (else
      (*listE (cdr L)
        (lambda (m)
          (* m (con (car L))))))))
Continuations in Scheme

Scheme provides a built-in mechanism for creating continuations. It has a long name: call-with-current-continuation. This name is usually abbreviated as call/cc.

call/cc takes a single function as its argument. That function also takes a single argument. That is, we use call/cc as

\[(\text{call/cc } \text{funct}) \quad \text{where} \quad \text{funct} \equiv (\text{lambda} \ (\text{con}) \ (\text{body}))\]

call/cc calls the function that it is given with the “current continuation” as the function’s argument.
Current Continuations

What is the current continuation? It is itself a function of one argument. The current continuation function represents the execution context within which the call/cc appears. The argument to the continuation is a value to be substituted as the return value of call/cc in that execution context.

For example, given

(+ (fct n) 3)

the current continuation for (fct n) is (lambda (x) (+ x 3))

Given (* 2 (+ (fct z) 10))

the current continuation for (fct z) is (lambda (m) (* 2 (+ m 10)))
To use \texttt{call/cc} to grab a continuation in (say) \((+ \ (\texttt{fct} \ n) \ 3)\) we make \((\texttt{fct} \ n)\) the body of a function of one argument. Let’s call that argument \texttt{return}. We therefore create
\[
(\lambda (\texttt{return}) \ (\texttt{fct} \ n))
\]
Then \[
(\texttt{call/cc} \\
\quad (\lambda (\texttt{return}) \ (\texttt{fct} \ n)))
\]
binds the current continuation to \texttt{return} and executes \((\texttt{fct} \ n)\).

We can ignore the current continuation bound to \texttt{return} and do a normal return

or

we can use \texttt{return} to force a return to the calling context of the \texttt{call/cc}.
The call \textit{(return value)} forces value to be returned as the value of call/cc in its context of call.

Example:

\begin{verbatim}
(* (call/cc (lambda (return) (/ (g return) 0))) 10)
\end{verbatim}

\begin{verbatim}
(define (g con) (con 5))
\end{verbatim}

Now during evaluation no divide by zero error occurs. Rather, when \textit{(g return)} is called, 5 is passed to con, which is bound to return. Therefore 5 is used as the value of the call to call/cc, and 50 is computed.
Continuations are Just Functions

Continuations may be saved in variables or data structures and called in the future to “reactive” a completed or suspended computation.

```
(define CC ()
(define (F)
  (let ((v (call/cc
              (lambda(here)
                (set! CC here)
                1))))
    (display "The ans is: ")
    (display v)
    (newline))
)
```

This displays The ans is: 1
At any time in the future, (CC 10) will display The ans is: 10