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:

```
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:

```
(*listC '(1 0 3) id) ⇒
(*listC '(0 3)
  (lambda (n) (* n (id 1)))) ⇒ 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 `*list2` expands to a series of multiplies, just like `*list` did.

```
(*listE '(1 2 3) id) ⇒
(*listE '(2 3)
  (lambda (m) (* m (id 1)))) ≡
(*listE '(2 3)
  (lambda (m) (* m 1))) ⇒
(*listE '(3)
  (lambda (m) (* m (* 2 1)))) ≡
(*listE '(3)
  (lambda (m) (* m 2))) ⇒
(*listE ()
  (lambda (m) (* m (* 3 2)))) ≡
(*listE () (lambda (n) (* n 6)))
   ⇒ (* 1 6) ⇒ 6
```

But for a list with a zero in it, we get a different execution path:

```
(*listE '(1 0 3) id) ⇒
(*listE '(0 3)
  (lambda (m) (* m (id 1)))) ⇒
(lambda (n) (* (con n)
   (* listE '(3) id))) ≡
(lambda (n) (* (* n 1)
   (* listE '(3) id))) ≡
(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))))))
  )
)
```