Apa (. F Ma Klasius. Logni n rpocolos varvoippin Latin TREEL LEMONT N TELEVISTI. TO GUOVE JIVETA XEWETIE!

> MINOÙ MAFON MY NÚGOL Scheme (+ a b) - Roadia 20 a ua 20 b. (+ (+ 23) 5)

Kille procedure Equipoletan povor 600 evaluated augunent.

(define (two-times x)(+ x x) )  $\Rightarrow (two-times 5) \Rightarrow \sim 10$ 

Trackour if, cond, and, or, let ... #1, #t

Nieres '(abc) = ls (lambda (x)) a = (car ls) (cdr ls) = '(bc)

Arasporn

(Cons 1 '(23)) = (123)

#### Exercise

Exercise 16.23

Explain why the test for termination within random-data is (negative? target).

## 16.8 Escaping from Flat Recursions

The call/cc operator gives the ability to escape from recursive computations while basically throwing out all the work that has stacked up. A simple example clarifies in what sense the mechanism avoids doing pending computations. We look at the problem of taking the product of a list of numbers and adding the number n to the product if the result is nonzero:

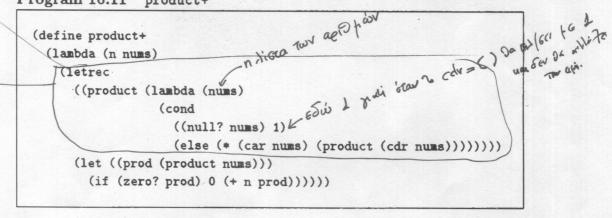
(product+ 5 '(3 6 2 7))  $\Rightarrow$  (+ 5 252)  $\Rightarrow$  257 (product+ 7 '(2 3 0 8))  $\Rightarrow$  0

We also distributed as 0.

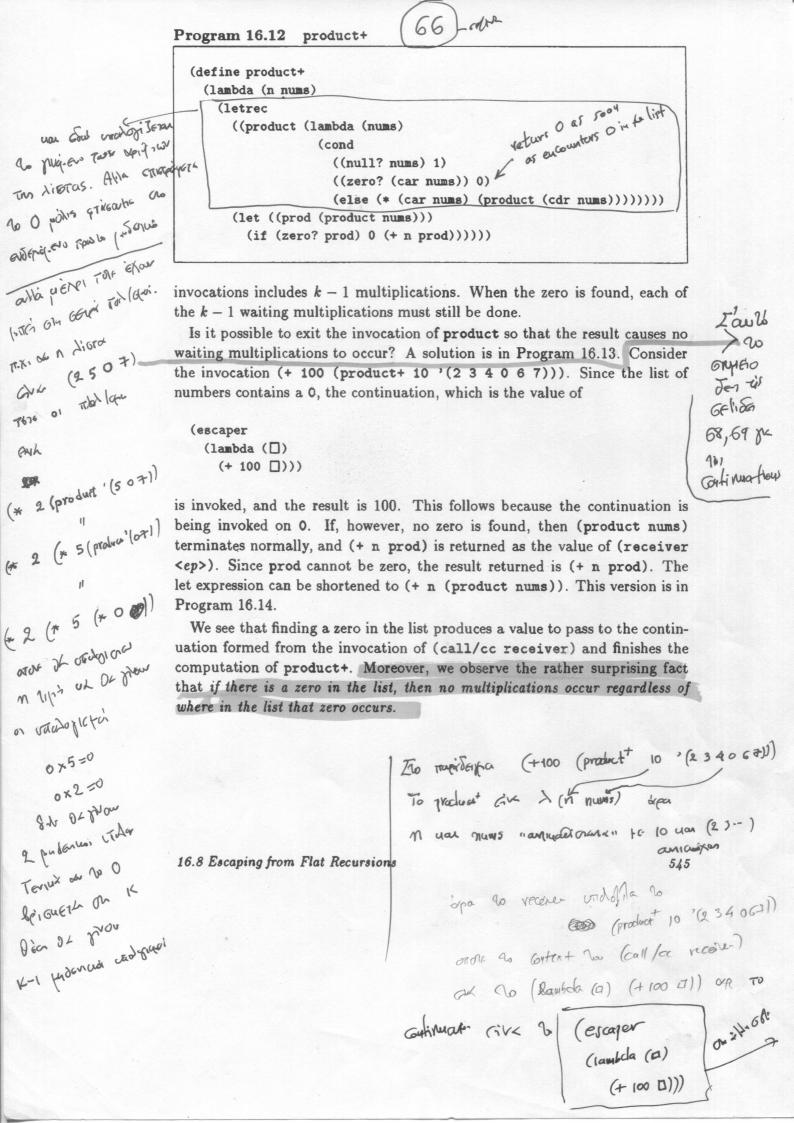
Here is the solution in a functional style:

va dogiser tise

Program 16.11 product+



This solution can be improved by adding a test to determine if one of the values in the list is zero. This stops the recursion upon encountering the first zero. This version is in Program 16.12. Consider the following subtle fact: Finding a zero in the list does not stop the computation of product. In fact, what happens is that if the first zero is in the kth position, then there are k-1 multiplications using zero. This is because the context of the product



```
To exit on zono (in to (escaper AD. E[]).

Confined on del to (escaper AD. E[]).
                          Program 16.13
                                            product+
 Aer høyn perokin o
                            (define product+
ad f to 0 Egglicles
                              (lambda (n nums)
                                (let ((receiver
                                         (lambda (exit-on-zero)
                                           (letrec
 Give to Gutimater
                                             ((product (lambda (nums)
                                                         (cond
                                                           ((null? nums) 1)
                                                           ((zero? (car nums)) (exit-on-zero 0))
 (exager AB. (+ 100 a))
                                                           (else (* (car nums)
                                                                    (product (cdr nums))))))))
 21. Morallow No
                                             (let ((prod (product nums)))
                                               (if (zero? prod) 0 (+ n prod)))))))
                                  (call/cc receiver)))
na empolemen
 relain ou Contrat
                          Program 16.14
                                           product+
                                                                         of And In recen
 ETTEN - UTCHEN TO ESTOPEN
                            (define product+
                              (lambda (n nums)
 ELINDERO P 100.
                                                                                 copietos en lisques
                                (let ((receiver
                                        (lambda (exit-on-zero)
DE WING works
                                          (letrec
"obd(ievovia" otalogictis.
                                            ((product (lambda (nums)
                                                         (cond
                                                           ((null? nums) 1)
                                                           ((zero? (car nums)) (exit-on-zero 0))
```

# 16.9 Escaping from Deep Recursions

Let us take a look at a slightly more complicated example. The problem is to redefine product+ for a larger class of lists. Specifically, we allow deep lists of numbers. Thus we can invoke

(else (\* (car nums)

(product (cdr nums))))))))

```
(product+ 5 '((1 2) (1 1 (3 1 1)) (((((1 1 0) 1) 4) 1) 1)))
```

(+ n (product nums))))))

(call/cc receiver))))

546

Introduction to Continuations

### 16.4 Continuations from Contexts and Escape Procedures

We are about to discuss call-with-current-continuation (or call/cc). If call/cc is not available on your Scheme, define it as follows:

### Program 16.1 call/cc

(define call/cc call-with-current-continuation)

The receiver is a procedure of one argument. Its argument is called a continuation. The continuation is also a procedure of one argument. Regardless of how we form the continuation, (call/cc receiver) is the same as (receiver continuation). What is left is to understand how continuation is formed. To form continuation, we first form the context, c, of (call/cc receiver) in some expression E. We then invoke (escaper c), which forms continuation. We have now completely characterized call/cc. All we have left to do is see how our understanding of how to form continuations leads us to determine correctly the evaluation of expressions using call/cc.

Consider the following expression:

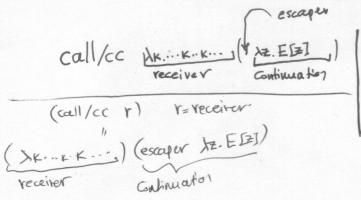
The context of (call/cc r) is the procedure, which is the value of

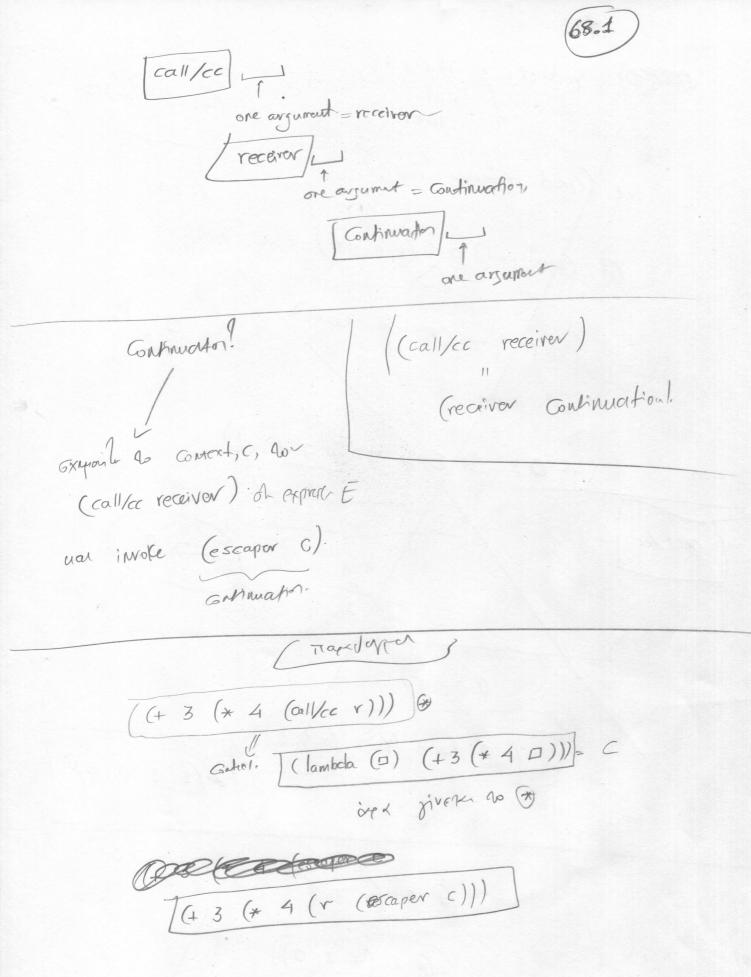
so our original expression means the same as:

That is, after the system forms the context of (call/cc r), the system passes it as an escape procedure to r. Since this is now just a simple invocation, all the rules for procedure invocation apply. A little practice is helpful. Let us consider r to be the value of (lambda (continuation) 6). What is the value of the expression derived from the call/cc expression above?

16.4 Continuations from Contexts and Escape Procedures

527





· ~ = (lambda (Continuation) 6). Tole (+3 (\* 4 (r (escaper c))) > (+3 (\* 46)) = 27 · r = (lambda (continuation) (Gutinuation 6)) (+3 (\* 4 (escaper (lambda (a) (+3 (\*4 12))))))))))))--) (+3 (+4 (+3 94)) Bora Aw (escaper (hambda (D) (+ 3 (\* 4 D)))) 6) > 27 · r= (lambda (Gutinuceria) (+ 2 (Construcción 6)1) TAF (+3 (+ 4 (+ 2 --To 16.13 reaffers to reaffer one (dêne I product + /ce receiver) lamba (n nums) ---ita vaarna a opigal lov graduat alt pe enderater no respective de Exiation la Gulinuation (Kapi) il 18/Auin 40 stack). (call/c receiver).



```
((lambda (continuation) 6)
(escaper (lambda (□) (+ 3 (* 4 □)))))
```

is 6; it does not use continuation, so the result is 27 (i.e., 3 + 4\*6). What about this one?

```
(+ 3 (* 4 ((lambda (continuation) (continuation 6))
(escaper (lambda (□) (+ 3 (* 4 □)))))))
```

The explicit invocation of continuation on 6 leads to

```
((escaper (lambda (□) (+ 3 (* 4 □)))) 6)
```

and then the result is 27. Is this one any different?

```
(+ 3 (* 4 ((lambda (continuation) (+ 2 (continuation 6)))
(escaper (lambda (□) (+ 3 (* 4 □))))))
```

The explicit invocation of continuation on 6 leads to

```
((escaper (lambda (□) (+ 3 (* 4 □)))) 6)
```

and then the result is 27. Remember, an escape invocation abandons its context, so (lambda ( $\square$ ) (+ 3 (\* 4 (+ 2  $\square$ )))) is abandoned. continuation has the value (escaper (lambda ( $\square$ ) ...  $\square$  ...)). Because the context of a call/cc invocation is turned into an escape procedure, we use the notation  $\langle ep \rangle$  for procedures that get passed to r.

Scheme supports procedures as values, and since  $\langle ep \rangle$  is a procedure, it is possible to invoke the same continuation more than once. In the next section there are three experiments with call/cc, and in the last experiment the same continuation is invoked twice. The countdown example of Chapter 17 shows what happens when the same continuation is invoked many times.