Άρα μ.ε των κλασική δομή η πρόσθεση καινούργιων στοιχείων πρέπει κάπου να τελειώσει. Το σύστημα γίνεται ασυνεπές

Μικρή αναφορά στη γλώσσα Scheme

(+ a b) – προσθέτει το a και το b.

(+ (* 2 3) 5)

Κάθε procedure εφαρμόζεται μόνον στα $\underbrace{\text{evaluated argument}}_{\text{πρώτα γίνεται αυτό}}$.

(define (two-times x)
    (+ x x)   ) ⇒ (two-times 5) ⇝ 10

Υπάρχουν if, cond, and, or, let ... #f, #t

Λίστες '(a b c) = ls

a = (car ls) (cdr ls) = '(b c)

(lambda (x)
    (+ x x))

(cons 1 '(2 3)) = (1 2 3)

Αναδρομή

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)) ⟹ (+ 5 252) ⟹ 257
(product+ 7 '(2 3 0 8)) ⟹ 0
```

να δημιουργεί λίγο το letrec.

αν ένα από αυτά είναι 0 τότε όλο το αποτέλεσμα είναι 0.

Here is the solution in a functional style:

Program 16.11 product+

```
(define product+
  (lambda (n nums)
    (letrec
      ((product (lambda (nums)
                  (cond
                    ((null? nums) 1)
                    (else (* (car nums) (product (cdr nums))))))))
      (let ((prod (product nums)))
        (if (zero? prod) 0 (+ n prod))))))
```

ορισμός του γινομένου των αριθμών της λίστας

η λίστα των αριθμών

εδώ 1 γιατί όταν το cdr = ( ) θα πολ/σει το 1 και δεν θα αλλάζει τον αριθ.

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**

66 σελ.

Program 16.12 product+

```
(define product+
  (lambda (n nums)
    (letrec
      ((product (lambda (nums)
                  (cond
                    ((null? nums) 1)
                    ((zero? (car nums)) 0)
                    (else (* (car nums) (product (cdr nums))))))))
      (let ((prod (product nums)))
        (if (zero? prod) 0 (+ n prod))))))
```

returns 0 as soon as encounters 0 in the list

και εδώ υπολογίζεται το γινόμενο των αριθμών της λίστας. Αλλά επιστρέφεται το 0 μόλις βρεθεί στο ενδεχόμενο πρώτο σημείο

αλλά μέχρι τότε έχουν γίνει όλοι οι πολ/σμοί. π.χ. αν η λίστα είναι (2 5 0 7) τότε οι πολ/σμοί είναι

(* 2 (product '(5 0 7))

=

(* 2 (* 5 (product '(0 7))

=

(* 2 (* 5 (* 0))

όταν θα υπολογίσει η λίστα θα γίνουν οι υπολογισμοί

0×5=0

0×2=0

δηλ. θα γίνουν 2 μηδενικοί πολ/σμοί. Γενικά αν το 0 βρίσκεται στη k θέση θα γίνουν k−1 μηδενικοί πολ/σμοί

invocations includes $k-1$ multiplications. When the zero is found, each of the $k-1$ waiting multiplications must still be done.

Is it possible to exit the invocation of **product** so that the result causes no waiting multiplications to occur? A solution is in Program 16.13. Consider the invocation `(+ 100 (product+ 10 '(2 3 4 0 6 7)))`. Since the list of numbers contains a `0`, the continuation, which is the value of

Σ' αυτό το σημείο δες τις σελίδες 68, 69 για continuations

```
(escaper
  (lambda (□)
    (+ 100 □)))
```

is invoked, and the result is 100. This follows because the continuation is being invoked on `0`. If, however, no zero is found, then `(product nums)` terminates normally, and `(+ n prod)` is returned as the value of `(receiver <ep>)`. Since **prod** cannot be zero, the result returned is `(+ n prod)`. The let expression can be shortened to `(+ n (product nums))`. This version is in Program 16.14.

We see that finding a zero in the list produces a value to pass to the continuation formed from the invocation of `(call/cc receiver)` and finishes the computation of **product+**. Moreover, we observe the rather surprising fact that *if there is a zero in the list, then no multiplications occur regardless of where in the list that zero occurs.*

Στο παράδειγμα (+100 (product+ 10 '(2 3 4 0 6 7)))

Το product+ είναι λ(n nums) άρα

n και nums «αντιστοιχούν» με 10 και (2 3 ...) αντίστοιχα

άρα το receiver υπολογίζει το (product+ 10 '(2 3 4 0 6 7))

οπότε το context του (call/cc receiver)

είναι το (lambda (□) (+ 100 □)) και το

continuation είναι το (escaper (lambda (□) (+ 100 □))) σελ. 68

67

απ' ότι φαίνεται

Άρα μόλις "βρεθεί" 0 αντί το 0 εφαρμόζει στο exit-on-zero που είναι το continuation άρα στο (escaper λ□.(+ 100 □)) δηλ. υπολογίζεις το (+ 100 0) = 0 και επιστρέφεται τελικά στο context επειδή υπάρχει το escaper δηλ. επιστρέφει το 100. Δεν υπάρχει κανένας "αναμένοντας" υπολογισμός.

Program 16.13 product+

το exit-on-zero είναι το continuation δηλ. το (escaper λ□. E[□]).

```
(define product+
  (lambda (n nums)
    (let ((receiver
            (lambda (exit-on-zero)
              (letrec
                ((product (lambda (nums)
                            (cond
                              ((null? nums) 1)
                              ((zero? (car nums)) (exit-on-zero 0))
                              (else (* (car nums)
                                       (product (cdr nums))))))))
                (let ((prod (product nums)))
                  (if (zero? prod) 0 (+ n prod)))))))
      (call/cc receiver))))
```

Program 16.14 product+

ορισμός του receiver

ορισμός του product

```
(define product+
  (lambda (n nums)
    (let ((receiver
            (lambda (exit-on-zero)
              (letrec
                ((product (lambda (nums)
                            (cond
                              ((null? nums) 1)
                              ((zero? (car nums)) (exit-on-zero 0))
                              (else (* (car nums)
                                       (product (cdr nums))))))))
                (+ n (product nums))))))
      (call/cc receiver))))
```

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

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

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)
```

call/cc is a procedure of one argument; we call the argument a *receiver*. 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:

```
(+ 3 (* 4 (call/cc r)))
```
(r είναι ο receiver)

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

```
(lambda (□) (+ 3 (* 4 □)))
```
είναι το context του (call/cc r)

so our original expression means the same as:

```
(+ 3 (* 4 (r (escaper (lambda (□) (+ 3 (* 4 □)))))))
```
(είναι η expression)

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?

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


escaper

call/cc $\underbrace{\lambda k.\dots k\dots k\dots}_{\text{receiver}}$ ($\underbrace{\lambda z.E[z]}_{\text{continuation}}$)

(call/cc r) r = receiver

||

$\underbrace{(\lambda k\dots k.k\dots)}_{\text{receiver}}$ (escaper $\underbrace{\lambda z.E[z]}_{\text{continuation}}$)

$\boxed{\text{call/cc}}$ ⎵
one argument = receiver

$\boxed{\text{receiver}}$ ⎵
one argument = continuation

$\boxed{\text{Continuation}}$ ⎵
one argument

Continuation!

σχηματίζει το context, C, του
(call/cc receiver) στη expression E
και invoke $\underbrace{\text{(escaper C)}}_{\text{continuation}}$

(call/cc receiver)
=
(receiver continuation)

Παράδειγμα

$\boxed{\text{(+ 3 (* 4 (call/cc r)))}}$ (*)

⇓ Contin. $\boxed{\text{(lambda (□) (+3 (* 4 □)))}}$ = C

όταν γίνεται το (*)

~~(+ 3 (* 4 (escaper c~~

$\boxed{\text{(+ 3 (* 4 (r (escaper c)))}}$

• r = (lambda (continuation) 6). Τότε

$$(+\ 3\ (*\ 4\ (r\ \underbrace{(\text{escaper}\ c)}_{\text{continuation}}))) \Rightarrow (+\ 3\ (*\ 4\ 6)) = 27$$

• r = (lambda (continuation) (continuation 6))

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

⇓

(+ 3 (* 4 (+3 24)) → 27

βγαίνει έξω

((escaper (lambda (□) (+ 3 (* 4 □)))) 6) ⇒ 27

• r = (lambda (continuation) (+ 2 (continuation 6)))

Τότε (+ 3 (* 4 (+ 2 --

Στο 16.13 προσθέτεις το πρόσθετο στις

(define $\boxed{\text{product+}}$ → lamba (n nums) ---------

όπου υπάρχει ο ορισμός του product+ αλλά με ενδεχόμενη επιτάχυνση του continuation (χωρίς να γεμίσει το stack).

όχι το πρόσθεμα στο

$\boxed{\text{(call/c receiver).}}$

69

The value of

```
((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 (□) (+ 3 (* 4 (+ 2 □)))) is abandoned. continuation has the value (escaper (lambda (□) ... □ ...)). Because the context of a call/cc invocation is turned into an escape procedure, we use the notation <*ep*> for procedures that get passed to r.

Scheme supports procedures as values, and since <*ep*> 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.