

# Εγγραφές Δραστηριοποίησης



Jackson Pollock, *The Key*, 1946 (action painting)

Κωστής Σαγώνας <[kostis@cs.ntua.gr](mailto:kostis@cs.ntua.gr)>  
Νίκος Παπασπύρου <[nickie@softlab.ntua.gr](mailto:nickie@softlab.ntua.gr)>

# Ερώτηση για... δέσιμο

---

- Κατά την εκτέλεση του προγράμματος, οι μεταβλητές δένονται (δυναμικά) με τιμές
- Οι τιμές αυτές πρέπει να αποθηκευτούν κάπου
- Κατά συνέπεια, οι μεταβλητές πρέπει κάπως να δεθούν με θέσεις μνήμης
- Πώς γίνεται αυτό;

# Συναρτησιακές γλώσσες συναντούν προστακτικές

---

- Οι προστακτικές γλώσσες κάνουν εμφανή την έννοια των θέσεων μνήμης: **`a := 42`**
  - Αποθήκευσε τον αριθμό **42** στη θέση μνήμης της μεταβλητής **a**
- Οι συναρτησιακές γλώσσες τις κρύβουν: **`val a = 42`**
  - Δέσε το όνομα **a** με την τιμή **42**
- Και τα δύο είδη γλωσσών πρέπει να “ενώσουν” μεταβλητές με τιμές που αναπαρίστανται στη μνήμη
- Άρα και οι δύο πρέπει να αντιμετωπίσουν την ίδια ερώτηση δεσίματος

# Περιεχόμενα

---

- Εγγραφές δραστηριοποίησης (**activation records**)
- Στατική δέσμευση εγγραφών δραστηριοποίησης
- Στοίβες από εγγραφές δραστηριοποίησης
- Χειρισμός φωλιασμένων ορισμών συναρτήσεων
- Συναρτήσεις ως παράμετροι
- Μακρόβιες εγγραφές δραστηριοποίησης

# Δραστηριοποιήσεις συναρτήσεων

---

- Ο χρόνος ζωής της εκτέλεσης μιας συνάρτησης, από τη στιγμή της κλήσης μέχρι την αντίστοιχη επιστροφή, ονομάζεται **δραστηριοποίηση (activation)** της συνάρτησης
- Όταν κάθε δραστηριοποίηση έχει το δικό της δέσιμο μεταβλητών σε θέσεις μνήμης, λέμε ότι έχουμε **μεταβλητές ειδικές για κάθε δραστηριοποίηση**
- Οι μεταβλητές αυτές ονομάζονται επίσης **δυναμικές (dynamic)** ή **αυτόματες (automatic)** μεταβλητές

# Μεταβλητές ειδικές για κάθε δραστηριοποίηση

---

Στις περισσότερες μοντέρνες γλώσσες προγραμματισμού, οι πιο συνήθεις αυτόματες μεταβλητές είναι αυτές που είναι βοηθητικές για κάθε δραστηριοποίηση συνάρτησης:

```
fun days2ms days =  
  let  
    val hours = days * 24.0  
    val minutes = hours * 60.0  
    val seconds = minutes * 60.0  
  in  
    seconds * 1000.0  
  end;
```

# Δραστηριοποίηση των μπλοκ

---

- Για μπλοκ που περιέχουν κώδικα, μιλάμε για
  - Τη **δραστηριοποίηση** του μπλοκ
  - Το **χρόνο ζωής** μιας εκτέλεσης του μπλοκ
- Μια μεταβλητή μπορεί να είναι ειδική για κάποια μόνο δραστηριοποίηση ενός συγκεκριμένου μπλοκ μιας συνάρτησης:

```
fun fact n =  
  if n = 0 then 1  
  else let val fac = fact (n-1) in n*fac  
end;
```

# Χρόνοι ζωής για τις μεταβλητές

---

- Οι περισσότερες προστακτικές γλώσσες έχουν κάποιο τρόπο να δηλώσουν ότι κάποιες μεταβλητές δένονται με μια συγκεκριμένη θέση μνήμης για όλη τη διάρκεια της εκτέλεσης του προγράμματος
- Προφανής λύση δέσμευσης: **στατική δέσμευση**  
(ο φορτωτής δεσμεύει χώρο για αυτές τις μεταβλητές)

```
int count = 0;
int nextcount() {
    count = count + 1;
    return count;
}
```



# Η εμφάνιση και ο χρόνος ζωής διαφέρουν

---

- Στις περισσότερες μοντέρνες γλώσσες, οι μεταβλητές με τοπική **εμφάνιση** έχουν **χρόνο ζωής** που συνήθως ταυτίζεται με τη δραστηριοποίηση του μπλοκ
- Όμως, οι δύο έννοιες είναι διαφορετικές μεταξύ τους, όπως π.χ. μπορεί να γίνει στη C μέσω του προσδιοριστή **static**:

```
int nextcount() {  
    static int count = 0;  
    count = count + 1;  
    return count;  
}
```

# Άλλοι χρόνοι ζωής για τις μεταβλητές

---

- Οι γλώσσες αντικειμενοστρεφούς προγραμματισμού χρησιμοποιούν μεταβλητές των οποίων ο χρόνος ζωής είναι συσχετισμένος με το χρόνο ζωής των αντίστοιχων αντικειμένων
- Κάποιες γλώσσες έχουν μεταβλητές των οποίων οι τιμές είναι **επίμονες (persistent)**: με άλλα λόγια οι μεταβλητές διατηρούν τις τιμές τους για πολλαπλές εκτελέσεις του ίδιου προγράμματος

# Εγγραφές δραστηριοποίησης

---

- Οι υλοποιήσεις των γλωσσών συνήθως συσκευάζουν όλες τις μεταβλητές που αναφέρονται σε μια συγκεκριμένη δραστηριοποίηση της συνάρτησης σε μια **εγγραφή δραστηριοποίησης (activation record)**
- Οι εγγραφές δραστηριοποίησης περιέχουν επίσης και όλα τα άλλα δεδομένα που σχετίζονται με δραστηριοποιήσεις, όπως:
  - Τη **διεύθυνση επιστροφής (return address)** της δραστηριοποίησης: δείχνει σε ποιο σημείο του προγράμματος πρέπει να πάει ο έλεγχος όταν επιστρέψει η συγκεκριμένη δραστηριοποίηση
  - Ένα **σύνδεσμο πρόσβασης (access link)** που δείχνει την εγγραφή δραστηριοποίησης της καλούσας συνάρτησης (calling function)

# Εγγραφές δραστηριοποίησης για μπλοκ

---

- Όταν εκτελείται ένα μπλοκ κώδικα, χρειαζόμαστε χώρο για τις τοπικές μεταβλητές του συγκεκριμένου μπλοκ
- Υπάρχουν διάφοροι τρόποι δέσμευσης αυτού του χώρου:
  - Δεσμεύουμε από πριν χώρο στην εγγραφή δραστηριοποίησης της περικλείουσας συνάρτησης
  - Επεκτείνουμε την εγγραφή δραστηριοποίησης της συνάρτησης όταν η εκτέλεση εισέλθει στο μπλοκ (και τη μειώνουμε ξανά όταν εξέλθουμε του μπλοκ)
  - Δεσμεύουμε κάποια ξεχωριστή εγγραφή δραστηριοποίησης για το μπλοκ
- Θα δούμε τον πρώτο από αυτούς τους τρόπους

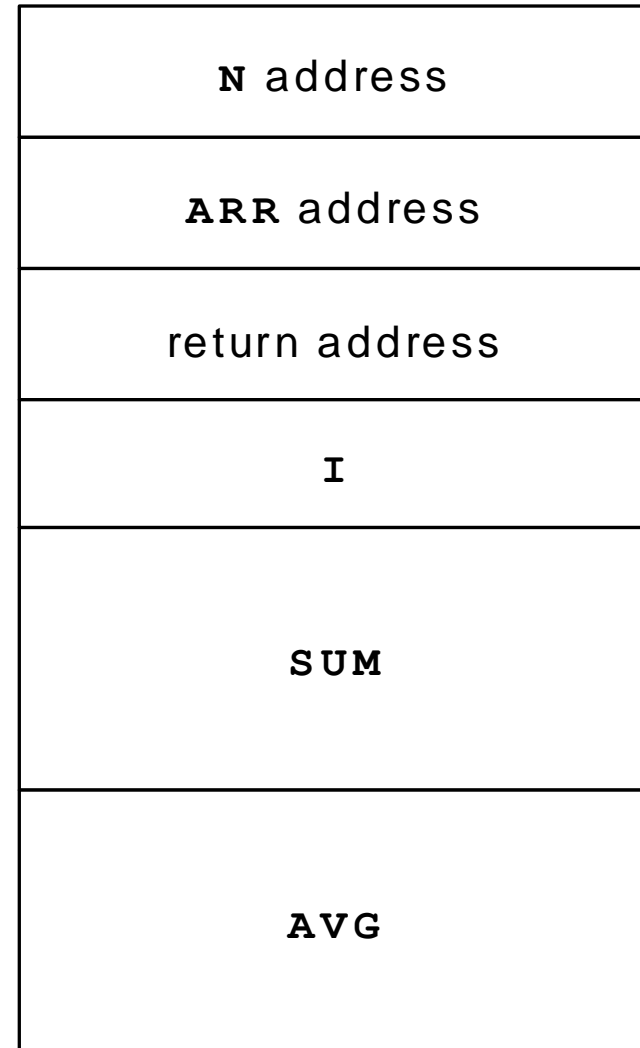
# Στατική δέσμευση (static allocation)

---

- Η απλούστερη προσέγγιση: δέσμευση **μίας** εγγραφής δραστηριοποίησης για κάθε μια συνάρτηση, στατικά
- Οι παλιές διάλεκτοι της Fortran και της Cobol χρησιμοποιούσαν (μόνο) αυτό το σύστημα δέσμευσης
- Αποτελεί απλό και πολύ γρήγορο τρόπο υλοποίησης

# Παράδειγμα

```
FUNCTION AVG (ARR, N)
DIMENSION ARR(N)
SUM = 0.0
DO 100 I = 1, N
    SUM = SUM + ARR(I)
100 CONTINUE
AVG = SUM / FLOAT(N)
RETURN
END
```



# Μειονέκτημα στατικής δέσμευσης

---

- Κάθε συνάρτηση έχει μία εγγραφή δραστηριοποίησης
- Μπορεί να υπάρξει **μία μόνο** δραστηριοποίηση συνάρτησης ζωντανή κάθε χρονική στιγμή!
- Οι μοντέρνες γλώσσες (συμπεριλαμβανομένων των μοντέρνων διαλέκτων της Cobol και της Fortran) δεν υπακούν σε αυτόν τον περιορισμό λόγω:
  - Αναδρομής
  - Πολυνηματικότητας (multithreading)

# Στοιίβες από εγγραφές δραστηριοποίησης (1)

---

- Για την υποστήριξη αναδρομής, πρέπει να δεσμεύσουμε μια νέα εγγραφή δραστηριοποίησης για κάθε δραστηριοποίηση της συνάρτησης
- **Δυναμική δέσμευση**: η εγγραφή δραστηριοποίησης δεσμεύεται όταν η συνάρτηση καλείται
- Σε πολλές γλώσσες, όπως η C, η εγγραφή αυτή αποδεσμεύεται όταν η συνάρτηση επιστρέψει



## Στοιίβες από έγγραφές δραστηριοποίησης (2)

---

- Με άλλα λόγια, η εκτέλεση δημιουργεί μια στοίβα από έγγραφές δραστηριοποίησης, όπου **πλαίσια (frames)**
  - σπρώχνονται στη στοίβα κατά την κλήση των συναρτήσεων, και
  - απομακρύνονται από τη στοίβα κατά την επιστροφή των συναρτήσεων



# Τρέχουσα εγγραφή δραστηριοποίησης

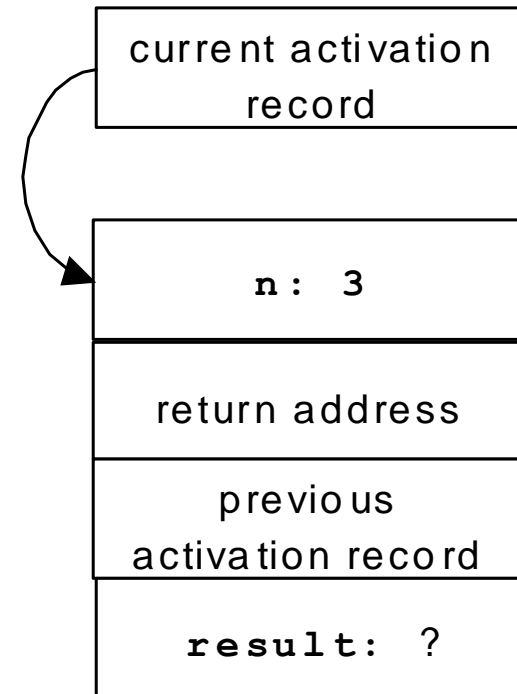
---

- Στη στατική δέσμευση η θέση κάθε εγγραφής δραστηριοποίησης καθορίζεται πριν την έναρξη εκτέλεσης του προγράμματος
- Στη δυναμική δέσμευση η θέση της **τρέχουσας εγγραφής δραστηριοποίησης (current activation record)** δεν είναι γνωστή παρά μόνο κατά το χρόνο εκτέλεσης
- Κάθε συνάρτηση πρέπει να ξέρει πώς θα βρει τη διεύθυνση της τρέχουσας εγγραφής δραστηριοποίησης
- Συχνά, ένας καταχωρητής της μηχανής δεσμεύεται για να περιέχει/κρατάει τη συγκεκριμένη τιμή

# Παράδειγμα σε C

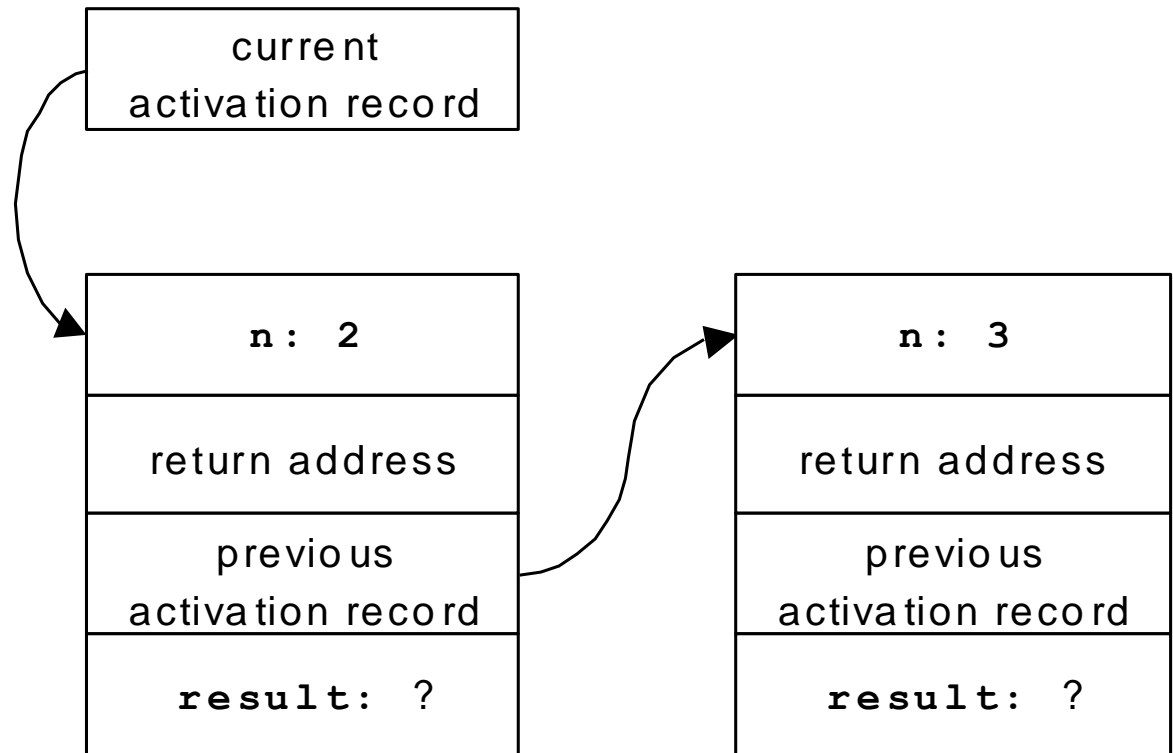
Αποτιμούμε την κλήση `fact(3)`. Η εικόνα δείχνει τα περιεχόμενα της στοίβας ακριβώς πριν την αναδρομική κλήση της `fact(2)` που θα δημιουργήσει τη δεύτερη εγγραφή δραστηριοποίησης.

```
int fact(int n) {  
    int result;  
    if (n < 2) result = 1;  
    else result = n * fact(n-1);  
    return result;  
}
```



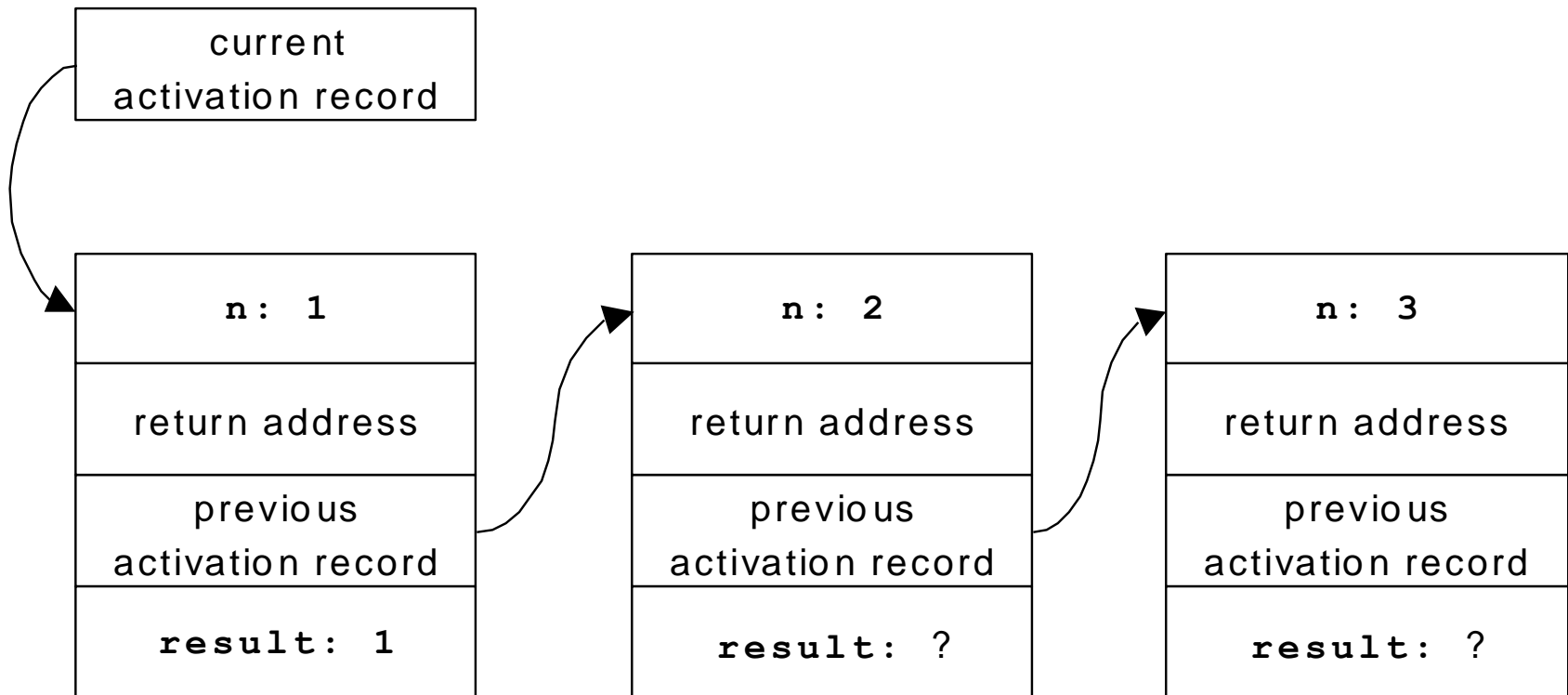
Τα περιεχόμενα της μνήμης ακριβώς πριν την τρίτη δραστηριοποίηση.

```
int fact(int n) {  
    int result;  
    if (n < 2) result = 1;  
    else result = n * fact(n-1);  
    return result;  
}
```



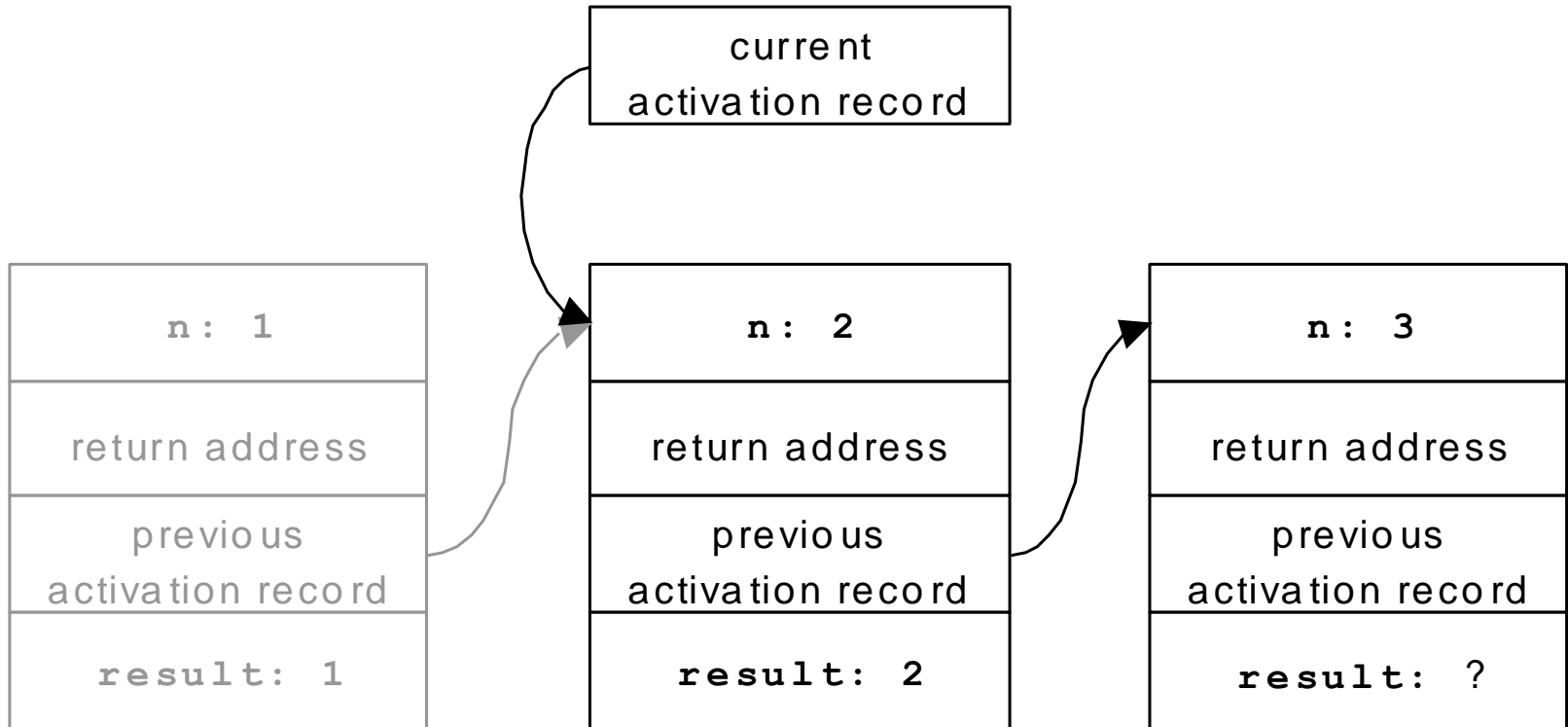
Τα περιεχόμενα της μνήμης ακριβώς πριν επιστρέψει η τρίτη δραστηριοποίηση.

```
int fact(int n) {  
    int result;  
    if (n < 2) result = 1;  
    else result = n * fact(n-1);  
    return result;  
}
```



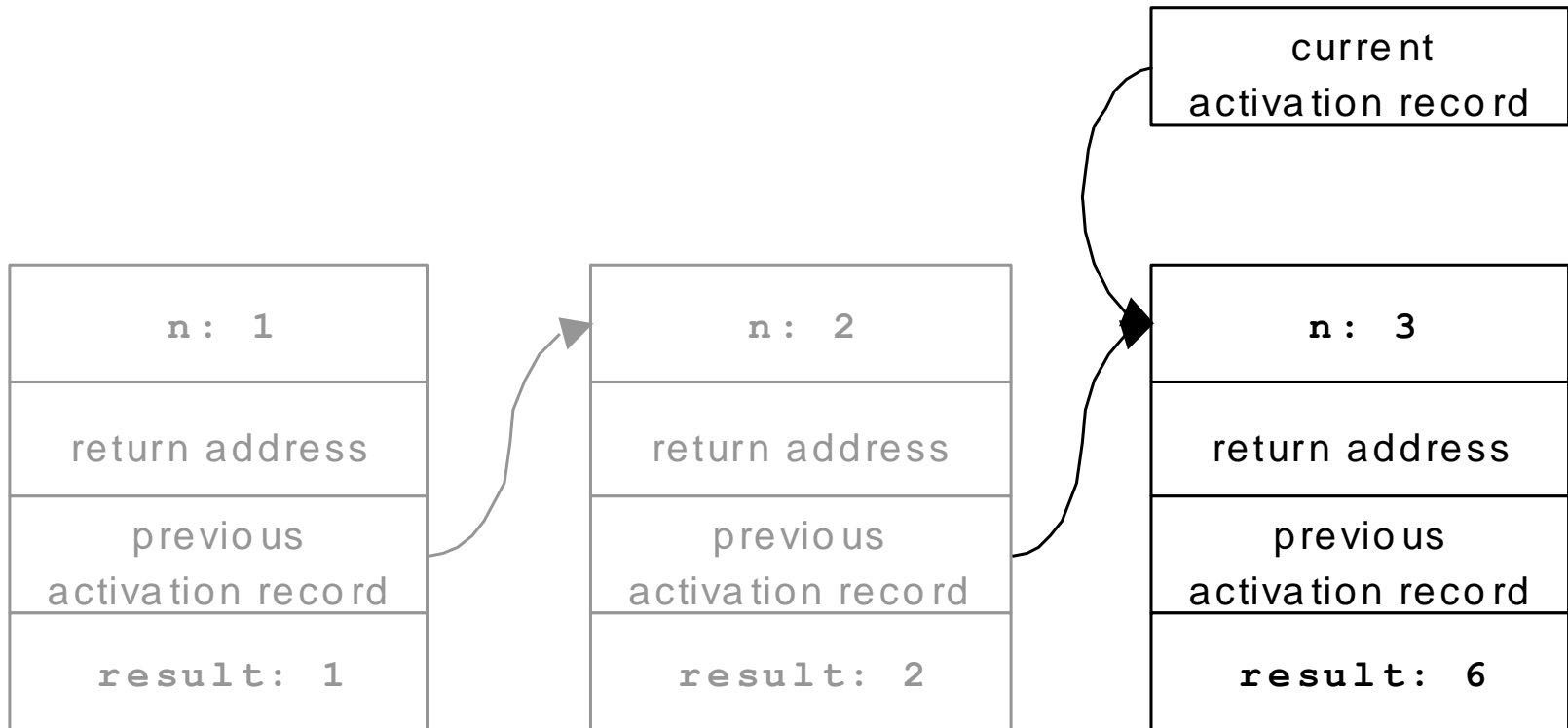
Η δεύτερη δραστηριοποίηση ακριβώς πριν επιστρέψει.

```
int fact(int n) {  
    int result;  
    if (n < 2) result = 1;  
    else result = n * fact(n-1);  
    return result;  
}
```



Η πρώτη δραστηριοποίηση ακριβώς πριν επιστρέψει με το αποτέλεσμα  $\text{fact}(3) = 6$ .

```
int fact(int n) {  
    int result;  
    if (n < 2) result = 1;  
    else result = n * fact(n-1);  
    return result;  
}
```



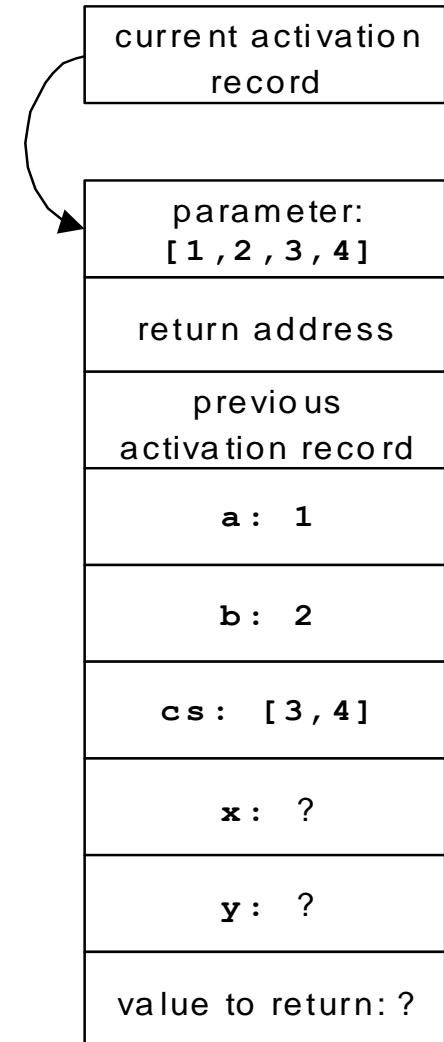
Οι εικόνες περιλαμβάνουν μια μικρή “απλοποίηση” όσον αφορά στο τι πράγματι αποθηκεύεται στη στοίβα

## Παράδειγμα σε ML

Αποτιμούμε την κλήση  
`halve [1,2,3,4]`

Η εικόνα δείχνει τα περιεχόμενα της μνήμης ακριβώς πριν την αναδρομική κλήση που δημιουργεί τη δεύτερη δραστηριοποίηση.

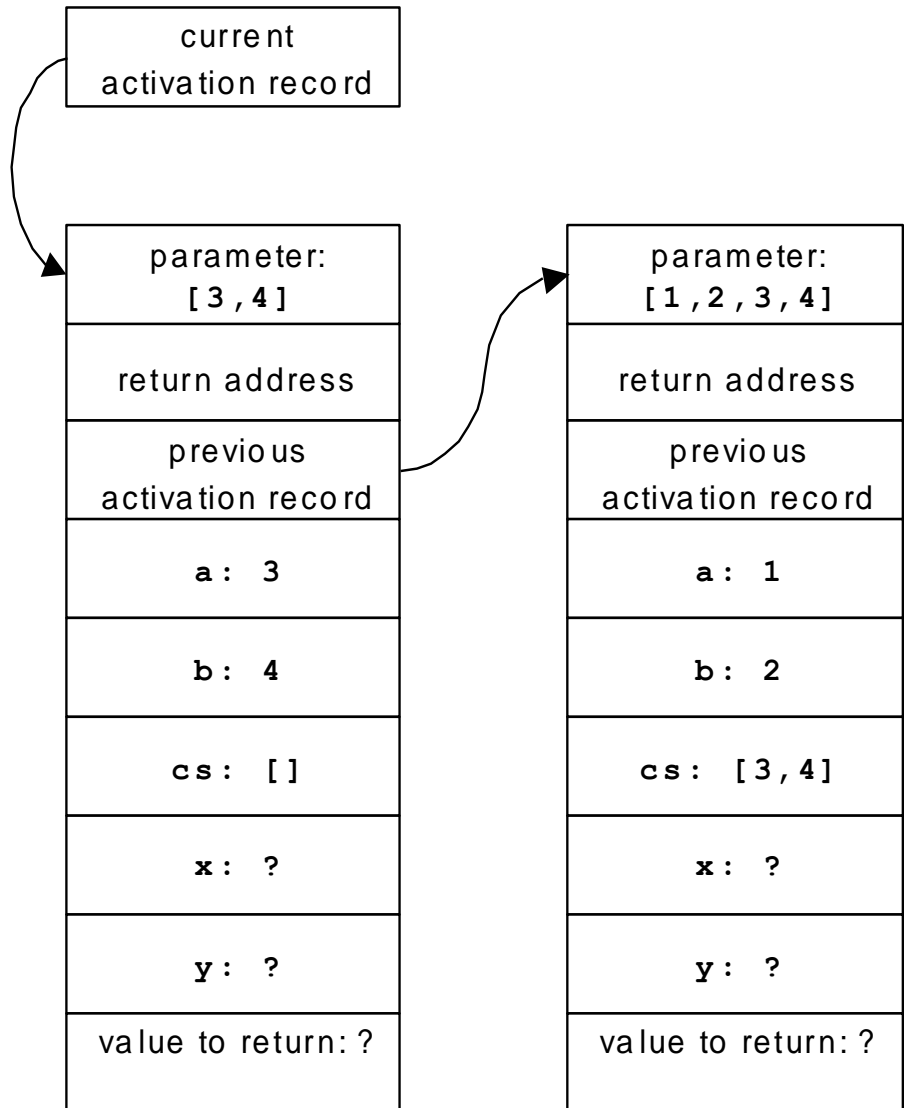
```
fun halve nil = (nil, nil)
  | halve [a] = ([a], nil)
  | halve (a::b::cs) =
    let
      val (x, y) = halve cs
    in
      (a::x, b::y)
    end;
```



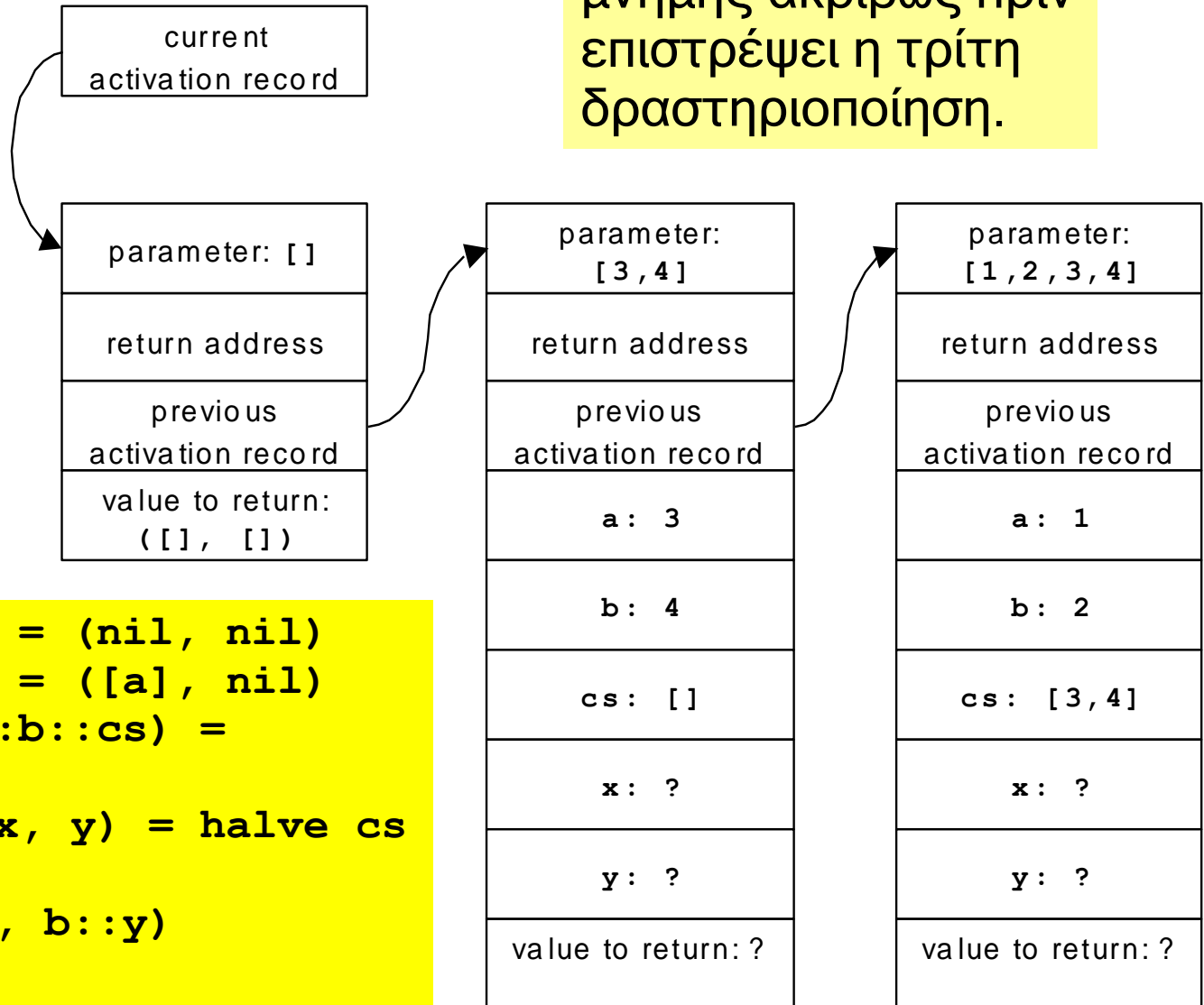


Τα περιεχόμενα της μνήμης ακριβώς πριν την τρίτη δραστηριοποίηση.

```
fun halve nil = (nil, nil)
  | halve [a] = ([a], nil)
  | halve (a::b::cs) =
    let
      val (x, y) = halve cs
    in
      (a::x, b::y)
    end;
```

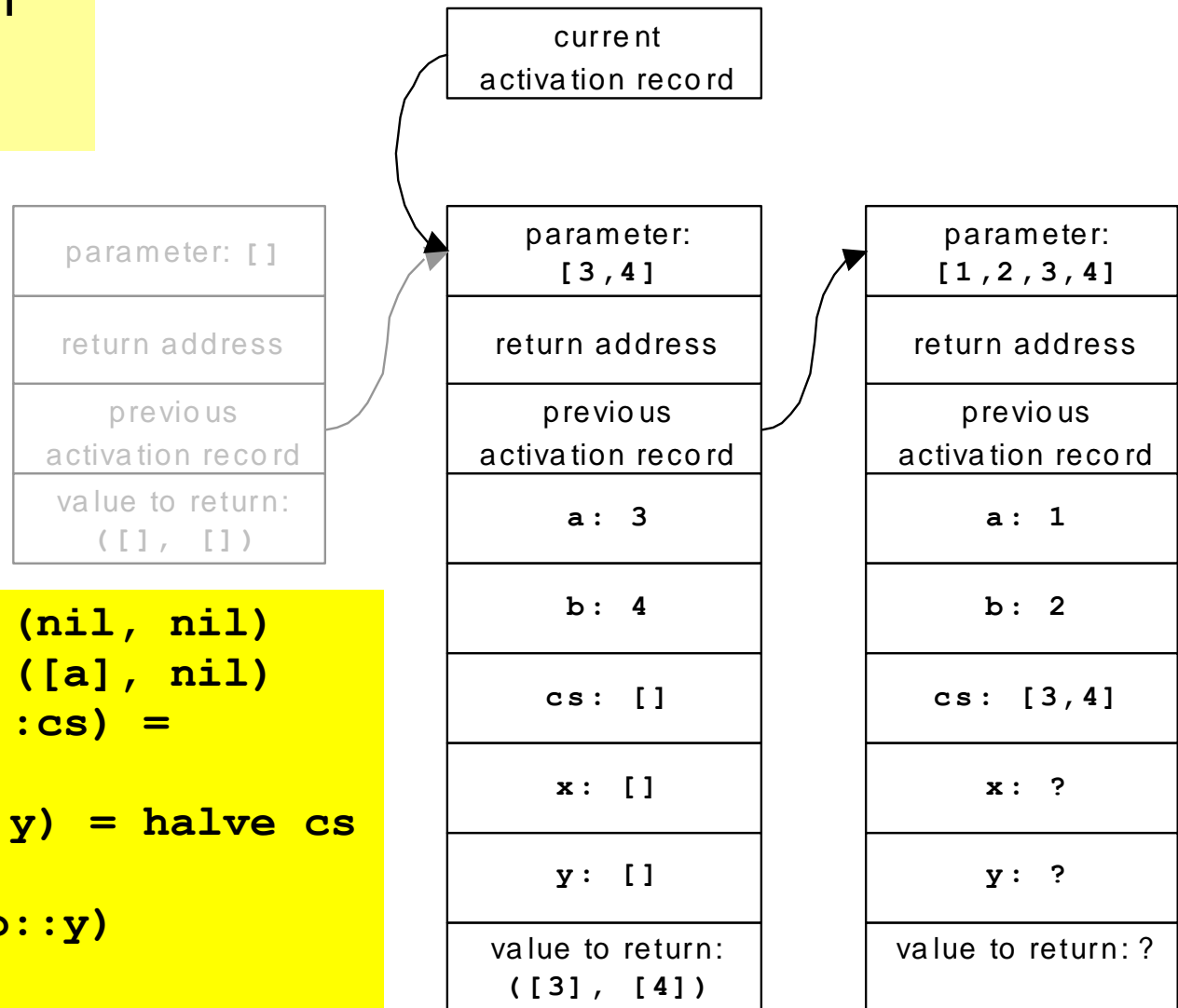


Τα περιεχόμενα της μνήμης ακριβώς πριν επιστρέψει η τρίτη δραστηριοποίηση.



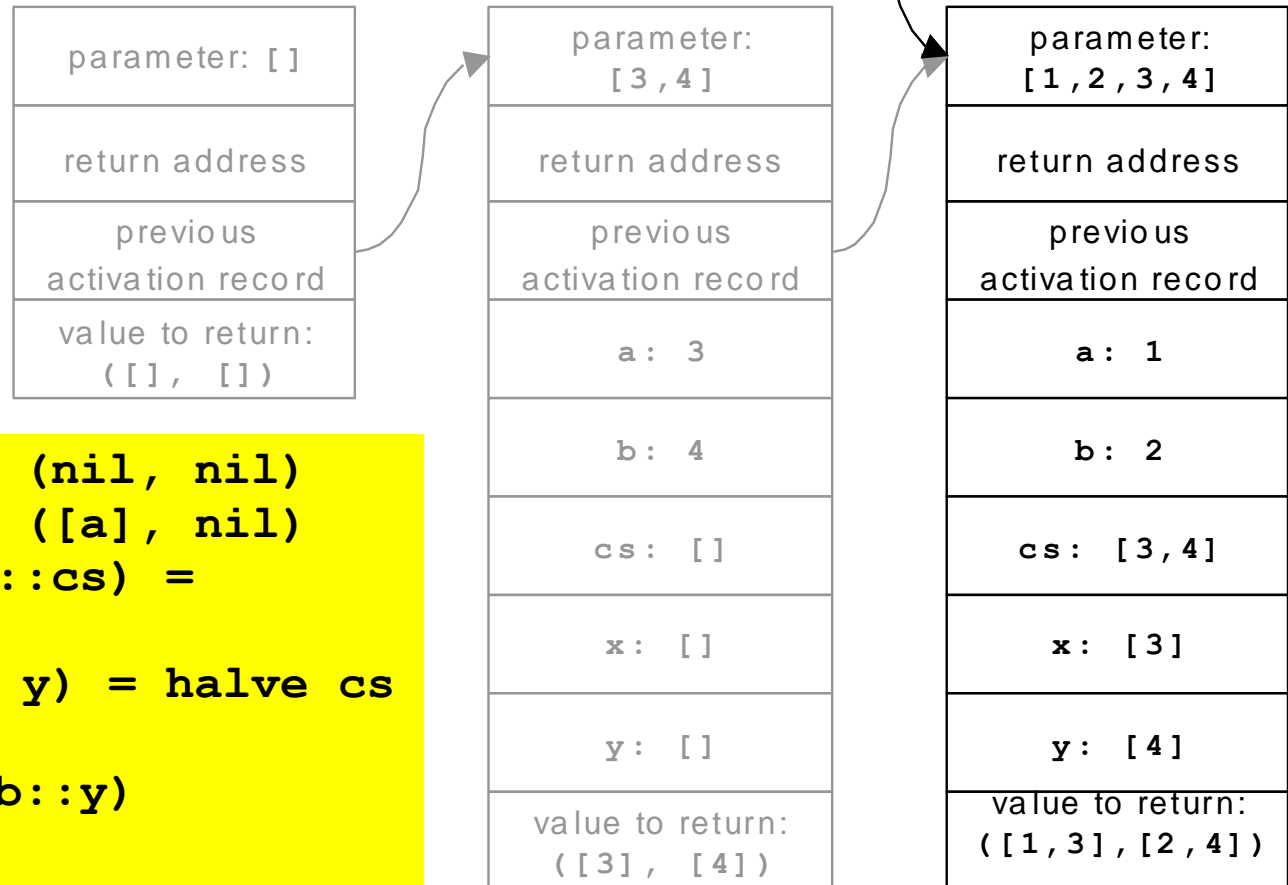
```
fun halve nil = (nil, nil)
| halve [a] = ([a], nil)
| halve (a::b::cs) =
  let
    val (x, y) = halve cs
  in
    (a::x, b::y)
  end;
```

Η δεύτερη δραστηριοποίηση ακριβώς πριν επιστρέψει.



```
fun halve nil = (nil, nil)
  | halve [a] = ([a], nil)
  | halve (a::b::cs) =
    let
      val (x, y) = halve cs
    in
      (a::x, b::y)
    end;
```

Η πρώτη δραστηριοποίηση ακριβώς πριν επιστρέψει με το αποτέλεσμα  
**halve [1,2,3,4] = ([1,3], [2,4])**



```

fun halve nil = (nil, nil)
| halve [a] = ([a], nil)
| halve (a::b::cs) =
  let
    val (x, y) = halve cs
  in
    (a::x, b::y)
  end;

```

# Χειρισμός φωλιασμένων συναρτήσεων

# Φωλιασμένες συναρτήσεις

---

- Ό,τι είδαμε μέχρι στιγμής επαρκεί για πολλές γλώσσες, συμπεριλαμβανομένης της C
- Αλλά όχι για γλώσσες που επιτρέπουν το ακόλουθο τρικ:
  - Οι ορισμοί των συναρτήσεων μπορεί να είναι φωλιασμένες μέσα σε άλλους ορισμούς συναρτήσεων
  - Οι εσωτερικές συναρτήσεις μπορεί να αναφέρονται σε τοπικές μεταβλητές των εξωτερικών συναρτήσεων (ακολουθώντας τους συνήθεις κανόνες εμβέλειας όταν υπάρχουν μπλοκ)
- Δηλαδή δεν επαρκεί για γλώσσες όπως η ML, η Ada, η Pascal, κ.α.

# Παράδειγμα

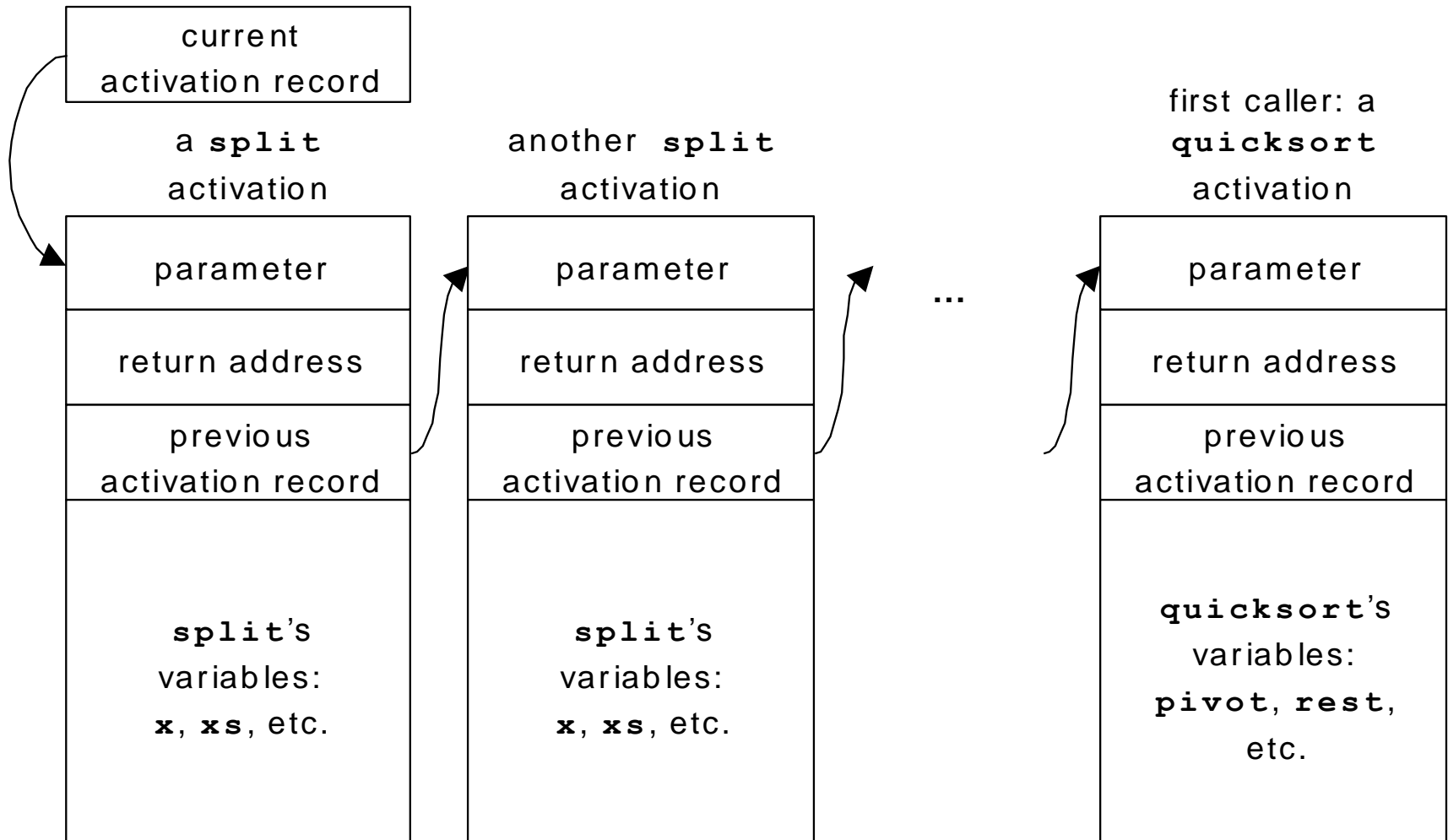
```
fun quicksort nil = nil
  | quicksort (pivot)::rest) =
    let
      fun split nil = (nil, nil)
        | split (x::xs) =
          let
            val (below, above) = split xs
          in
            if x < pivot then (x::below, above)
            else (below, x::above)
          end;
      val (below, above) = split rest
    in
      quicksort below @ [pivot] @ quicksort above
    end;
```

# Το πρόβλημα

---

- Πώς μπορεί η δραστηριοποίηση της εσωτερικής συνάρτησης (`split`) να βρει την εγγραφή δραστηριοποίησης της εξωτερικής συνάρτησης (`quicksort`);
- Παρατηρήστε ότι δεν είναι απαραίτητα η προηγούμενη εγγραφή δραστηριοποίησης, επειδή η συνάρτηση που κάλεσε την εσωτερική συνάρτηση:
  - μπορεί να είναι μια άλλη εσωτερική συνάρτηση, ή
  - μπορεί να είναι μια αναδρομική κλήση συνάρτησης (όπως συμβαίνει στο παράδειγμά μας με τη `split`)

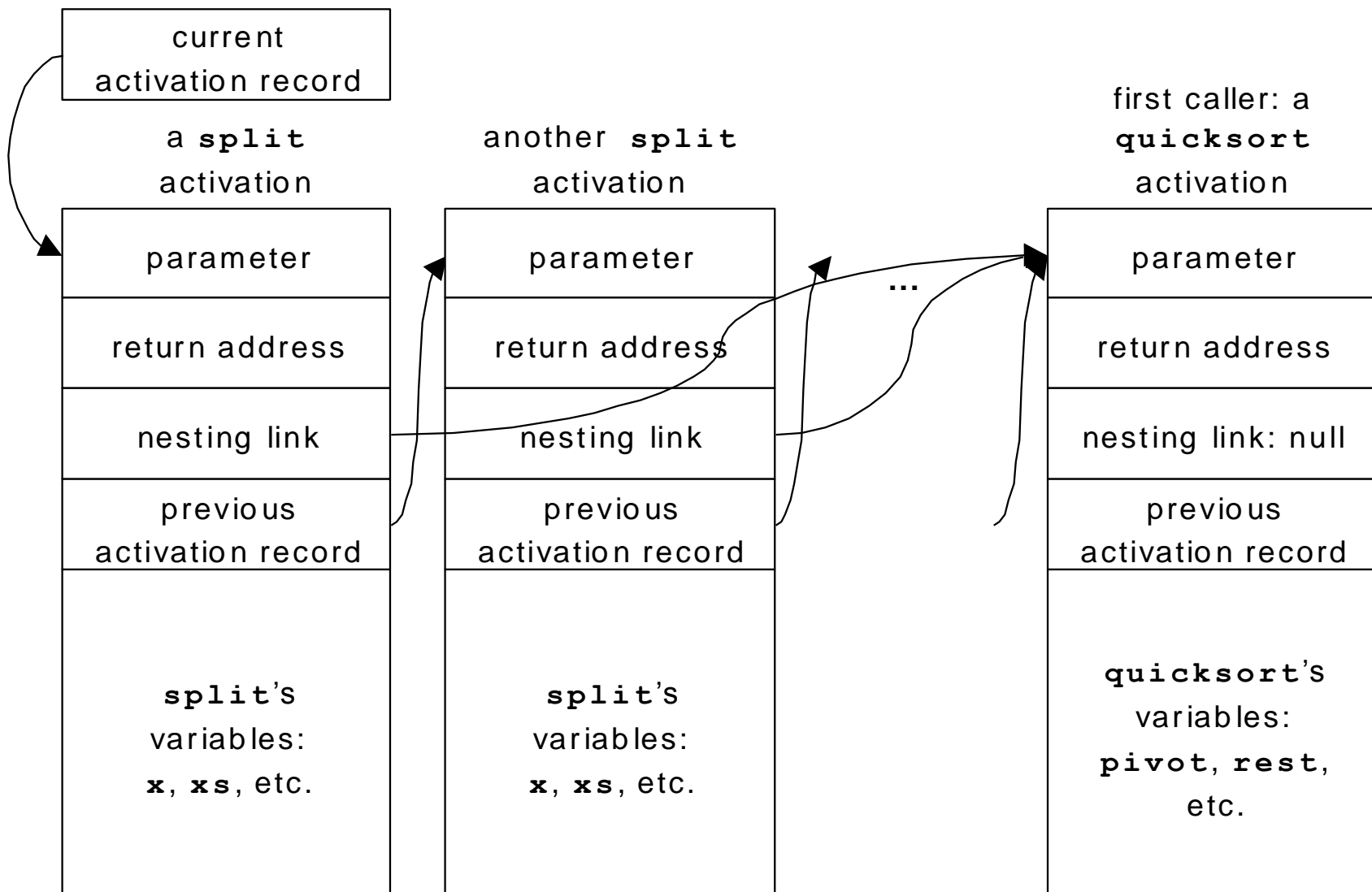




## Σύνδεσμος φωλιάσματος (nesting link)

---

- Μια εσωτερική συνάρτηση πρέπει να είναι σε θέση να βρει τη διεύθυνση της πιο πρόσφατης δραστηριοποίησης της **αμέσως εξωτερικής** της συνάρτησης
- Μπορούμε να κρατήσουμε αυτό το σύνδεσμο στην εγγραφή δραστηριοποίησης

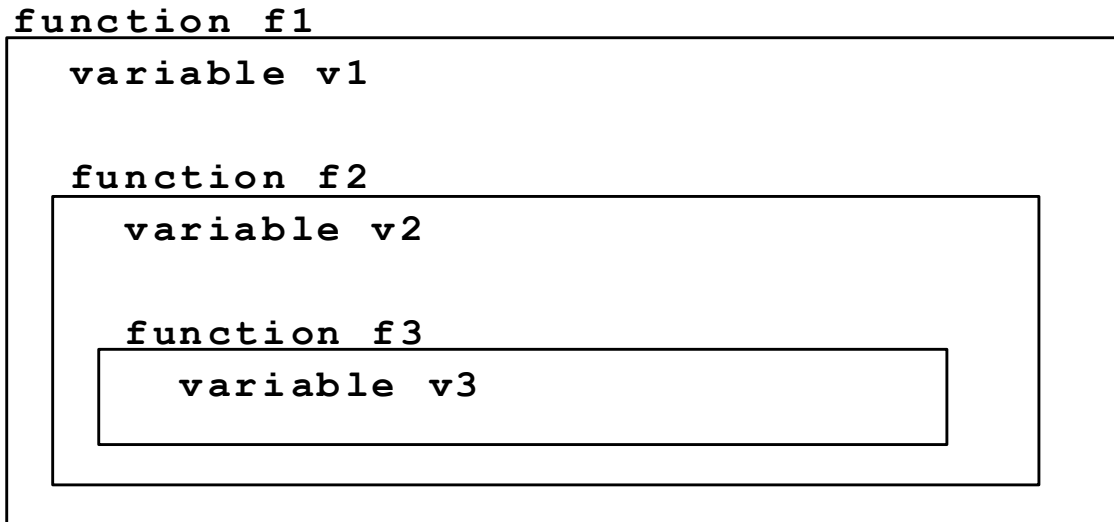


# Πώς παίρνουν τιμές οι σύνδεσμοι φωλιάσματος;

---

- Εύκολα αν υπάρχει μόνο ένα επίπεδο φωλιάσματος
  - Όταν καλείται μια εξωτερική συνάρτηση:
    - θέτουμε την τιμή null στο σύνδεσμο
  - Όταν καλείται μια εσωτερική συνάρτηση από μία εξωτερική:
    - ο σύνδεσμος φωλιάσματος παίρνει ως τιμή την εγγραφή δραστηριοποίησης της καλούσας συνάρτησης
  - Όταν καλείται μια εσωτερική συνάρτηση από μία εσωτερική:
    - ο σύνδεσμος φωλιάσματος παίρνει ως τιμή το σύνδεσμο φωλιάσματος της καλούσας συνάρτησης
- Πιο πολύπλοκα από τον παραπάνω τρόπο εάν υπάρχουν πολλαπλά επίπεδα φωλιάσματος...

# Πολλαπλά επίπεδα φωλιάσματος



- Οι αναφορές στο ίδιο επίπεδο (π.χ. της **f1** στη **v1**, της **f2** στη **v2**, ...) χρησιμοποιούν την τρέχουσα εγγραφή δραστηριοποίησης
- Οι αναφορές σε ονόματα που βρίσκονται σε **κ** επίπεδα φωλιάσματος διατρέχουν **κ** συνδέσμους της αλυσίδας

# Αυτή και άλλες λύσεις

---

- Το πρόβλημα είναι οι αναφορές από εσωτερικές συναρτήσεις σε μεταβλητές κάποιας εξωτερικής
- Λύσεις του προβλήματος αποτελούν οι:
  - **Σύνδεσμοι φωλιάσματος** των εγγραφών δραστηριοποίησης (όπως δείξαμε στις προηγούμενες διαφάνειες)
  - **Displays**: οι σύνδεσμοι φωλιάσματος δε βρίσκονται στις εγγραφές δραστηριοποίησης, αλλά συλλέγονται σε ένα στατικό πίνακα
  - **Λάμβδα άρση (Lambda lifting)**: οι αναφορές του προγράμματος αντικαθίστανται από αναφορές σε νέες, κρυφές παραμέτρους

# Παράδειγμα λάμβδα άρσης

---

```
fun quicksort nil = nil
  | quicksort (pivot::rest) =
    let
      fun split (nil, _) = (nil, nil)
        | split (x::xs, lpv) =
          let
            val (below, above) = split (xs, lpv)
          in
            if x < lpv then (x::below, above)
            else (below, x::above)
          end;
      val (below, above) = split (rest, pivot)
    in
      quicksort below @ [pivot] @ quicksort above
    end;
```

# Συναρτήσεις ως παράμετροι



# Συναρτήσεις ως παράμετροι

---

- Όταν περνάμε μια συνάρτηση ως παράμετρο, τι ακριβώς είναι αυτό που πρέπει να μεταβιβαστεί;
- Ο κώδικας της συνάρτησης πρέπει να είναι μέρος αυτού που περνιέται: είτε σε μορφή πηγαίου κώδικα, ή σε μορφή μεταγλωττισμένου κώδικα, ή ως δείκτης σε κώδικα, ή μέσω υλοποίησης κάποιας άλλης μορφής
- Κάποιες γλώσσες, χρειάζονται κάτι περισσότερο...

# Παράδειγμα

---

```
fun addXToAll (x,theList) =  
  let  
    fun addX y = y + x  
  in  
    map addX theList  
  end;
```

- Η παραπάνω συνάρτηση προσθέτει την τιμή του **x** σε κάθε στοιχείο της λίστας **theList**
- Η **addXToAll** καλεί τη **map**, η **map** καλεί τη **addX**, και η **addX** αναφέρεται στη μεταβλητή **x** στην εγγραφή δραστηριοποίησης της συνάρτησης **addXToAll**

# Σύνδεσμοι φωλιάσματος ξανά

---

- Όταν η `map` καλέσει την `addX`, τι σύνδεσμος φωλιάσματος θα δοθεί στην `addX` ;
  - Όχι η εγγραφή δραστηριοποίησης της `map` διότι η `addX` δεν είναι φωλιασμένη μέσα στη `map`
  - Όχι ο σύνδεσμος φωλιάσματος της `map` διότι η `map` δεν είναι κάπου φωλιασμένη
- Για να δουλέψει η συνάρτηση `addXToAll`, η παράμετρος που περνά η `addX` στη `map` πρέπει να περιλαμβάνει το σύνδεσμο φωλιάσματος που θα χρησιμοποιηθεί όταν κληθεί η `addX`

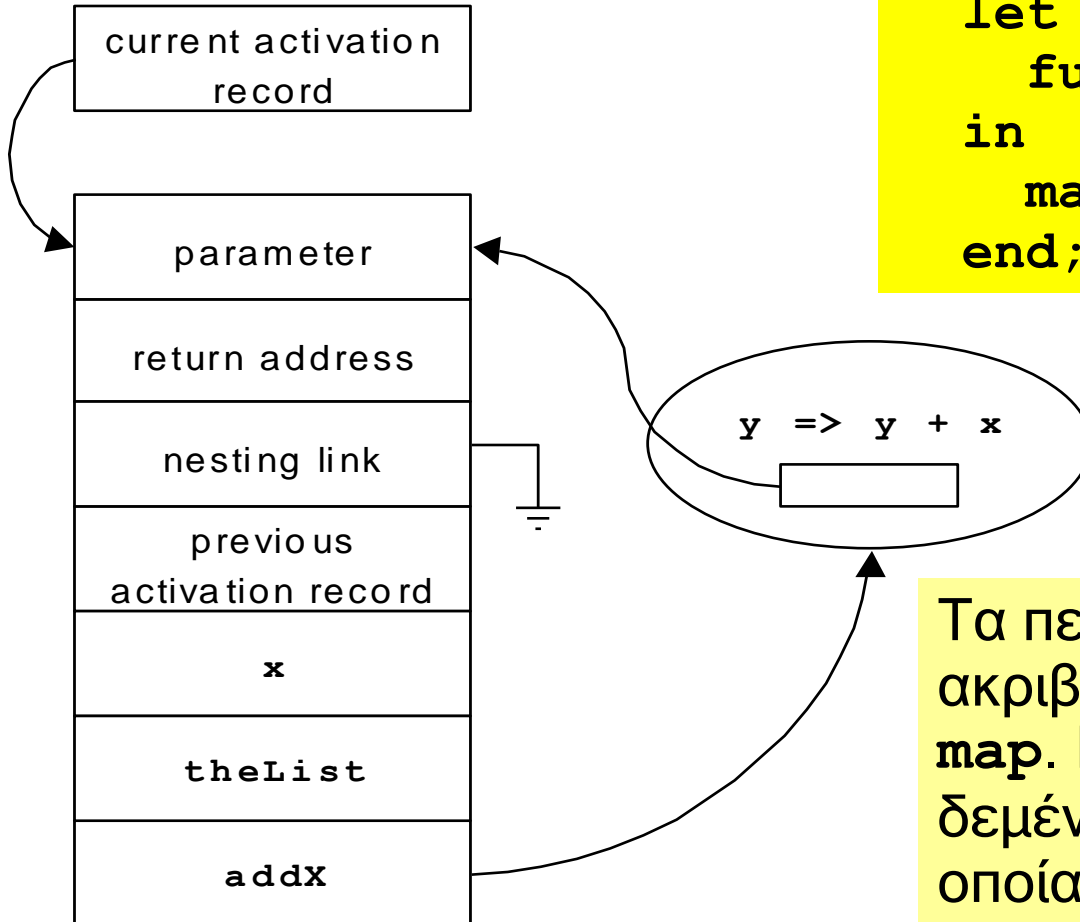
```
fun addXToAll (x, theList) =  
  let  
    fun addX y = y + x  
  in  
    map addX theList  
  end;
```

# Όχι μόνο για τις παραμέτρους

---

- Πολλές γλώσσες επιτρέπουν σε συναρτήσεις να περαστούν ως παράμετροι
- Οι συναρτησιακές γλώσσες επιτρέπουν περισσότερα είδη λειτουργιών σε συναρτήσεις-τιμές:
  - Πέρασμα συναρτήσεων ως παραμέτρους
  - Επιστροφή συναρτήσεων από συναρτήσεις
  - Κατασκευή συναρτήσεων από εκφράσεις
- Οι συναρτήσεις-τιμές περιλαμβάνουν τόσο τον κώδικα που θα εκτελεστεί όσο και το σύνδεσμο φωλιάσματος που θα χρησιμοποιηθεί όταν κληθεί η συνάρτηση

# Παράδειγμα



```
fun addXToAll (x, theList) =  
  let  
    fun addX y = y + x  
  in  
    map addX theList  
  end;
```

Τα περιεχόμενα της μνήμης ακριβώς πριν την κλήση της `map`. Η μεταβλητή `addX` είναι δεμένη με μια συνάρτηση-τιμή η οποία περιλαμβάνει κώδικα και ένα σύνδεσμο φωλιάσματος.

# Μακρόβιες εγγραφές δραστηριοποίησης

## Μια ακόμα περιπλοκή

---

- Τι συμβαίνει εάν μια συνάρτηση-τιμή χρησιμοποιείται από τη συνάρτηση που τη δημιούργησε μετά την επιστροφή της συνάρτησης-δημιουργού;

```
val test =  
  let  
    val f = funToAddX 3  
  in  
    f 5  
  end;
```

```
fun funToAddX x =  
  let  
    fun addX y = y + x  
  in  
    addX  
  end;
```

```

val test =
  let
    val f = funToAddX 3
  in
    f 5
  end;

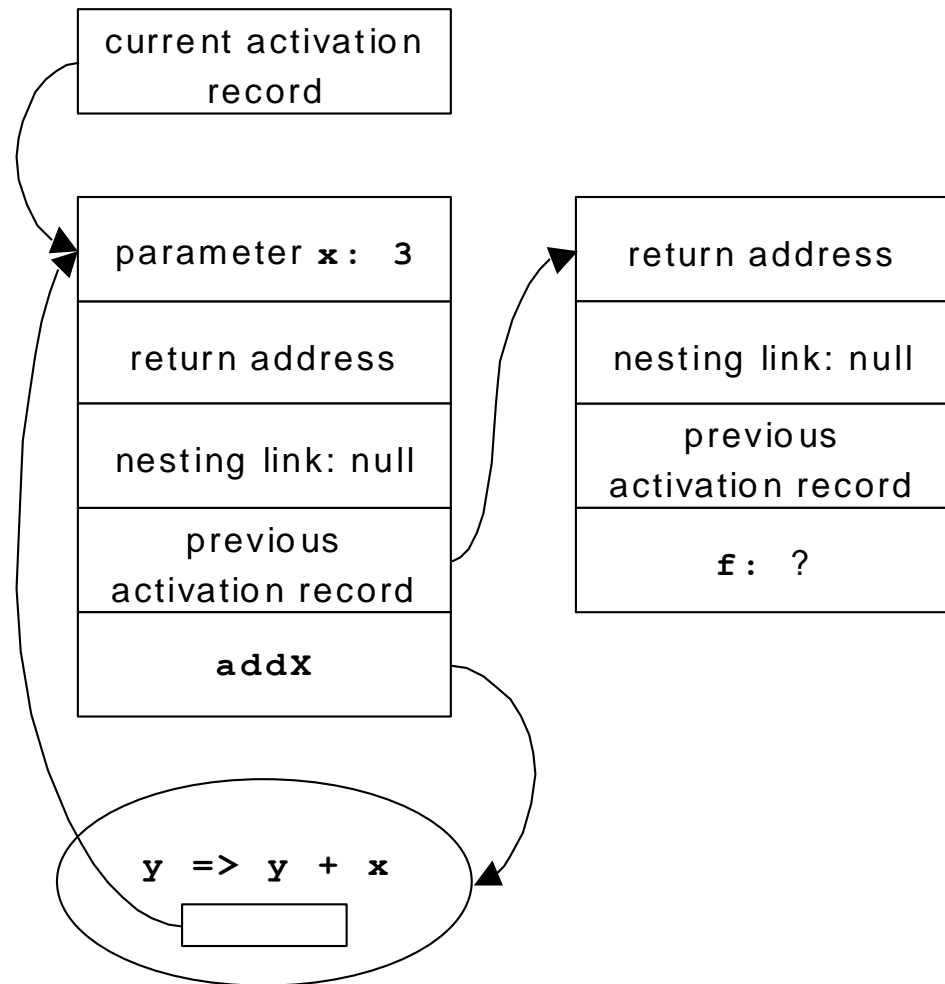
```

```

fun funToAddX x =
  let
    fun addX y = y + x
  in
    addX
  end;

```

Τα περιεχόμενα της μνήμης ακριβώς πριν η `funToAddX` επιστρέψει.





```

val test =
  let
    val f = funToAddX 3
  in
    f 5
  end;

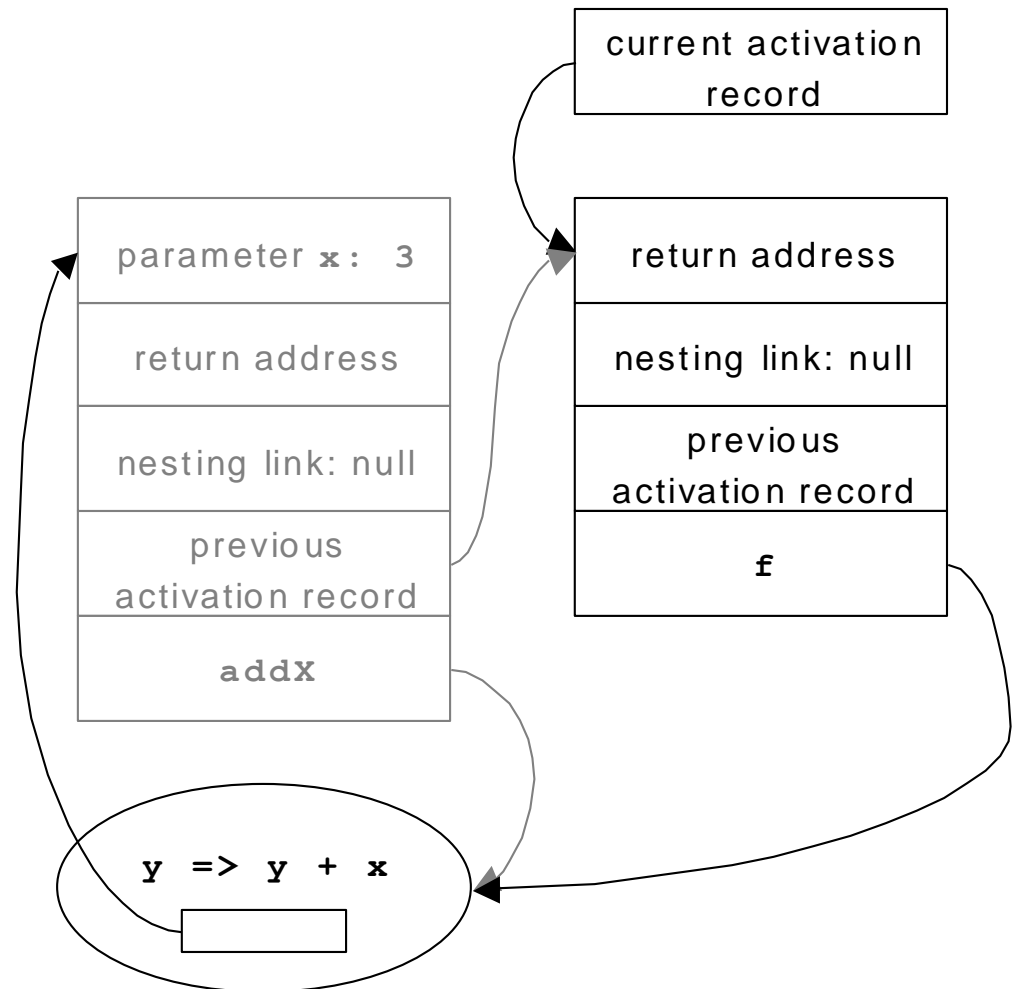
```

```

fun funToAddX x =
  let
    fun addX y = y + x
  in
    addX
  end;

```

Μετά την επιστροφή της `funToAddX`, η `f` δεσμεύεται με τη νέα συνάρτηση-τιμή.



# Το πρόβλημα

---

- Όταν η `test` καλέσει την `f`, η συνάρτηση θα χρησιμοποιήσει το σύνδεσμο φωλιάσματος για να προσπελάσει το `x`
- Με άλλα λόγια, θα χρησιμοποιήσει ένα δείκτη σε μια εγγραφή δραστηριοποίησης για μια δραστηριοποίηση που έχει τερματίσει
- Αυτό θα αποτύχει εάν η υλοποίηση της γλώσσας αποδεσμεύσει την εγγραφή δραστηριοποίησης τη στιγμή επιστροφής της συνάρτησης

# Η λύση του προβλήματος

---

- Στην ML, και σε άλλες γλώσσες όπου υπάρχει η ίδια περιπλοκή, οι εγγραφές δραστηριοποίησης δεν μπορούν πάντα να δεσμευθούν και να αποδεσμευθούν ακολουθώντας δομή στοίβας
- Αυτό διότι ακόμα και όταν μια συνάρτηση επιστρέψει, μπορεί να υπάρχουν σύνδεσμοι στην εγγραφή δραστηριοποίησής της
- Οι εγγραφές δεν μπορεί να αποδεσμευθούν παρά μόνο όταν καταστούν απροσπέλαστες
- Αυτό συνήθως συμβαίνει με **αυτόματη διαχείριση μνήμης** ή αλλιώς με **συλλογή σκουπιδιών (garbage collection)**

# Συμπερασματικά

---

- Όσο πιο “σοφιστικέ” είναι μια γλώσσα, τόσο πιο δύσκολο είναι να δεθούν οι μεταβλητές με θέσεις μνήμης
  - **Στατική δέσμευση**: δουλεύει σε γλώσσες που επιτρέπουν το πολύ μια δραστηριοποίηση κάθε συνάρτησης κάθε στιγμή (όπως σε πρώιμες διαλέκτους της Fortran και της Cobol)
  - **Δυναμική δέσμευση σε στοίβα**: δουλεύει σε γλώσσες που δεν επιτρέπουν φωλιασμένες συναρτήσεις (όπως η C)
  - **Σύνδεσμοι φωλιάσματος** (ή κάτι αντίστοιχο): επιβάλλεται σε γλώσσες που επιτρέπουν φωλιασμένες συναρτήσεις (όπως η ML, η Ada και η Pascal). Οι συναρτήσεις τιμές πρέπει να περιλαμβάνουν τόσο κώδικα όσο και σύνδεσμο φωλιάσματος
  - Κάποιες γλώσσες (όπως η ML) επιτρέπουν αναφορές σε εγγραφές δραστηριοποιήσεων συναρτήσεων που έχουν περατώσει την εκτέλεσή τους. Κατά συνέπεια, οι εγγραφές δραστηριοποίησης δε μπορούν να αποδεσμευθούν αμέσως μετά την επιστροφή τους και η χρήση στοίβας δεν επαρκεί.