Διαδικαστικός Προγραμματισμός

Βασίλης Παλιουράς
paliuras@ece.upatras.gr

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct x {
          int a;
          struct x * next; } X;
typedef X * X_ptr;
typedef X * List;

X_ptr createnode (int a) {
    X_ptr temp ;
    temp = malloc(sizeof (X_ptr));
    temp ->   a = a;
    temp ->next = NULL;
    return temp;
}


void append( List * x, X_ptr a) {
    if (*x == NULL) {
        *x = a;
        return ;
        }
    append (& ( ((*x)->next)), a);
    return ;
}

List mymain () {
    static List y  = NULL;
    X_ptr a ;
    int x;
    scanf("%d", x);
    a = createnode(1);

    append(&y, a);
    return y;
}
```

Μερικές αναδρομικές συναρτήσεις είναι tail recursive:

Το trace back δεν συνδράμει στη λύση

Το αποτέλεσμα υπολογίζεται στην ευθεία διαδρομή

Κατά το trace back ολοκληρώνονται τα return

```
void append( List * x, X_ptr a) {
    if (*x == NULL) {
        *x = a;
        return ;
        }
    append (& ( ((*x)->next)), a);
    return ;
}
```

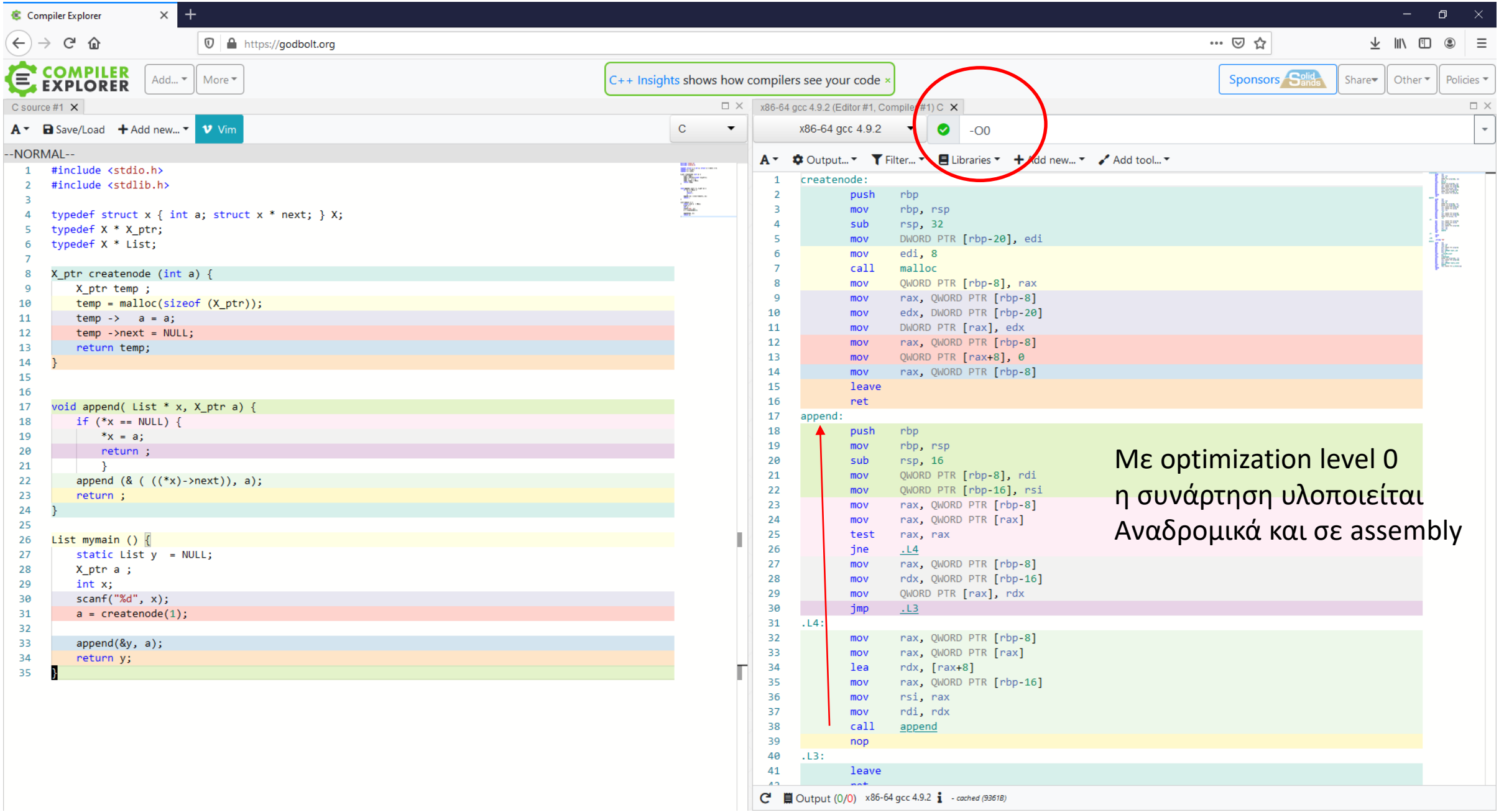
```
void append( List * x, X_ptr a) {
    while (*x != NULL) {
        *x = (*x) -> next;
        }
    *x = a;
    return ;
}
```

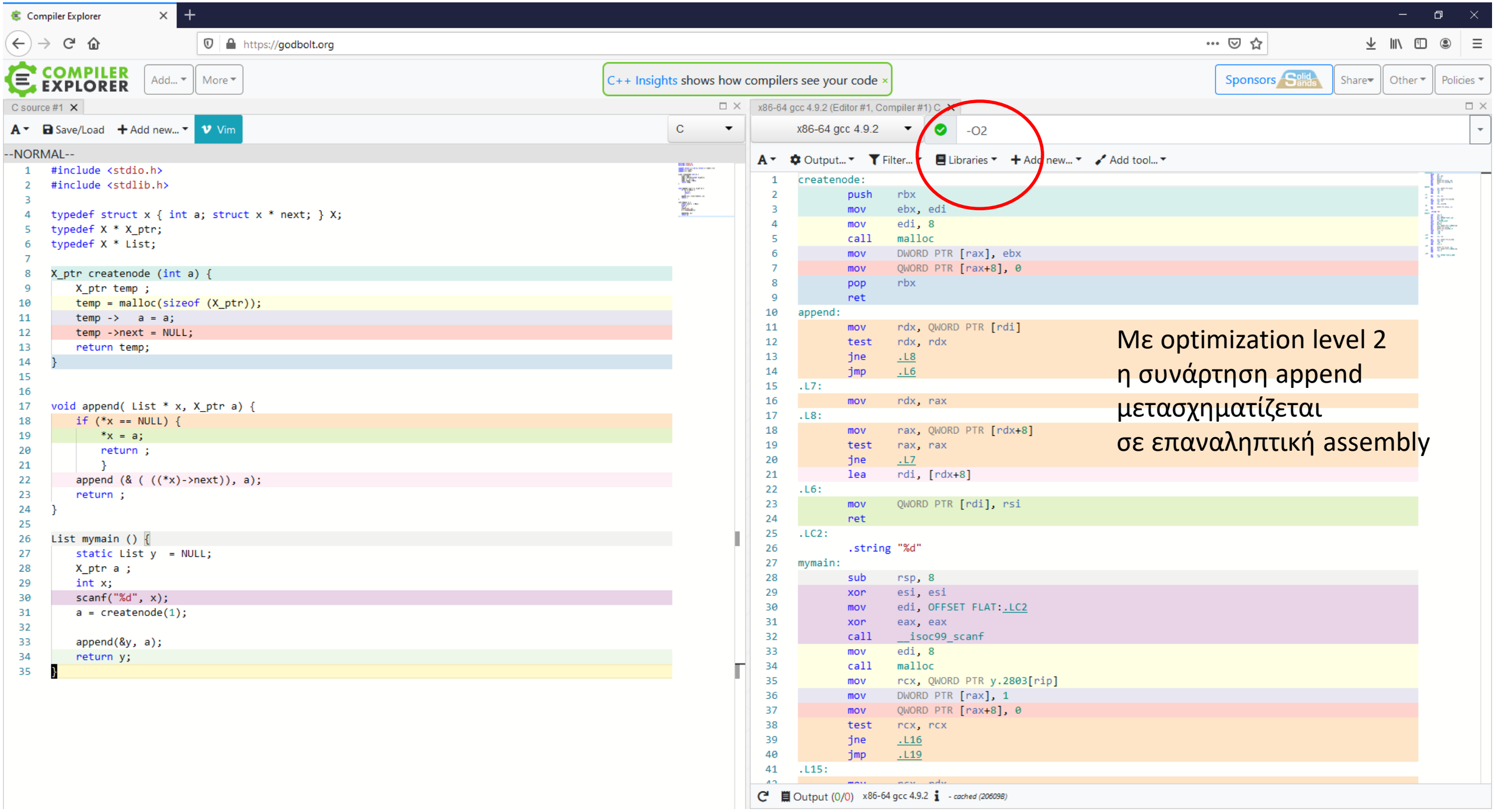
Μερικές αναδρομικές συναρτήσεις είναι tail recursive:
Δεν συνδράμει στη λύση το trace back

Η αναδρομική κλήση είναι το τελευταίο πράγμα πριν το return ή είναι η μόνη έκφραση του return (πχ return f(x-1);)

Μετασχηματίζονται σε επαναληπτικές λύσεις

Οι compilers εφαρμόζουν τέτοιες μετατροπές.


Compiler Explorer
https://godbolt.org
COMPILER EXPLORER
Add...
More
C++ Insights shows how compilers see your code
Sponsors
Share
Other
Policies
C source #1
Save/Load
Add new...
Vim
C
--NORMAL--
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct x { int a; struct x * next; } X;
5 typedef X * X_ptr;
6 typedef X * List;
7
8 X_ptr createnode (int a) {
9 X_ptr temp ;
10 temp = malloc(sizeof (X_ptr));
11 temp -> a = a;
12 temp ->next = NULL;
13 return temp;
14 }
15
16
17 void append(List * x, X_ptr a) {
18 if (*x == NULL) {
19 *x = a;
20 return ;
21 }
22 append (& (((*x)->next)), a);
23 return ;
24 }
25
26 List mymain () {
27 static List y = NULL;
28 X_ptr a ;
29 int x;
30 scanf("%d", x);
31 a = createnode(1);
32
33 append(&y, a);
34 return y;
35 }
x86-64 gcc 4.9.2 (Editor #1, Compile #1) C
x86-64 gcc 4.9.2
-O0
Output...
Filter...
Libraries
Add new...
Add tool...
1 createnode:
2 push rbp
3 mov rbp, rsp
4 sub rsp, 32
5 mov DWORD PTR [rbp-20], edi
6 mov edi, 8
7 call malloc
8 mov QWORD PTR [rbp-8], rax
9 mov rax, QWORD PTR [rbp-8]
10 mov edx, DWORD PTR [rbp-20]
11 mov DWORD PTR [rax], edx
12 mov rax, QWORD PTR [rbp-8]
13 mov QWORD PTR [rax+8], 0
14 mov rax, QWORD PTR [rbp-8]
15 leave
16 ret
17 append:
18 push rbp
19 mov rbp, rsp
20 sub rsp, 16
21 mov QWORD PTR [rbp-8], rdi
22 mov QWORD PTR [rbp-16], rsi
23 mov rax, QWORD PTR [rbp-8]
24 mov rax, QWORD PTR [rax]
25 test rax, rax
26 jne .L4
27 mov rax, QWORD PTR [rbp-8]
28 mov rdx, QWORD PTR [rbp-16]
29 mov QWORD PTR [rax], rdx
30 jmp .L3
31 .L4:
32 mov rax, QWORD PTR [rbp-8]
33 mov rax, QWORD PTR [rax]
34 lea rdx, [rax+8]
35 mov rax, QWORD PTR [rbp-16]
36 mov rsi, rax
37 mov rdi, rdx
38 call append
39 nop
40 .L3:
41 leave
Output (0/0) x86-64 gcc 4.9.2 - cached (93618)
Με optimization level 0
η συνάρτηση υλοποιείται
Αναδρομικά και σε assembly




```c
#include <stdio.h>
#include <stdlib.h>

typedef struct x { int a; struct x * next; } X;
typedef X * X_ptr;
typedef X * List;

X_ptr createnode (int a) {
    X_ptr temp ;
    temp = malloc(sizeof (X_ptr));
    temp ->   a = a;
    temp ->next = NULL;
    return temp;
}


void append( List * x, X_ptr a) {
    if (*x == NULL) {
        *x = a;
        return ;
        }
    append (& ( ((*x)->next)), a);
    return ;
}

List mymain () {
    static List y  = NULL;
    X_ptr a ;
    int x;
    scanf("%d", x);
    a = createnode(1);

    append(&y, a);
    return y;
}
```

```asm
createnode:
        push    rbx
        mov     ebx, edi
        mov     edi, 8
        call    malloc
        mov     DWORD PTR [rax], ebx
        mov     QWORD PTR [rax+8], 0
        pop     rbx
        ret
append:
        mov     rdx, QWORD PTR [rdi]
        test    rdx, rdx
        jne     .L8
        jmp     .L6
.L7:
        mov     rdx, rax
.L8:
        mov     rax, QWORD PTR [rdx+8]
        test    rax, rax
        jne     .L7
        lea     rdi, [rdx+8]
.L6:
        mov     QWORD PTR [rdi], rsi
        ret
.LC2:
        .string "%d"
mymain:
        sub     rsp, 8
        xor     esi, esi
        mov     edi, OFFSET FLAT:.LC2
        xor     eax, eax
        call    __isoc99_scanf
        mov     edi, 8
        call    malloc
        mov     rcx, QWORD PTR y.2803[rip]
        mov     DWORD PTR [rax], 1
        mov     QWORD PTR [rax+8], 0
        test    rcx, rcx
        jne     .L16
        jmp     .L19
.L15:
```

- Να γραφεί μια συνάρτηση που να αθροίζει ακεραίους αποθηκευμένους σε απλά διασυνδεδεμένη λίστα
  - Με αναδρομική συνάρτηση
  - Με tail-recursive αναδρομική συνάρτηση
  - Με επανάληψη

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
    int data;
    struct node * next ;
} Node;
typedef Node * Node_ptr;
typedef Node * List;

Node_ptr create(int data) ;
List prepend(List lst, Node_ptr new_node_ptr);
void report (List lst) ;

int sum(List lst);
int sumtail(int temp, List lst);
int sumwrap(List lst);
```

- Ορίζουμε
  - βασικούς τύπους
  - Πρότυπα συναρτήσεων

```
Node_ptr create(int data) {
    Node_ptr new_node_ptr = malloc (sizeof (Node));
    if (new_node_ptr==NULL) return new_node_ptr;

    new_node_ptr->data = data;
    new_node_ptr->next = NULL;

    return new_node_ptr;

}


List prepend(List lst, Node_ptr new_node_ptr) {
    if (new_node_ptr==NULL) return lst;

    new_node_ptr->next = lst;

    return new_node_ptr;
}
```

```
void report(List lst) {

    while(lst!=NULL) {
        printf("%d\t", lst->data);
        lst = lst->next;
    }

    return;
}
```

```
int main()
{
    List lst = NULL;

    lst=prepend(lst, create(1));
    lst=prepend(lst, create(2));
    lst=prepend(lst, create(3));

    report(lst);

    printf("\n%d", sum(lst));
    printf("\n%d", sumtail(0,lst));
    printf("\n%d", sumwrap(lst));

    return 0;
}
```

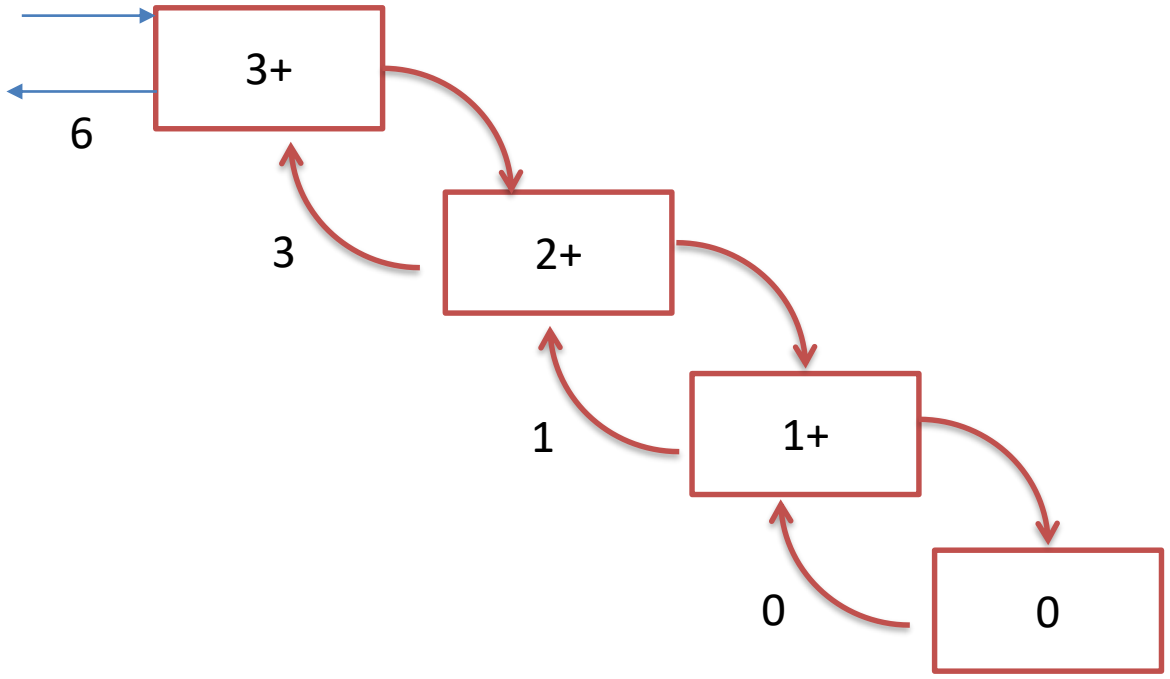
Δήλωση κενής λίστας

Η συνάρτηση create δημιουργεί έναν κόμβο στο heap, και επιστρέφει δείκτη σε αυτόν.

Η συνάρτηση prepend αξιοποιεί τον δείκτη σε κόμβο που επιστρέφει η create, για να τοποθετήσει τον κόμβο πρώτο στη λίστα.

```
int sum(List lst) {
    if (lst==NULL) return 0;

    return lst->data + sum(lst->next);
}
```

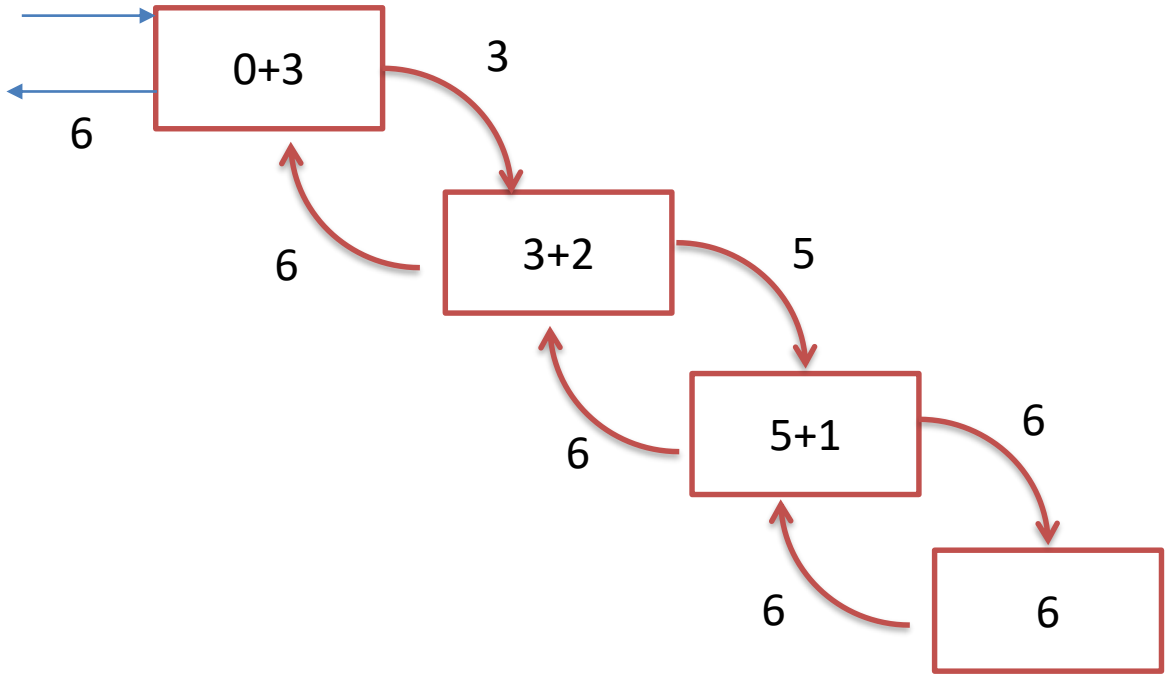


- Η λίστα περιλαμβάνει τα στοιχεία 3 -> 2 -> 1
- Το αποτέλεσμα υπολογίζεται κατά το back tracking
- Είναι αναγκαίο να μένουν ανοικτές οι κλήσεις μέχρι την ολοκλήρωση του υπολογισμού

Tail-recursive αναδρομή

```
int sumtail(int temp, List lst) {
    if (lst==NULL) return temp;

    return sumtail(temp+lst->data, lst->next);
}
```



- Το αποτέλεσμα υπολογίζεται κατά την ευθεία διαδοχή κλήσεων
- Κατά την επιστροφή, δεν γίνονται υπολογισμοί (άρα μπορεί να αποφευχθεί)
- Αυτόματη αντιστοίχιση σε iterative υπολογισμό

Αντιστοίχιση tail-recursive αναδρομής και επανάληψης

```
int sumtail(int temp, List lst) {
    if (lst==NULL) return temp;

    return sumtail(temp+lst->data, lst->next);
}
```

Η αρχικοποίηση του temp γίνεται στην κλήση της sumtail

```
int sumiter(List lst) {
    int temp = 0;
    while (lst!=NULL) {
        temp = temp + lst->data;
        lst = lst->next;
    }
    return temp;
}
```

Wrapper functions

- Για να μην αλλάξει η λίστα παραμέτρων στην κλήση συνάρτησης, χρησιμοποιούμε wrapper functions

```
int sumwrap(List lst) {
    return sumtail(0, lst);
}
```

- Η αρχικοποίηση του `temp` της `sumtail` κρύβεται στην κλήση της `sumwrap`

Tail-recursive άθροιση στοιχείων πίνακα

```
int main () {
      int arr[3] = {1,2,3};
      printf("\n%d", sumarray(0,arr,3));
      return 0;
   }
```

```
int sumarray(int temp, int *array, int len) {
    if (len==0) return temp ;

    return sumarray (temp + *array, array+1, len - 1);
}
```

Χρησιμοποιώ αναδρομή.
Δεν αποθηκεύω αναλυτικά την τιμή της εξόδου.
Η κάθε πύλη ρωτάει τις προηγούμενες πύλες για να μάθει τις τιμές των εξόδων τους.
ΜΗΝ ΧΡΗΣΙΜΟΠΟΙΗΣΕΤΕ ΑΥΤΌΝ ΤΟΝ ΚΩΔΙΚΑ ΩΣ ΕΧΕΙ!!!

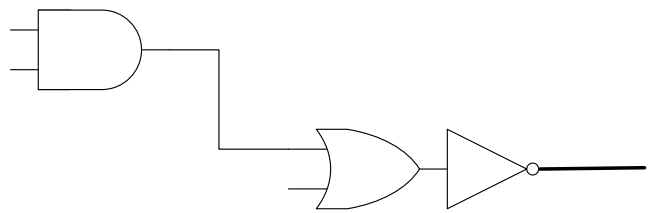
ΜΙΑ ΛΥΣΗ

```c
#include <stdio.h>
#include <stdlib.h>
int myand (int, int);
int mynot (int);
int myor (int, int);
int input(void);
typedef int (*Gatetype)();

typedef struct gate {
        Gatetype type;
        struct gate * in1;
        struct gate * in2;
        } Gate;
int evaluate(Gate);
```

Ορισμός τύπου δείκτη σε συνάρτηση της οποίας δεν μας ενδιαφέρει η λίστα παραμέτρων και η οποία επιστρέφει ακέραιο.

```
int main(void) {
  int d;
  Gate input1 = { input, NULL, NULL};
  Gate input2 = { input, NULL, NULL};
  Gate input3 = { input, NULL, NULL};
  Gate g1 = {and, &input1, &input2};
  Gate g2 = { or, &g1, &input3};
  Gate g3 = {not, &g2, NULL};
  d = evaluate(g3);
  printf("result: %d\n", d);
  return 0;
}
```



```c
int myand (int a, int b) {
    return a * b;
}

int myor (int a, int b) {
    return a+b>0;
}
int mynot (int a) {
    return 1 -a ;
}

int input (void) {
    int a;
    printf("Enter input: ");
    scanf("%d", &a);
    return a;
}
```

```
int evaluate(Gate g) {
    int out;
    int a1=-1, a2=-1;

    if (g.in1!=NULL)
       a1 = evaluate(*g.in1);
    if (g.in2!=NULL)
       a2 = evaluate(*g.in2);

    out = (*g.type)(a1,a2);

 return out;
}
```

Από τη λίστα παραμέτρων η κάθε συνάρτηση θα χρησιμοποιήσει όσες χρειάζεται!

Το dereference δεν είναι αναγκαίο σε function pointer

```c
#include <stdio.h>
#include <stdlib.h>
#define AND 0
#define OR 1
#define NOT 2
#define INPUT 3

int myand (int, int);
int mynot (int);
int myor  (int, int);
int input (void);

typedef struct gate {
        int (*type)();
        struct gate * in1;
        struct gate * in2;
        } Gate;

typedef struct gate4file {
                int type;
                int in1;
                int in2;
} Gate4file;
```

```
void file2eval ( Gate g[], const Gate4file gf[], int gates)
{
        int i;

        int (*f[])()= {and, or, not, input};

        for (i=0;i<gates; i++) {
                g[i].type = f[gf[i].type];
                g[i].in1 = gf[i].in1!=-1?&g[gf[i].in1]:NULL;
                g[i].in2 = gf[i].in2!=-1?&g[gf[i].in2]:NULL;
        }

}
```

```
int main( ) {
  int d;
  FILE *f;

  Gate g[6];
  Gate h[6];
  g[0].type= input; g[0].in1= NULL; g[0].in2= NULL;     /* input 1 */
  g[1].type= input; g[1].in1= NULL; g[1].in2= NULL;     /* input 2 */
  g[2].type= input; g[2].in1= NULL; g[2].in2= NULL;     /* input 3 */
  g[3].type= and;   g[3].in1= &g[0];g[3].in2= &g[1];    /* g1 */
  g[4].type= or;   g[4].in1= &g[3];g[4].in2= &g[2];     /* g2 */
  g[5].type= not;  g[5].in1= &g[4];g[5].in2= NULL;      /* g3 */

  Gate4file gf[6] = { {INPUT, -1, -1},
                      {INPUT, -1, -1},
                      {INPUT, -1, -1},
                      {AND,  0,  1},
                      {OR,  3,  2},
                      {NOT,  4, -1}};
  Gate4file hf[6];
  f = fopen("circ2.dat", "wb");
  fwrite(gf, sizeof(Gate4file), 6, f);
  fclose(f);
  f = fopen("circ2.dat", "rb");
  fread(hf, sizeof(Gate4file), 6, f);
  fclose(f);
  file2eval(h, hf, 6);
  d = evaluate(h[5]);
  printf("result: %d\n", d);
  return 0;
}
```