ΗΥ340 : ΓΛΩΣΣΕΣ ΚΑΙ ΜΕΤΑΦΡΑΣΤΕΣ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

```
VAR i:Integer;

    FUNCTION(Symbol) replicate

            x = (function(x,y){return x+y;});

              class DelFunctor: public std::unary_function<
```

ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης

Φροντιστήριο 5ο
Τελικός κώδικας και εικονική μηχανή

Μέρος 1ο
Περιβάλλον εκτέλεσης

Αντιστοίχιση μεταβλητών σε θέση μνήμης

- Στον ενδιάμεσο κώδικα, τα περισσότερα ορίσματα τύπου *expr** (εκτός από τις σταθερές και συναρτήσεις) αντιστοιχούν πάντα σε κάποιο **σύμβολο** του προγράμματος μας (πεδίο *sym* του *expr*)
- Αυτό το σύμβολο θα πρέπει να αντιστοιχηθεί στο περιβάλλον εκτέλεσης σε κάποια συγκεκριμένη θέση μνήμης
- Η αντιστοίχιση αυτή σχετίζεται με:
  - Το *scope space* του συμβόλου
  - Το *offset* που έχει το σύμβολο

Οργάνωση Μνήμης (1/3)

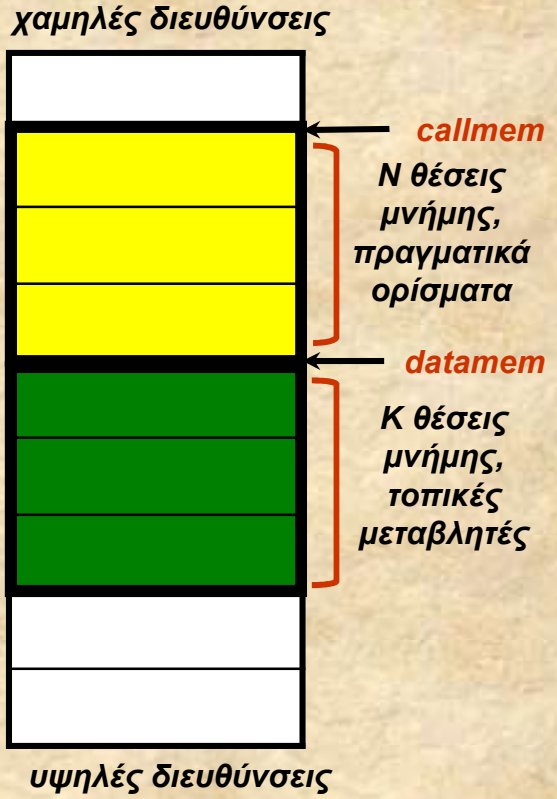
- **Μεταβλητές καθολικής εμβέλειας**
  - Το πλήθος τους είναι γνωστό σε *compile-time*, οπότε δεσμεύουμε για αυτές ένα συνεχόμενο τμήμα μνήμης
  - Ένας καταχωρητής (*globalmem*) της εικονικής μηχανής θα υποδεικνύει την αρχική διεύθυνση αυτού του τμήματος
  - Αναφερόμαστε σε μια global μεταβλητή **x** ως

mem[globalmem + x→offset]

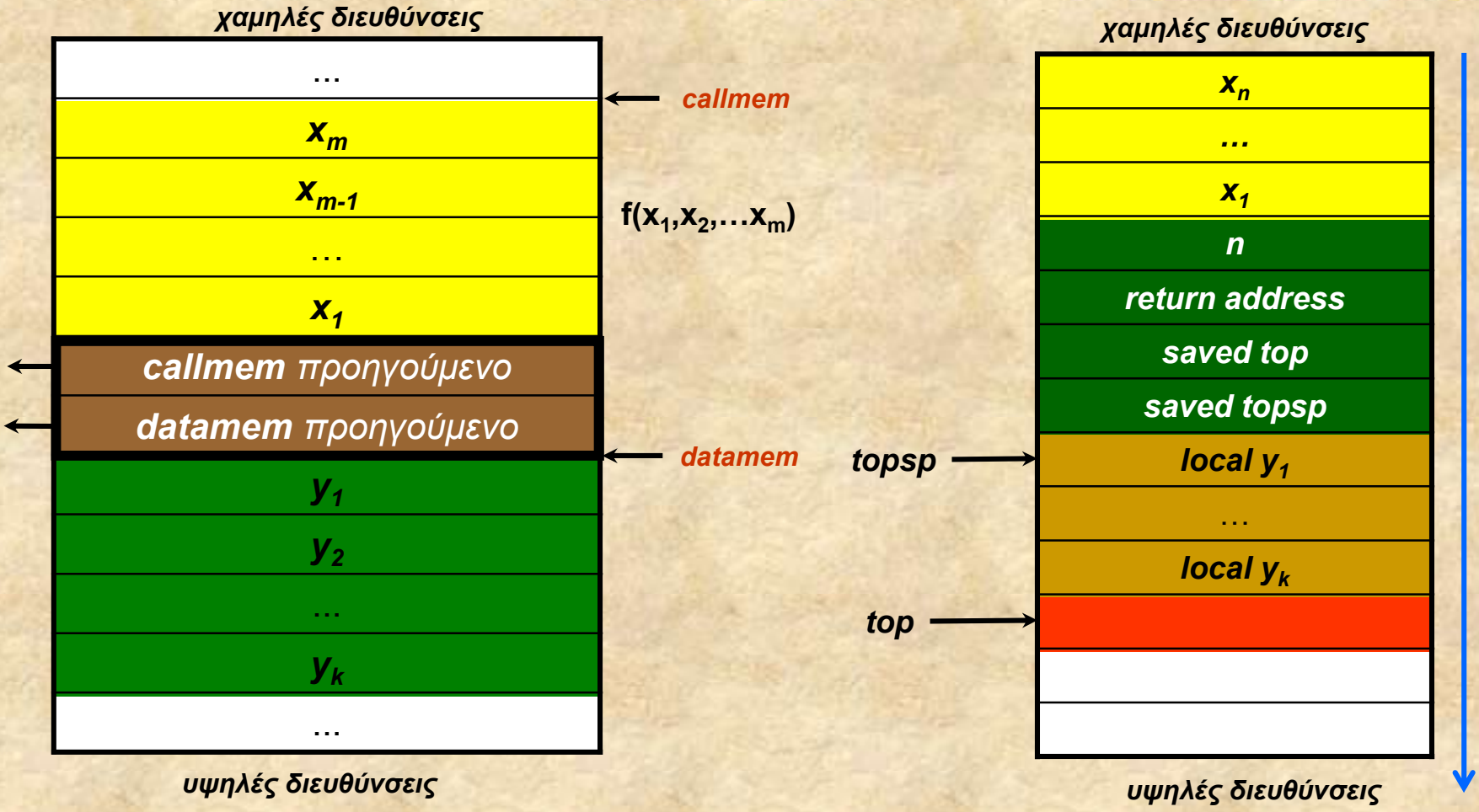
Οργάνωση Μνήμης (2/3)

Τυπικά ορίσματα και τοπικές μεταβλητές

- Για την κλήση μιας συνάρτησης χρειαζόμαστε χώρο για τα πραγματικά ορίσματα και τις τοπικές μεταβλητές της
- Η συνάρτηση που καλείται **δεν** είναι γνωστή σε compile-time άρα η αντιστοίχιση θα πρέπει να γίνει **δυναμικά**
- Δεσμεύουμε N θέσεις για τα πραγματικά ορίσματα και K θέσεις για τις τοπικές μεταβλητές
  - Αναφερόμαστε σε τυπικό όρισμα **x** ως
    - ***mem[datamem –x→offset-1]***
  - Αναφερόμαστε σε τοπική μεταβλητή **y** ως
    - ***mem[datamem +y→offset-1]***
- Όταν έχουμε εμφωλευμένες (ή αναδρομικές) κλήσεις συναρτήσεων τι κάνουμε?
  - Πρέπει να σώσουμε τους καταχωρητές
  - Ώστε να τους επαναφέρουμε μετά



Οργάνωση Μνήμης (3/3)

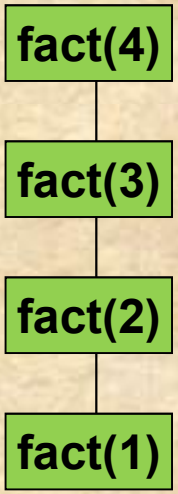


Προσοχή: *Επειδή προσθέσαμε επιπλέον πράγματα αλλάζει λίγο ο υπολογισμός θέσης ενός τυπικού ορίσματος ή μιας τοπικής μεταβλητής*

Δέντρο ενεργοποίησης

```
function fact (n) {
        if (n == 1)
            return 1;
        else
            return n * fact(n-1);
}
```

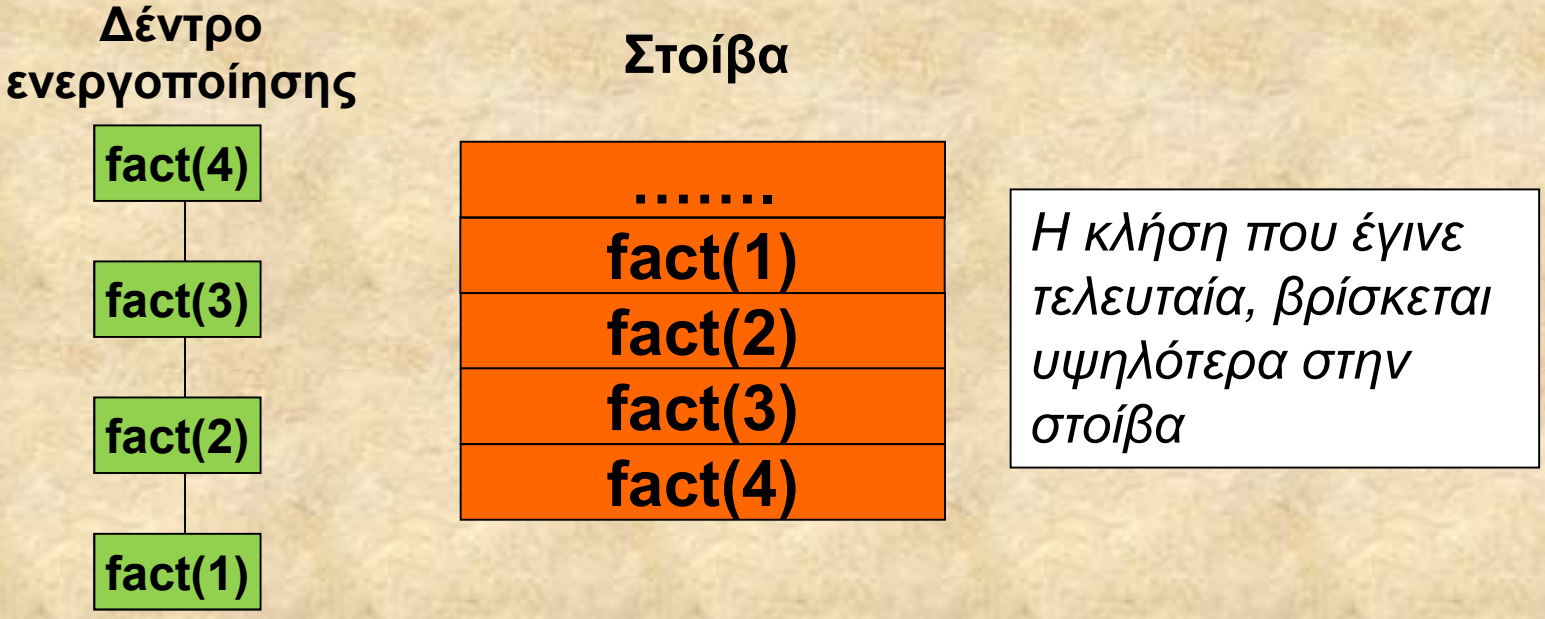
Δέντρο ενεργοποίησης



Κάθε ενεργοποίηση απαιτεί διαφορετικό περιβάλλον, το μέγεθος του οποίου ισούται με: ***αριθμός ορισμάτων + αριθμός τοπικών μεταβλητών***

Activation records (1/5)

- Κατά την κλήση μιας συνάρτησης, παραχωρείται μνήμη δυναμικά (και γίνεται push στην στοίβα), ενώ μετά το τέλος της κλήσης αυτή η μνήμη απελευθερώνεται (και γίνεται pop από την στοίβα το αντίστοιχο active record).



Activation records (2/5)

- Παράδειγμα (code)

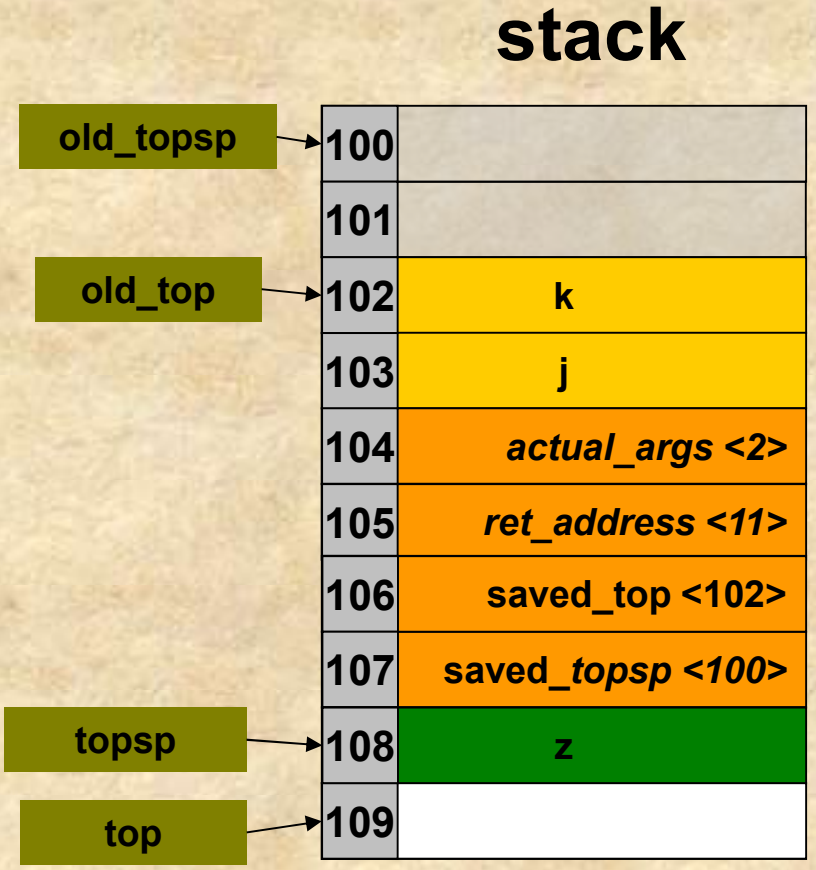
```
function f (x, y) {
        z = x * y;
        if (z == 10) {
                z = 1;
        }
        return z;
}
i = f (j, k);
```

```
1:      funcstart   f
2:      mul  x y  z
3:      if_eq   z  10  5
4:      jump 6
5:      assign  1   z
6:      return   z
7:      funcend  f
8:      param k
9:      param j
10:     call f
11:     getretval  _t1
12:     assign  _t1  i
```

Activation records (3/5)

- Παράδειγμα (stack)

```
function f (x, y) {
        z = x * y;
        if (z == 10) {
                z = 1;
        }
        return z;
}
i = f (j, k);
```

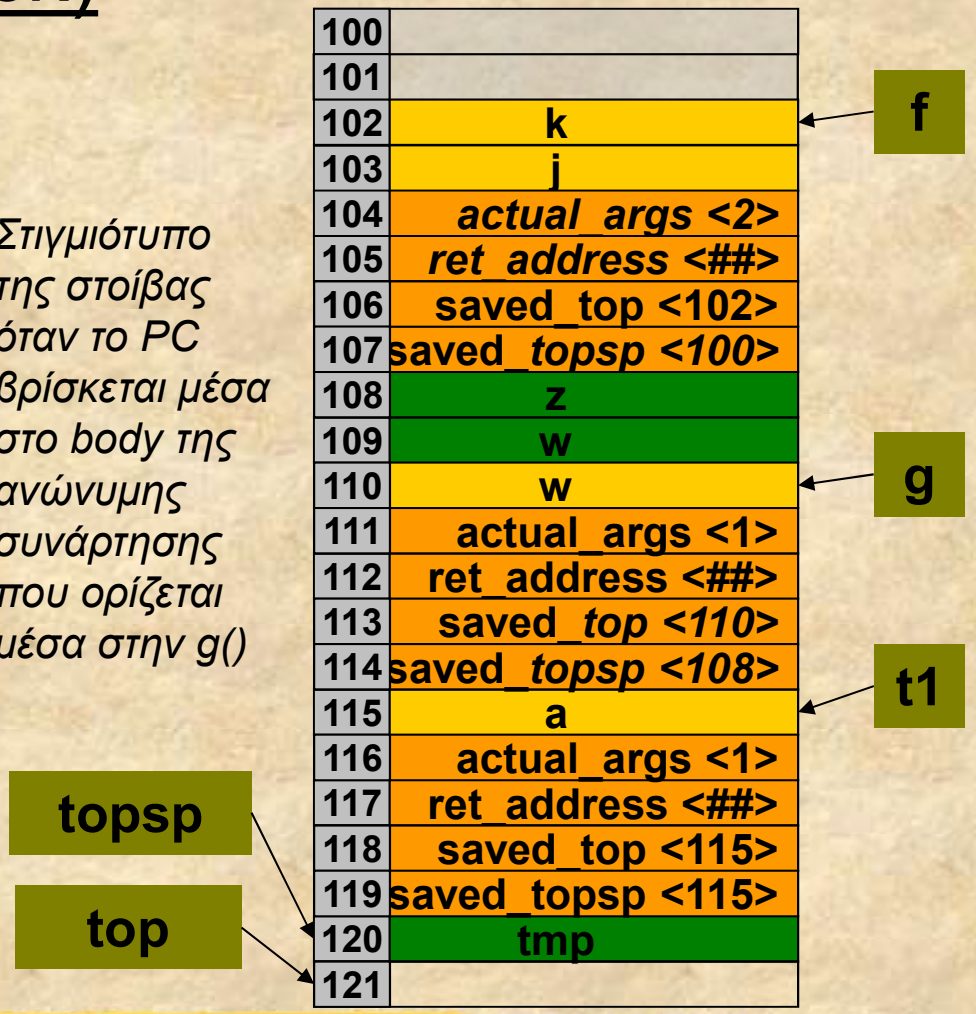


Activation records (4/5)

- Παράδειγμα 2 (stack)

```
function g (a) {
        return (function (x){
            tmp = 2;
            return (tmp – x);
        } )(a);
}

function f (x, y) {
        z = 2*x + y;
        w = z – x/y;
        g(w);
        return z;
}

i = f (j, k);
```

Στιγμιότυπο της στοίβας όταν το PC βρίσκεται μέσα στο body της ανώνυμης συνάρτησης που ορίζεται μέσα στην g()

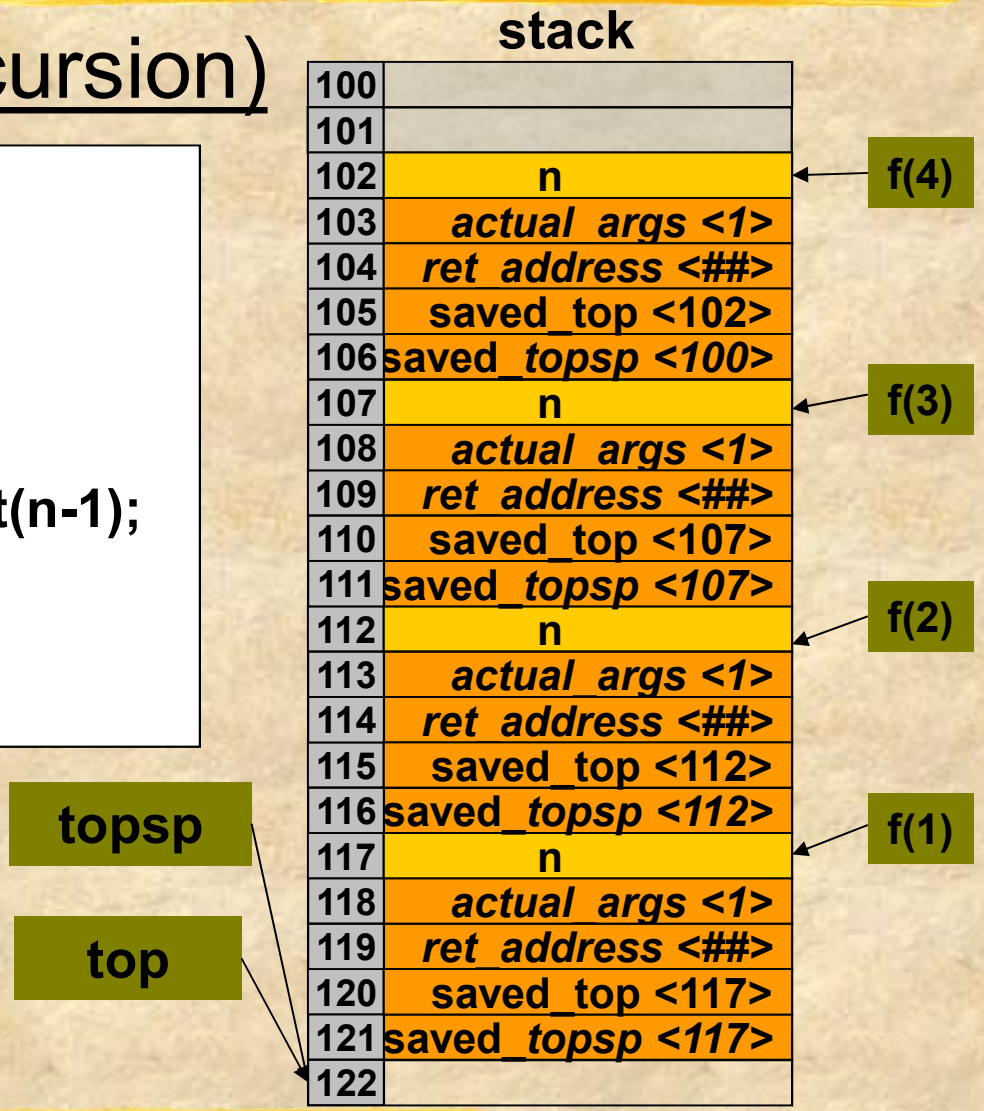


Activation records (5/5)

- Παράδειγμα 3 (recursion)

```
function fact (n) {
        if (n == 1)
                return 1;
        else
                return n * fact(n-1);
}

x = fact(4);
```

Στιγμιότυπο της στοίβας όταν το PC βρίσκεται μέσα στο body της αναδρομικής fact() με όρισμα n=1 (αρχική κλήση n=4)



Κλήση συνάρτησης (1/2)

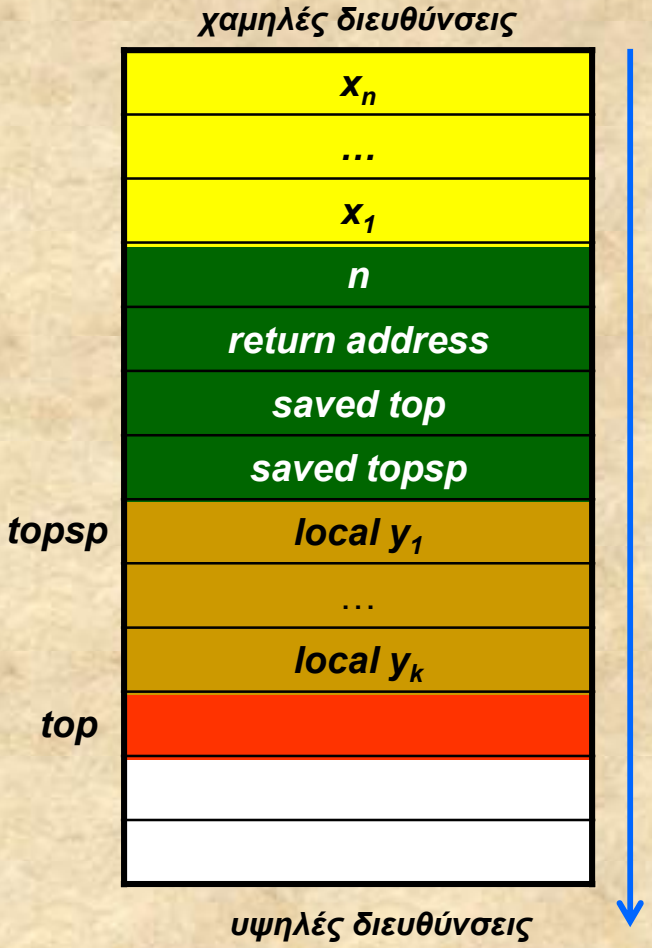
Ακολουθία κλήσης. Χρησιμοποιείται ο δείκτης *top*, ο οποίος σε κάθε push μειώνεται κατά 1 (θεωρούμε τη στοίβα να μεγαλώνει από υψηλές προς χαμηλές διευθύνσεις).

- Ενέργειες του κλητευτή
  - Αποτίμησε τα *n* πραγματικά ορίσματα ($x_1,\ldots,x_n$)
  - Τοποθέτησε τα ορίσματα στη στοίβα με τη σειρά $x_n \ldots x_1$
  - Τοποθέτησε στη στοίβα τον αριθμό των ορισμάτων *n*
  - Τοποθέτησε στη στοίβα τη διεύθυνση επιστροφής από κλήση
  - Τοποθέτησε στη στοίβα την τιμή του top πριν αρχίσει η ακολουθία κλήσης, δηλ. *top*+*n*+2
  - Τοποθέτησε στη στοίβα την τιμή του *topsp*
  - Θέσε τον μετρητή προγράμματος στη διεύθυνση της συνάρτησης

call

- Ενέργειες του καλούμενου
  - Θέσε το topsp ίσο με την τιμή του top
  - Ανέβασε το top τόσες θέσεις όσες οι τοπικές μεταβλητές

funcstart



Κλήση συνάρτησης (2/2)

Ακολουθία επιστροφής από κλήση. **funcend**

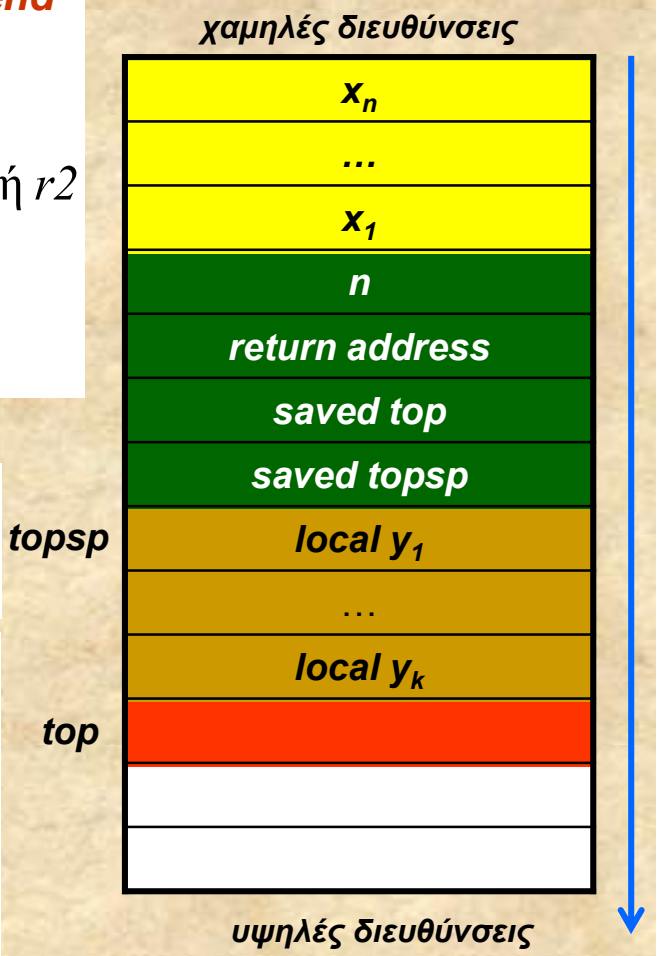
- Αποθήκευσε την τιμή του *top* σε έναν καταχωρητή *r1*
- Επανέφερε το *top* στην προηγούμενη σωμένη τιμή
- Αποθήκευσε την διεύθυνση επιστροφής σε έναν καταχωρητή *r2*
- Επανέφερε την προηγούμενη διεύθυνση του *topsp*
- Καθάρισε την στοίβα από το *r1-1*+έως και το *top*
- Θέσε *pc = r2*

Η ακολουθία εντολών επιστροφής από κλήση σχεδιάζεται έτσι ώστε να επιστρέψουμε στην εντολή ακριβώς μετά την κλήση, με *top* και *topsp* να δείχνουν στο activation record του κλητευτή.

Εναπόθεση επιστρεφόμενου αποτελέσματος.

- Αποθήκευσε το αποτέλεσμα σε έναν ειδικό καταχωρητή *retval* η σε μία θέση στην στοίβα που κρατείται για το σκοπό αυτό ακριβώς πριν το πρώτο πραγματικό όρισμα.
- *Jump* στο τέλος της συνάρτησης

return



Μέρος 2ο
Παραγωγή τελικού κώδικα

Παραγωγή τελικού κώδικα (1/2)

- Το ρεπερτόριο των εντολών του ενδιάμεσου κώδικα είναι αρκετά κοντά σε αυτό του τελικού κώδικα
  - Φεύγει η uminus που μπορεί να υλοποιηθεί με πολλαπλασιασμό με το -1
  - Φεύγουν οι λογικοί τελεστές (ολική αποτίμηση) καθώς υλοποιούνται με jumps και assignments
- Πρέπει να κωδικοποιήσουμε τις εντολές σε ένα συγκεκριμένο αριθμό από bytes
  - 4 bytes ανά εντολή, άρα 1 byte ανά όρισμα
  - Όμως ορισμένα ορίσματα δεν χωράνε
  - Χρησιμοποιούμε πίνακες σταθερών
    - Δημιουργούμε πίνακες με τις εμφανίσεις των σταθερών που εμφανίζονται στις εντολές
    - Αντικαθιστούμε τις εντολές με το index τους μέσα στον πίνακα

Παραγωγή τελικού κώδικα (2/2)

```
x = 10;
y = "hello";
function f(x,y)   { return x+y; }
z = "world";
w = cos(3.1415);
print(x,y,z,w);
```

Πίνακας σταθερών strings	
0	"hello"
1	"world"

Πίνακας αριθμητικών σταθερών	
0	10
1	3.1415

Πίνακας συναρτήσεων χρήστη	
0	*address* 3, *local size* 1, *id* "f"

Πίνακας συναρτήσεων βιβλιοθήκης	
0	"cos"
1	"print"

Κάθε όρισμα έχει δύο πεδία: (α) τύπο ορίσματος, και (β) την πληροφορία του ορίσματος. Στον παρακάτω κώδικα της εικονικής μηχανής αναγράφεται για κάθε όρισμα ο τύπος αλλά και η περίπτωση του ορίσματος στην οποία αναφέρεται.

0	**assign**	01(global), 0:x	04(num), 0:10	
1	**assign**	01(global), 1:y	05(string), 0:"hello"	
2	**jump**	00(label), 8		
3	**enterfunc**	08(userfunc), 0:f		
4	**add**	03(local), 0:_t0	02(formal), 0:x	02(formal), 1:y
5	**assign**	10(retval)	03(local), 0:_t0	
6	**jump**	00(label), 7		
7	**exitfunc**	08(userfunc), 0:f		
8	**assign**	01(global), 2:z	05(string), 1:"world"	
9	**pusharg**	04(num), 1:3.1415		

Από ενδιάμεσο σε τελικό κώδικα

- Στον ενδιάμεσου κώδικα έχουμε εντολές που δέχονται *expr** ενώ στον τελικό κώδικα χρειαζόμαστε *vmarg**
- Η μετατροπή των ορισμάτων αυτών γίνεται χρησιμοποιώντας τη συνάρτηση ***make_operand***

```
unsigned consts_newstring (char* s);
unsigned consts_newnumber (double n);
unsigned libfuncs_newused(char* s);

void make_operand (expr* e, vmarg* arg) {
    switch (e->type) {

        /* All those below use a variable for storage
        */
        case var_e :
        case tableitem_e:
        case arithexpr_e:
        case boolexpr_e:
        case newtable_e: {

            arg->val = e->sym->offset;

            switch (e->sym->space) {
                case programvar:    arg->type = global_a;   break;
                case functionlocal: arg->type = local_a;    break;
                case formalarg:     arg->type = formal_a;   break;
                default: assert(0);
            }
        }

        /* Constants */

        case constbool_e: {
            arg->val = e->boolConst;
            arg->type = bool_a;     break;
        }

        case conststring_e: {
            arg->val = consts_newstring(e->strConst);
            arg->type = string_a;   break;
        }
```

struct vmarg { vmarg_t type; unsigned val; }

```
        case costnum_e: {
            arg->val = consts_newnumber(e->numConst);
            arg->type = number_a;   break;
        }

        case nil_e: arg->type = nil_a ; break;

        /* Functions */

        case programfunc_e: {
            arg->type = userfunc_a;
            arg->val = e->sym->taddress;
            break;
        }

        case libraryfunc_e: {
            arg->type = libfunc_a;
            arg->val = libfuncs_newused(e->sym->name);
            break;
        }

        default: assert(0);
    }
}
```

Παραγωγή τελικού κώδικα

- Η παραγωγή τελικού κώδικα γίνεται μέσω της συνάρτησης emit όπως και στον ενδιάμεσο κώδικα (με την διαφορά ότι έχει ένα όρισμα τύπου instruction*)

```
struct instruction {
        vmopcode  opcode;
        vmarg     result;
        vmarg     arg1;
        vmarg     arg2;
        unsigned  srcLine;
};
```

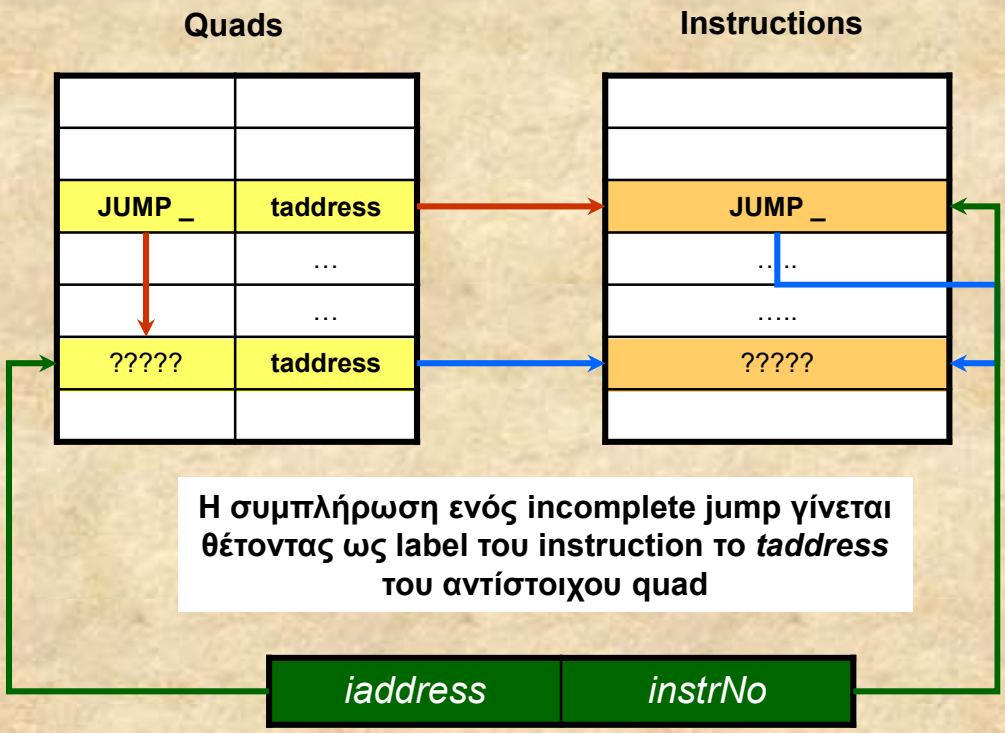
Incomplete jumps (1/2)

- Τα target labels των jump εντολών του ενδιάμεσου κώδικα θα πρέπει να τροποποιηθούν στις αντίστοιχες jump εντολές τελικού κώδικα.
  - Καθώς η αντιστοίχιση εντολών ενδιάμεσου κώδικα τελικού κώδικα δεν είναι 1-1
- Εισάγουμε το πεδίο taddress στις εντολές ενδιάμεσου κώδικα και το συμπληρώνουμε κατά την παραγωγή τελικού κώδικα με τον αριθμό της πρώτης εντολής τελικού κώδικα που αντιστοιχεί σε αυτή

```
struct incomplete_jump {
    unsigned            instrNo;    // The jump instruction number.
    unsigned            iaddress;   // The i-code jump-target address.
    incomplete_jump*    next;       // A trivial linked list.
};

incomplete_jump*    ij_head = (incomplete_jump*) 0;
unsigned            ij_total = 0;

void add_incomplete_jump (unsigned instrNo, unsigned iaddress);
```

```
patch_incomplete_jumps() {
        for each incomplete jump x do {
                if x.iaddress = intermediate code size then
                        instructions[x.instrNo].result = target code size;
                else
                        instructions[x.instrNo].result = quads[x.iaddress].taddress;
        }
}
```

Incomplete jumps (2/2)



Στο *iaddress* αποθηκεύουμε τον αριθμό της εντολής ενδιάμεσου κώδικα προορισμού του jump

Στο *instrNo* αποθηκεύουμε τον αριθμό της εντολής τελικού κώδικα που παράγεται για το jump στον ενδιάμεσο κώδικα

Τελικός κώδικας για Λογικές Εκφράσεις (1/2)

ενδιάμεσος κώδικας

...			
NOT	arg1	result	*taddress*
...			

τελικός κώδικας

...			
jeq	arg1	false	_
assign	false	result	
jump			_
assign	true	result	
...			

+2

+3

ενδιάμεσος κώδικας

...				
OR	arg1	arg2	result	*taddress*
...				

τελικός κώδικας

...			
jeq	arg1	true	_ +4
jeq	arg2	true	+3
assign	false	result	
jump			_
assign	true	result	
...			

+2

ΛΟΓΙΚΕΣ ΕΚΦΡΑΣΕΙΣ

Τελικός κώδικας για Λογικές Εκφράσεις (2/2)

ενδιάμεσος κώδικας

...				
AND	arg1	arg2	result	*taddress*
...				

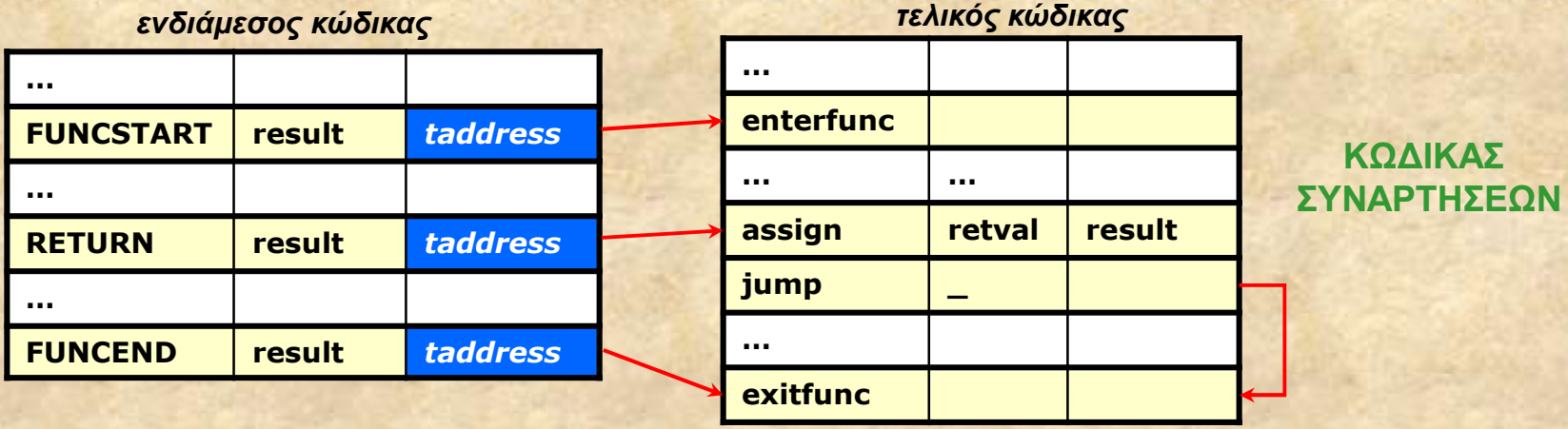
τελικός κώδικας

...			
jeq	arg1	false	_ +4
jeq	arg2	false	+3
assign	true	result	
jump			_
assign	false	result	
...			

+2

ΛΟΓΙΚΕΣ ΕΚΦΡΑΣΕΙΣ

Τελικός κώδικας για Συναρτήσεις



- Επειδή το target label του jump δεν είναι γνωστό πριν παραχθεί κώδικας για το ***funcend***, διατηρείται μία λίστα (*returnList*) για κάθε τέτοιο jump και θα συμπληρώνονται τα target labels αυτών των jump αφού παραχθεί κώδικας για το ***funcend***
- Επειδή όμως υποστηρίζεται ο ορισμός συνάρτησης μέσα σε μία άλλη χρησιμοποιείται μία στοίβα (*funcstack*):
  - Στο ***funcstart*** γίνεται push η συνάρτηση στην στοίβα
  - Στο ***funcend*** γίνεται pop και γίνονται patch τα target labels των εντολών jump που βρίσκονται στην *returnList*.
  - Στο ***return*** εισάγουμε τον αριθμό εντολής τελικού κώδικα του jump στην *returnList* της συνάρτησης που βρίσκεται στην κορυφή της στοίβας

Τεχνική παραγωγής τελικού κώδικα

```
extern void generate_ADD (quad*);
extern void generate_SUB (quad*);
extern void generate_MUL (quad*);
extern void generate_DIV (quad*);
extern void generate_MOD (quad*);
extern void generate_NEWTABLE (quad*);
extern void generate_TABLEGETELM (quad*);
extern void generate_TABLESETELEM (quad*);
extern void generate_ASSIGN (quad*);
extern void generate_NOP (quad*);
extern void generate_JUMP (quad*);
extern void generate_IF_EQ (quad*);
extern void generate_IF_NOTEQ (quad*);
extern void generate_IF_GREATER (quad*);
extern void generate_IF_GREATEREQ (quad*);
extern void generate_IF_LESS (quad*);
extern void generate_IF_LESSEQ (quad*);
extern void generate_NOT (quad*);
extern void generate_OR (quad*);
extern void generate_PARAM (quad*);
extern void generate_CALL (quad*);
extern void generate_GETRETVAL (quad*);
extern void generate_FUNCSTART (quad*);
extern void generate_RETURN (quad*);
extern void generate_FUNCEND (quad*);
```

Παραγωγή τελικού κώδικα με χρήση των generate functions

```
/* Ensure that the order of presence in the array is equal to
   the enumerated constant value of the respective i-code
   instruction.
*/
typedef void (*generator_func_t)(quad*);

generator_func_t generators[] = {
    generate_ADD,
    generate_SUB,
    generate_MUL,
    generate_DIV,
    generate_MOD,
    generate_NEWTABLE,
    generate_TABLEGETELM,
    generate_TABLESETELEM,
    generate_ASSIGN,
    generate_NOP,
    generate_JUMP,
    generate_IF_EQ,
    generate_IF_NOTEQ,
    generate_IF_GREATER,
    generate_IF_GREATEREQ,
    generate_IF_LESS,
    generate_IF_LESSEQ,
    generate_NOT,
    generate_OR,
    generate_PARAM,
    generate_CALL,
    generate_GETRETVAL,
    generate_FUNCSTART,
    generate_RETURN,
    generate_FUNCEND
};

void generate (void) {
    for (unsigned i = 0; i<total; ++i)
        (*generators[quads[i].op])(quads+i);
}
```

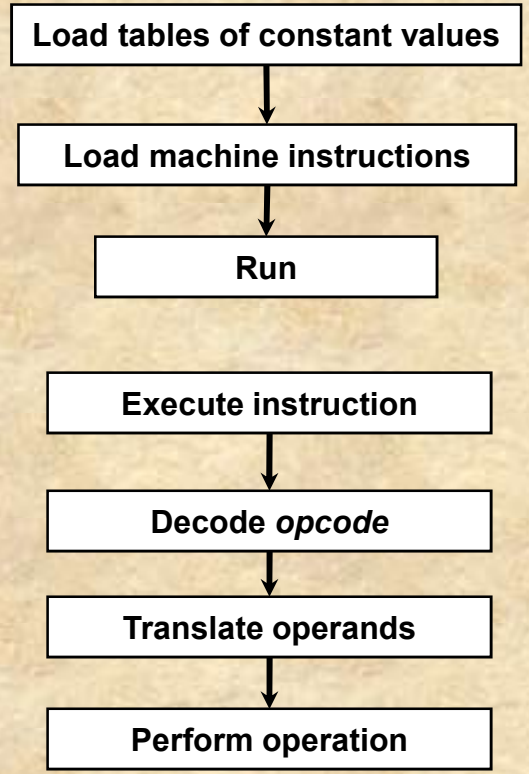
Μέρος 3ο
Εικονική μηχανή

Εικονική Μηχανή

- Φορτώνει και εκτελεί ένα αρχείο τελικού κώδικα

avmbinaryfile → *magicnumber arrays code*
magicnumber → **340200501** #unsigned
arrays → *strings numbers userfunctions libfunctions*
strings → *total (string)**
total → **unsigned**
string → *size* (**char**)* #null terminated
size → **unsigned**
numbers → *total* (**double***)
userfunctions → *total (userfunc)**
userfunc → *address localsize id*
address → **unsigned**
localsize → **unsigned**
id → *string*
libfunctions → *strings*
code → *total (instruction)**
instruction → *opcode operand operand operand*
opcode → **byte**
operand → *type value*
type → **byte**
value → **unsigned**

Γραμματική δομής τελικού κώδικα



Λειτουργία εικονικής μηχανής

Μετατροπή Ορισμάτων (1/2)

- Στις εντολές υπάρχουν αρκετοί διαφορετικοί τύποι ορισμάτων
  - Κελιά μνήμης (μεταβλητές)
  - Σταθερές τιμές χρήστη (π.χ. ακέραιες τιμές ή strings constants)
  - Συναρτήσεις βιβλιοθήκης και χρήστη
- Για να απλοποιήσουμε τη υλοποίηση των εντολών μετατρέπουμε όλα τα ορίσματα σε θέσεις μνήμης χρησιμοποιώντας βοηθητικούς καταχωρητές
  - Χρησιμοποιούνται **μόνο για τιμές** καθώς οι μεταβλητές είναι πάντα σε θέσεις μνήμης στη στοίβα
  - Σε αυτούς θα αποθηκεύουμε πάντα τα ορίσματα arg1 και arg2

```
struct avm_table;
struct avm_memcell {
    avm_memcell_t type;
    union {
        double          numVal;
        char*           strVal;
        unsigned char   boolVal;
        avm_table*      tableVal;
        unsigned        funcVal;
        char*           libfuncVal;
    } data;
};
```

```
enum avm_memcell_t {
    number_m    =0,
    string_m    =1,
    bool_m      =2,
    table_m     =3,
    userfunc_m  =4,
    libfunc_m   =5,
    nil_m       =6,
    undef_m     =7
};
```

Μετατροπή Ορισμάτων (2/2)

```
avm_memcell* avm_translate_operand (vmarg* arg, avm_memcell* reg) {

    switch (arg->type) {

        /* Variables */
        case global_a:   return &stack[AVM_STACKSIZE-1-arg->val];
        case local_a:    return &stack[topsp-arg->val];
        case formal_a:   return &stack[topsp+AVM_STACKENV_SIZE+1+arg->val];

        case retval_a:   return &retval;
```

Μεταβλητές

```
case number_a:  {
    reg->type = number_m;
    reg->data.numVal = consts_getnumber(arg->val);
    return reg;
}

case string_a:  {
    reg->type = string_m;
    reg->data.strVal = consts_getstring(arg->val);
    return reg;
}

case bool_a: {
    reg->type = bool_m;
    reg->data.boolVal = arg->val;
    return reg;
}

case nil_a: reg->type = nil_m; return reg;
```

Constants

```
case userfunc_a: {
    reg->type = userfunc_m;
    reg->data.funcVal = arg->val;
    return reg;
}

case libfunc_a: {
    reg->type = libfunc_m;
    reg->data.libfuncVal = libfuncs_getused(arg->val);
    return reg;
}
```

Functions

Συνάρτηση μετατροπής ορισμάτων vmarg σε avm_memcell**

Execution Cycle

```
execute_func_t executeFuncs[]={
    execute_assign,
    execute_add,
    execute_sub,
    execute_mul,
    execute_div,
    execute_mod,
    execute_uminus,
    execute_and,
    execute_or,
    execute_not,
    execute_jeq,
    execute_jne,
    execute_jle,
    execute_jge,
    execute_jlt,
    execute_jgt,
    execute_call,
    execute_pusharg,
    execute_funcenter,
    execute_funcexit,
    execute_newtable,
    execute_tablegetelem,
    execute_tablesetelem,
    execute_nop
};
```

Υλοποίηση του κύκλου εκτέλεσης εντολών της εικονικής μηχανής χρησιμοποιώντας ένα dispatcher με τις συναρτήσεις execute

```
unsigned char   executionFinished = 0;
unsigned        pc = 0;
unsigned        currLine = 0;
unsigned        codeSize = 0;
instruction*    code = (instruction*) 0;
#define AVM_ENDING_PC   codeSize

void execute_cycle (void) {
    if (executionFinished)
        return;
    else
    if (pc == AVM_ENDING_PC) {
        executionFinished = 1;
        return;
    }
    else {
        assert(pc < AVM_ENDING_PC);
        instruction* instr = code + pc;
        assert(
            instr->opcode >= 0 &&
            instr->opcode <= AVM_MAX_INSTRUCTIONS
        );
        if (instr->srcLine)
            currLine = instr->srcLine;
        unsigned oldPC = pc;
        (*executeFuncs[instr->opcode])(instr);
        if (pc == oldPC)
            ++pc;
    }
}
```

Υλοποίηση εντολών - Συναρτήσεις

Call: Σώσιμο περιβάλλοντος και κλήση

```
void avm_callsaveenvironment (void) {
    avm_push_envvalue(totalActuals);
    avm_push_envvalue(pc+1);
    avm_push_envvalue(top + totalActuals + 2);
    avm_push_envvalue(topsp);
}
```

Funcenter

```
case userfunc_m : {
    pc = func->data.funcVal;
    assert(pc < AVM_ENDING_PC);
    assert(code[pc].opcode == funcenter_v);
    break;
}

case string_m:  avm_calllibfunc(func->data.strVal);      break;
case libfunc_m: avm_calllibfunc(func->data.libfuncVal); break;
```

Funcexit

```
totalActuals = 0;
userfunc* funcInfo = avm_getfuncinfo(pc);
topsp = top;
top = top - funcInfo->localSize;
```

Pusharg

```
avm_assign(&stack[top], arg);
++totalActuals;
avm_dec_top();
```

```
void execute_funcexit (instruction* unused) {
    unsigned oldTop = top;
    top     = avm_get_envvalue(topsp + AVM_SAVEDTOP_OFFSET);
    pc      = avm_get_envvalue(topsp + AVM_SAVEDPC_OFFSET);
    topsp   = avm_get_envvalue(topsp + AVM_SAVEDTOPSP_OFFSET);

    while (oldTop++ <= top) /* Intentionally ignoring first. */
        avm_memcellclear(&stack[oldTop]);
}
```

Υλοποίηση εντολών – Συναρτήσεις Βιβλιοθήκης (1/5)

- Υλοποιημένες σε native κώδικα (C/C++)
- Απαιτούν χειροκίνητη ακολουθία εντολών εισόδου και εξόδου
  - Χρησιμοποιώντας τη συνάρτηση *avm_calllibfunc*
- Αλληλεπιδρούν με τις συναρτήσεις alpha
  - Λαμβάνουν τα ορίσματα τους από τη στοίβα
  - Μπορούν να επιστρέψουν τιμές θέτοντας το καταχωρητή retval
- Γίνονται installed κατά την αρχικοποίηση της εικονικής μηχανής

Υλοποίηση εντολών – Συναρτήσεις Βιβλιοθήκης (2/5)

```
typedef void (*library_func_t)(void);
library_func_t avm_getlibraryfunc (char* id); /* Typical hashing. */

void avm_calllibfunc (char* id) {
    library_func_t f = avm_getlibraryfunc(id);
    if (!f) {
        avm_error("unsupported lib func '%s' called!", id);
        executionFinished = 1;
    }
    else {
        /*  Notice that enter function and exit function
            are called manually!
        */
        topsp = top;     /* Enter function sequence. No stack locals. */
        totalActuals = 0;
        (*f)();          /* Call library function. */
        if (!executionFinished) /* An error may naturally occur inside. */
            execute_funcexit((instruction*) 0); /* Return sequence. */
    }
}
```

CALLING FUNCTIONS

Καθώς οι συναρτήσεις βιβλιοθήκης είναι υλοποιημένες σε C/C++, πρέπει να καλέσουμε χειροκίνητα την ακολουθία εντολών που οφείλει να εκτελέσει μία συνάρτηση όταν αρχίζει να εκτελείται, καθώς και τις εντολές της ακολουθίας εξόδου. Προσέξτε ότι οι συναρτήσεις βιβλιοθήκης δεν περιέχουν τοπικές μεταβλητές στοίβας της εικονικής μηχανής (μπορούν βεβαίως να έχουν τοπικές μεταβλητές του προγράμματος C/C++).

Υλοποίηση εντολών – Συναρτήσεις Βιβλιοθήκης (3/5)

```
unsigned avm_totalactuals (void) {
    return avm_get_envvalue(topsp + AVM_NUMACTUALS_OFFSET);
}

avm_memcell* avm_getactual (unsigned i) {
    assert(i < avm_totalactuals());
    return &stack[topsp + AVM_STACKENV_SIZE + 1 + i];
}

/*  Implementation of the library function 'print'.
    It displays every argument at the console.
*/
void libfunc_print (void) {
    unsigned n = avm_totalactuals();
    for (unsigned i = 0; i < n; ++i) {
        char* s = avm_tostring(avm_getactual(i));
        puts(s);
        free(s);
    }
}

/*  With the following every library function is manually
    added in the VM library function resolution map.
*/
void avm_registerlibfunc (char* id, library_func_t addr);
```

CALLING FUNCTIONS

- Πως υλοποιούμε τις συναρτήσεις βιβλιοθήκης; Ως απλές C/C++ συναρτήσεις οι οποίες λαμβάνουν τα πραγματικά ορίσματα από τη στοίβα της εικονικής μηχανής.
- Το ίδιο ισχύει και στις συναρτήσεις βιβλιοθήκης για γλώσσες γενικού σκοπού και παραγωγή κώδικα για πραγματικές μηχανές.
- Καθώς ο τρόπος χρήσης της στοίβας και των καταχωρητών είναι θέμα του compiler, η υλοποίηση συναρτήσεων βιβλιοθήκης για έναν compiler δεν είναι de facto portable και σε έναν άλλον compiler της ίδιας γλώσσας και για την ίδια μηχανή.

Υλοποίηση εντολών – Συναρτήσεις Βιβλιοθήκης (4/5)

- **Εδώ παρουσιάζουμε και την υλοποίηση της *typeof* συνάρτησης βιβλιοθήκης, η οποία επιτρέπει runtime type identification (ελέγχει ακόμη και undefined variables).**

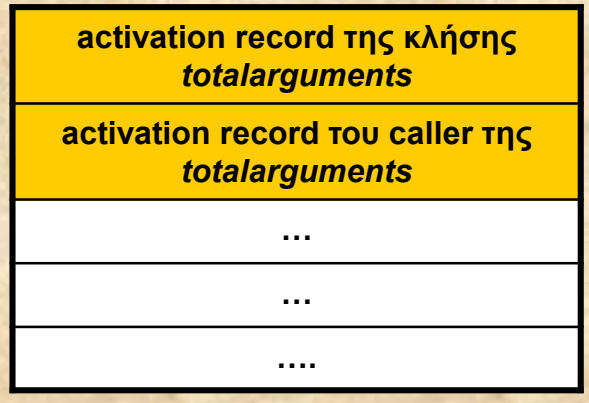
- **Στην περίπτωση που μίας συνάρτηση βιβλιοθήκης χρειάζεται συγκεκριμένο αριθμό από arguments, προφανώς υλοποιεί εσωτερικά και τη λογική έλεγχου και ανάλογα μπορεί να εξάγει ένα *runtime error*.**

- **Επιπλέον φαίνεται ο τρόπος με τον οποίο υλοποιούμε συναρτήσεις βιβλιοθήκης οι οποίες επιστρέφουν τιμές (απλώς θέτουν τον *retval* register).**

```
void libfunc_typeof (void) {

    unsigned n = avm_totalactuals();

    if (n !=1)
        avm_error("one argument (not %d) expected in 'typeof'!", n);
    else {

        /*  Thats how a library function returns a result.
            It has to only set the 'retval' register!
        */
        avm_memcellclear(&retval); /* Don't forget to clean-it up! */
        retval.type = string_m;
        retval.data.strVal = strdup(typeStrings[avm_getactual(0)->type]);
    }
}
```

LIBRARY FUNCTIONS

Υλοποίηση εντολών – Συναρτήσεις Βιβλιοθήκης (5/5)




```
function f() {
    n = totalarguments();
}
```

- Όταν κληθεί η *totalarguments*, στην υλοποίηση της σε C/C++ η κλήση *avm_totalactuals*() επιστρέφει τον αριθμό των arguments στην ίδια την συνάρτηση και όχι στον caller αυτής όπως θα έπρεπε.
- Επομένως για να επιστραφεί ο σωστός αριθμός πρέπει να κινηθούμε ένα activation record κάτω.

```
void libfunc_totalarguments (void) {

    /* Get topsp of previous activation record.
    */
    unsigned p_topsp = avm_get_envvalue(topsp + AVM_SAVEDTOPSP_OFFSET);
    avm_memcellclear(&retval);

    if (!p_topsp) { /* If 0, no previous activation record. */
        avm_error("'totalarguments' called outside a function!");
        retval.type = nil_m;
    }
    else {
        /* Extract the number of actual arguments for the previous
           activation record. */
        retval.type = number_m;
        retval.data.numVal = avm_get_envvalue(p_topsp + AVM_NUMACTUALS_OFFSET);
    }
}
```

Με παρόμοιο τρόπο υλοποιήστε την *argument*(i)

Υλοποίηση αριθμητικών εντολών

```
typedef double (*arithmetic_func_t)(double x, double y);

double add_impl (double x, double y) { return x+y; }
double sub_impl (double x, double y) { return x-y; }
double mul_impl (double x, double y) { return x*y; }
double div_impl (double x, double y) { return x/y; /* Error check? */ }
double mod_impl (double x, double y) {
    return ((unsigned) x) % ((unsigned) y); /* Error check? */
}

/* Dispatcher just for arithmetic functions. */
arithmetic_func_t arithmeticFuncs[] = {
    add_impl,
    sub_impl,
    mul_impl,
    div_impl,
    mod_impl
};
```

```
#define execute_add execute_arithmetic
#define execute_sub execute_arithmetic
#define execute_mul execute_arithmetic
#define execute_div execute_arithmetic
#define execute_mod execute_arithmetic
```

Η υλοποίηση των αριθμητικών εντολών. Μην ξεχάσετε τους ελέγχους για runtime error.

```
void execute_arithmetic (instruction* instr) {

    avm_memcell* lv  = avm_translate_operand(&instr->result, (avm_memcell*) 0);
    avm_memcell* rv1 = avm_translate_operand(&instr->arg1, &ax);
    avm_memcell* rv2 = avm_translate_operand(&instr->arg2, &bx);

    assert(lv && (&stack[0] <= lv && &stack[top] > lv || lv==&retval));
    assert(rv1 && rv2);

    if (rv1->type != number_m || rv2->type != number_m) {
        avm_error("not a number in arithmetic!");
        executionFinished = 1;
    }
    else {
        arithmetic_func_t op = arithmeticFuncs[instr->opcode - add_v];
        avm_memcellclear(lv);
        lv->type             = number_m;
        lv->data.numVal      = (*op)(rv1->data.numVal, rv2->data.numVal);
    }
}
```

ARITHMETIC OPERATIONS

Υλοποίηση συσχετιστικών εντολών διάταξης

- Με τρόπο παρόμοιο των αριθμητικών εκφράσεων υλοποιούνται και οι συσχετιστικοί τελεστές διάταξης < <= > >=, δηλ. οι εντολές JGE, JGT, JLE, JLT, καθώς αφορούν μόνο αριθμούς.
- Προσοχή θέλει το γεγονός ότι δεν χρειάζεται να μετατρέψουμε το operand στο οποίο είναι αποθηκευμένη η διεύθυνση (label) της εντολής προορισμού
- Οι βοηθητικές συναρτήσεις **`comparisonFuncs`** θα είναι έχουν αντίστοιχο signature, δηλ.
  - **`bool (*cmp_func) (double, double)`**

Υλοποίηση συσχετιστικών εντολών ισότητας (1/3)

- Οι εντολές συσχετιστικών τελεστών ισότητας έχουν διαφορετική υλοποίηση ώστε να ικανοποιούνται οι σημασιολογικοί κανόνες της γλώσσας
  - Σύγκριση με undefined προκαλεί *runtime error,*
  - αλλιώς οτιδήποτε είναι συγκρίσιμο == με nil και το αποτέλεσμα είναι true μόνο εάν και τα δύο nil
  - αλλιώς σύγκριση με boolean απαιτεί μετατροπή σε boolean τιμή
  - αλλιώς η σύγκριση επιτρέπεται μόνο μεταξύ ομοειδών (ιδίου τύπου)

Υλοποίηση συσχετιστικών εντολών ισότητας (2/3)

```
typedef unsigned char (*tobool_func_t)(avm_memcell*);

unsigned char number_tobool (avm_memcell* m) { return m->data.numVal != 0; }
unsigned char string_tobool (avm_memcell* m) { return m->data.strVal[0] != 0; }
unsigned char bool_tobool (avm_memcell* m) { return m->data.boolVal; }
unsigned char table_tobool (avm_memcell* m) { return 1; }
unsigned char userfunc_tobool (avm_memcell* m) { return 1; }
unsigned char libfunc_tobool (avm_memcell* m) { return 1; }
unsigned char nil_tobool (avm_memcell* m) { return 0; }
unsigned char undef_tobool (avm_memcell* m) { assert(0); return 0;  }

tobool_func_t toboolFuncs[]={
    number_tobool,
    string_tobool,
    bool_tobool,
    table_tobool,
    userfunc_tobool,
    libfunc_tobool,
    nil_tobool,
    undef_tobool
};

unsigned char avm_tobool (avm_memcell* m) {
    assert(m->type >= 0 && m->type < undef_m);
    return (*toboolFuncs[m->type])(m);
}
```

Η μετατροπή σε boolean θα μας χρειαστεί και καλό είναι να έχουμε ταχύτατη (ως προς την εκτέλεση) υλοποίηση.

EQUALITY OPERATIONS

Υλοποίηση συσχετιστικών εντολών ισότητας (3/3)

```
void execute_jeq (instruction* instr) {

    assert(instr->result.type == label_a);

    avm_memcell* rv1 = avm_translate_operand(&instr->arg1, &ax);
    avm_memcell* rv2 = avm_translate_operand(&instr->arg2, &bx);

    unsigned char result = 0;

    if (rv1->type == undef_m || rv2->type == undef_m)
        avm_error("'undef' involved in equality!");
    else
    if (rv1->type == nil_m || rv2->type == nil_m)
        result =  rv1->type == nil_m && rv2->type == nil_m;
    else
    if (rv1->type == bool_m || rv2->type == bool_m)
        result = (avm_tobool(rv1) == avm_tobool(rv2));
    else
    if (rv1->type != rv2->type)
        avm_error(
            "%s == %s is illegal!",
            typeStrings[rv1->type],
            typeStrings[rv2->type]
        );
    else {
        /* Equality check with dispatching. */
    }

    if(!executionFinished && result)
        pc = instr->result.val;
}
```

EQUALITY OPERATIONS

```
char* typeStrings[]={
    "number",
    "string",
    "bool",
    "table",
    "userfunc",
    "libfunc",
    "nil",
    "undef"
};
```

Θεωρούμε ότι ενσωματώνουμε στην *avm_error* και την εντολή *executionFinished*=1

Εδώ συμπληρώστε την υλοποίηση. Αρκεί να κάνετε dispatching ως προς τον τύπο του *rv1*

Εάν δεν είχαμε runtime error και το αποτέλεσμα είναι *true*, πρέπει να εφαρμοστεί το *jump* (δηλ. να θέσουμε το PC)

Υλοποίηση εντολών πινάκων (1/3)

Οι τιμές των πινάκων γνωρίζουμε ότι πάντα αποθηκεύονται σε κάποια μεταβλητή. Επομένως δεν χρειαζόμαστε βοηθητικό καταχωρητή.

```
void execute_newtable (instruction* instr) {
    avm_memcell* lv  = avm_translate_operand(&instr->result, (avm_memcell*) 0);
    assert(lv && (&stack[0] <= lv && &stack[top] > lv || lv==&retval));

    avm_memcellclear(lv);

    lv->type             = table_m;
    lv->data.tableVal    = avm_tablenew();
    avm_tableincrefcounter(lv->data.tableVal);
}

avm_memcell*    avm_tablegetelem (
                    avm_table*      table,
                    avm_memcell*    index
                );

void            avm_tablesetelem (
                    avm_table*      table,
                    avm_memcell*    index,
                    avm_memcell*    content

                );
```

Δεν ξεχνάμε την αύξηση του μετρητή αναφορών, καθώς κατά τη δημιουργία ο πίνακας έχει τον μετρητή αυτόν στην τιμή 0.

TABLES

Υλοποίηση εντολών πινάκων (2/3)

```
void execute_tablegetelem (instruction* instr) {

    avm_memcell* lv  = avm_translate_operand(&instr->result, (avm_memcell*) 0);
    avm_memcell* t   = avm_translate_operand(&instr->arg1, (avm_memcell*) 0);
    avm_memcell* i   = avm_translate_operand(&instr->arg2, &ax);

    assert(lv && (&stack[0] <= lv && &stack[top] > lv || lv==&retval));
    assert(t && &stack[0] <= t && &stack[top] > t);
    assert(i);

    avm_memcellclear(lv);
    lv->type = nil_m;   /* Default value. */

    if (t->type != table_m) {
        avm_error("illegal use of type %s as table!", typeStrings[t->type]);
    }
    else {
        avm_memcell* content = avm_tablegetelem(t->data.tableVal, i);
        if (content)
            avm_assign(lv, content);
        else {
            char* ts = avm_tostring(t);
            char* is = avm_tostring(i);
            avm_warning("%s[%s] not found!", ts, is);
            free(ts);
            free(is);
        }
    }
}
```

Ενδέχεται το στοιχείο που ζητείται για το συγκεκριμένο κλειδί απλώς να μην υπάρχει ή να μην υποστηρίζεται κλειδί του συγκεκριμένου τύπου.

TABLES

Υλοποίηση εντολών πινάκων (3/3)

- Η υλοποίηση της ***execute_tablesetelem*** είναι απλή, αλλά βασίζεται στην ***avm_tablesetelem*** η οποία είναι κατασκευαστικά πιο απαιτητική
  - Να μην ξεχάσετε ότι με *nil* index αφαιρείται το στοιχείο, δηλ. το *nil* δεν μπορεί να αποθηκεύεται σε πίνακες
  - και ότι πρέπει να χρησιμοποιείτε τις ***avm_assign*** και ***avm_memcellclear***

```
void execute_tablesetelem (instruction* instr) {
    avm_memcell* t    = avm_translate_operand(&instr->result, (avm_memcell*) 0);
    avm_memcell* i    = avm_translate_operand(&instr->arg1, &ax);
    avm_memcell* c    = avm_translate_operand(&instr->arg2, &bx);

    assert(t && &stack[0] <= t && &stack[top] > t);
    assert(i && c);

    if (t->type != table_m)
        avm_error("illegal use of type %s as table!", typeStrings[t->type]);
    else
        avm_tablesetelem(t->data.tableVal, i, c);
}
```

Γιατί δεν ελέγχουμε και για *retval* register?