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

Διάλεξη 13η
ΠΕΡΙΒΑΛΛΟΝ ΕΚΤΕΛΕΣΗΣ – II τελευταία

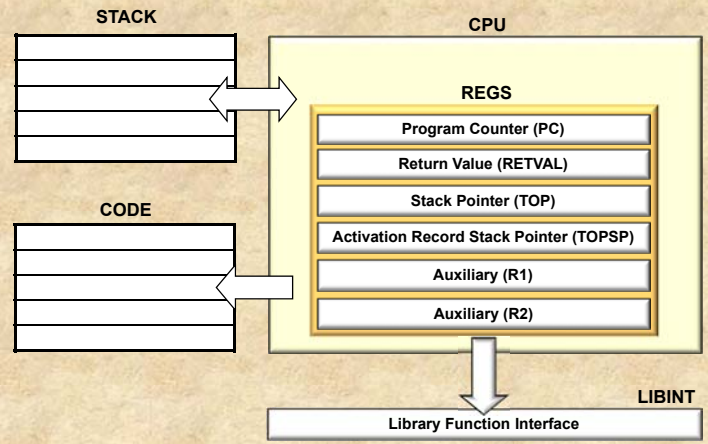
Περιεχόμενα

- *Αρχιτεκτονική της εικονικής μηχανής alpha*
- Ρεπερτόριο εντολών
- Διαχείριση μνήμης
- Δυναμικοί συσχετιστικοί πίνακες

Αρχιτεκτονική της εικονικής μηχανής alpha (1/4)

- Η εικονική μηχανή alpha έχει τα εξής δομικά συστατικά στοιχεία
  - Τμήμα μνήμης κώδικα (*CODE)– read only*
  - Τμήμα στοίβας για τη μνήμη του προγράμματος (*STACK)– read / write*
  - Επεξεργαστής (*CPU)* και ορισμένοι καταχωρητές (*REGS)*
  - Σύνδεση συναρτήσεων βιβλιοθήκης – library interface (*LIBINT)*

Αρχιτεκτονική της εικονικής μηχανής alpha (2/4)



Αρχιτεκτονική της εικονικής μηχανής alpha (3/4)

- Δεν πρέπει να ξεχνάμε ότι αναφερόμαστε σε εικονική μηχανή, με υλοποίηση σε μία γλώσσα προγραμματισμού όπως η C/C++
- Όλες οι διαθέσιμες συναρτήσεις βιβλιοθήκης υλοποιούνται και αυτές στη γλώσσα υλοποίησης της μηχανής, δηλ. στη C/C++
- Οι θέσεις μνήμης της μηχανής είναι τύπου δεδομένων C/C++ που «μας εξυπηρετεί», ώστε να μπορεί να αποθηκεύει τιμές *όλων* των τύπων της γλώσσας alpha
- Όταν αναφερόμαστε σε «μνήμη» της εικονικής μηχανής, στην πράξη σημαίνει μνήμη του προγράμματος C/C++ το οποίο και υλοποιεί (εξομοιώνει) την εικονική μηχανή

Αρχιτεκτονική της εικονικής μηχανής alpha (4/4)

- Όταν αναφερόμαστε σε καταχωρητές της εικονικής μηχανής εννοούμε απλώς κάποιες ειδικές αντίστοιχες μεταβλητές του προγράμματος εξομοιωτή που παίζουν το ρόλο αυτών των καταχωρητών
- Όταν αναφερόμαστε στη CPU της εικονικής μηχανής εννοούμε εκείνο το τμήμα προγράμματος εξομοίωσης το οποίο υλοποιεί πλήρως τη λογική εκτέλεσης των εντολών της εικονικής μηχανής
- Όταν αναφερόμαστε στο τμήμα κώδικα της εικονικής μηχανής εννοούμε τόσο τις δομές C/C++ όσο και την αντίστοιχη δυναμική μνήμη του προγράμματος εξομοίωσης όπου αποθηκεύεται ο φορτωμένος κώδικας μηχανής από κάποιο αρχείο

Περιεχόμενα

- Αρχιτεκτονική της εικονικής μηχανής alpha
- *Ρεπερτόριο εντολών*
- Διαχείριση μνήμης
- Δυναμικοί συσχετιστικοί πίνακες

Ρεπερτόριο εντολών *avm* (1/9)

Το ρεπερτόριο εντολών της μηχανής alpha είναι αρκετά κοντά σε αυτό του ενδιάμεσου κώδικα

Εντολή	Σημασιολογία
assign	Απλή εντολή εκχώρησης σε θέση μνήμης από θέση μνήμης η καταχωρητή.
add	Αριθμητική πράξη / μοναδιαίο μείον και εκχώρηση με ορίσματα θέσεις μνήμης ή σταθερές και προορισμό θέση μνήμης.
sub	
mul	
div	
mod	
uminus	
callfunc	Κλήση συνάρτησης.
enterfunc	Είσοδος σε συνάρτησης.
exitfunc	Έξοδος από συνάρτηση και επιστροφή στην εντολή που έπεται της κλήσης.
and	Λογική πράξη / άρνηση με εκχώρηση σε όρισμα θέση μνήμης ή σταθερά και προορισμό θέση μνήμης.
or	
not	

Οι εντολές λογικών εκφράσεων δεν είναι απαραίτητες καθώς μπορούν να υλοποιηθούν μέσω των υπολοίπων. Το ίδιο και το μοναδιαίο μείον.

Ρεπερτόριο εντολών *avm* (2/9)

jump	Εντολές άλματος.
jeq	
jne	
jgt	
jlt	
jge	
jle	
newtable	Δημιουργία νέου δυναμικού πίνακα και ανάθεση σε προορισμό θέση μνήμης.
tablegetelem	Εξαγωγή στοιχείου πίνακα, με πίνακα να είναι θέση μνήμης, δείκτη να είναι σταθερά ή θέση μνήμης και προορισμό να είναι θέση μνήμης.
tablesetelem	Μεταβολή / δημιουργία / διαγραφή στοιχείου πίνακα, με πίνακα να είναι θέση μνήμης, δείκτη να είναι σταθερά ή θέση μνήμης.
pusharg	Εισαγωγή πραγματικού ορίσματος στη στοίβα.
nop	Απραξία.

Ρεπερτόριο εντολών *avm* (3/9)

Οι εντολές της μηχανής alpha είναι της παρακάτω μορφής ,με κάθε τμήμα να είναι ένας ακέραιος αριθμός. Δηλ. κάθε εντολή καταλαμβάνει χώρο τεσσάρων ακριβώς ακέραιων (ο ο κωδικός μπορεί εναλλακτικά να αποθηκευτεί σε ένα μόνο byte).

Κωδικός	Αποτέλεσμα	Όρισμα 1	Όρισμα 2

Η ερμηνεία του κάθε ορίσματος δίνεται παρακάτω. Όπως φαίνεται, ανεξαρτήτως τύπου ορίσματος, καταφέρνουμε να αποθηκεύσουμε όλη την πληροφορία τιμής σε έναν ακέραιο. Ειδική μέριμνα απαιτείται για τύπους των οποίων οι τιμές που απαιτούν μεγαλύτερο χώρο.

Κωδικός ορίσματος	Ερμηνεία περιεχομένου ορίσματος	Χρήση
00	Ετικέτα εντολής	Εντολές άλματος
01	Καθολική μεταβλητή, *offset*	Θέση μνήμης
02	Τυπικό όρισμα, *offset*	Θέση μνήμης
03	Τοπική μεταβλητή, *offset*	Θέση μνήμης
04	Σταθερά number, *index*	Τιμή
05	Σταθερά string, *index*	Τιμή
06	Σταθερά boolean, τιμές 0 ή 1	Τιμή
07	Σταθερά nil	Τιμή
08	Συνάρτηση χρήστη, *index*	Τιμή
09	Συνάρτηση βιβλιοθήκης, *index*	Τιμή
10	Καταχωρητής αποτελέσματος	Τιμή

Ρεπερτόριο εντολών *avm* (4/9)

- Ορισμένα ορίσματα φαίνεται να μην μπορούν να αποθηκευτούν σε έναν ακέραιο
  - Αριθμοί, καθώς χρησιμοποιούμε double precision
  - Strings, αφού πρόκειται για δυναμικούς πίνακες χαρακτήρων που μπορεί να ξεπεράσουν τα 4 bytes
  - Συναρτήσεις βιβλιοθήκης, καθώς φέρουν ονόματα ως strings
    - Γιατί δεν μπορούμε να χρησιμοποιήσουμε διευθύνσεις συναρτήσεων
- *Η λύση είναι η εξής*
  - Κατά την παραγωγή κώδικα δημιουργούμε έναν αντίστοιχο πίνακα με όλες τις σταθερές τιμές που εμφανίζονται στις εντολές
  - Αντικαθιστούμε την εμφάνιση μίας τέτοιας τιμής σε κάποια εντολή από την αντίστοιχη ακέραια θέσης της στον πίνακα αυτό
  - οι πίνακες αυτοί αποτελούν αναπόσπαστο τμήμα του κώδικα του προγράμματος και λέγονται «*πίνακες σταθερών τιμών*»

Repertório εντολών *avm* (5/9)

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

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.

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"

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		

Repertório εντολών *avm* (6/9)

•Για τα ορίσματα που είναι θέσεις μνήμης, καθώς υπάρχει η πληροφορία τόσο για τον τύπο τους (καθολικές, τοπικές, τυπικές) όσο και για το offset, η θέση μνήμης προσδιορίζεται από την συνάρτηση περιβάλλοντος που έχουμε δώσει στην προηγούμενη διάλεξη.

•Κάτι που επίσης παρατηρείται είναι ότι από τους καταχωρητές ο μόνος που εμφανίζεται στον τελικό κώδικα είναι αυτός για την αποθήκευση επιστρεφόμενου αποτελέσματος.

10	**callfunc**	09(libfunc), 0:cos		
11	**assign**	01(global), 4:_t0	10(retval)	
12	**assign**	01(global), 3:w	01(global), 4:_t0	
13	**pusharg**	01(global), 3:w		
14	**pusharg**	01(global), 2:z		
15	**pusharg**	01(global), 1:y		
16	**pusharg**	01(global), 0:x		
17	**callfunc**	09(libfunc), 1:print		
18	**assign**	01(global), 5:_t0	10(retval)	

Repertório εντολών *avm* (7/9)

Η δομή του binary file που περιέχει κώδικα της εικονικής μηχανής alpha σε context free grammar. Τα αρχεία κώδικα μηχανής alpha έχουν την κατάληξη *.abc* (alpha binary code)

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**

•Η ίδια συντακτική δομή μπορεί να χρησιμοποιηθεί και για την αποθήκευση σε μορφή κειμένου.

•Η ανάγνωση τόσο ενός binary όσο και ενός text file με τελικό κώδικα μπορεί να να υλοποιηθεί καλύτερα με τη χρήση ενός RDP.

•Στην προκειμένη περίπτωση, δεν είναι ακριβώς context free η γραμματική, καθώς σε όλες τις περιπτώσεις που εμφανίζεται ένα *total* μη τερματικό, θα πρέπει να ακολουθούν ακριβώς τόσα αντίστοιχα στοιχεία. Αυτό όμως υλοποιείται πολύ εύκολα με ανακύκλωση μέσα στην αντίστοιχη συνάρτηση ανάλυσης του RDP (context sensitive).

•Η αντίστοιχη υλοποίηση με S/R parser απαιτεί σημασιολογικούς κανόνες και προσωρινή αποθήκευση των *total* (σχετικά πιο πολύπλοκο από RDP).

Repertório εντολών *avm* (8/9)

```
avmbinaryfile() {
        return magicnumber() and arrays() and code() ;
}
magicnumber() {
        return match(UNSIGNED) and currtoken.intVal == MAGICNUMBER;
}
arrays() {
        return strings() and numbers() and userfunctions() and libfunctions();
}
strings() {
        unsigned n;
        if !match(UNSIGNED) then
                return false;
        else
        for n = currtoken.intVal; n; --n do
                string();
}
```

•Για καλύτερο έλεγχο της υλοποίησης παράγουμε κώδικα τόσο σε δυαδική όσο και σε μορφή κειμένου και υποστηρίζουμε φόρτωμα τελικού κώδικα και από τις δύο μορφές.
•Προτεραιότητα δίνουμε στον κώδικα σε μορφή κειμένου καθώς είναι εύκολο να ελεγχθεί η ορθότητά του με το μάτι.
•Μπορείτε να χρησιμοποιήσετε ειδικά keywords για τον τύπο των ορισμάτων ώστε να είναι πιο ευανάγνωστος ο τελικός κώδικας.

Ρεπερτόριο εντολών *avm* (9/9)

Για αποφυγή σύγκρουσης ονομάτων με τις εντολές ενδιάμεσου κώδικα προφανώς θα πρέπει να βάλετε κάποιο επίθεμα (εδώ έχουμε _v)

```
enum vmopcode {
    assign_v,       add_v,          sub_v,
    mul_v,          div_v,          mod_v,
    uminus_v,       and_v,          or_v,
    not_v,          jeq_v,          jne_v,
    jle_v,          jge_v,          jlt_v,
    jgt_v,          call_v,         pusharg_v,
    funcenter_v,    funcexit_v,     newtable_v,
    tablegetelem_v, tablesetelem_v, nop_v,
};

enum vmarg_t {
    label_a     =0,
    global_a    =1,
    formal_a    =2,
    local_a     =3,
    number_a    =4,
    string_a    =5,
    bool_a      =6,
    nil_a       =7,
    userfunc_a  =8,
    libfunc_a   =9,
    retval_a    =10
};
```

```
struct vmarg {
    vmarg_t     type;
    unsigned    val;
};

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

struct userfunc {
    unsigned    address;
    unsigned    localSize;
    char*       id;
};

double*     numConsts;
unsigned    totalNumConsts;
char**      stringConsts;
unsigned    totalStringConsts;
char**      namedLibfuncs;
unsigned    totalNamedLibfuncs;
userfunc*   userFuncs;
unsigned    totalUserFuncs;
```

Οι πίνακες των σταθερών ορισμάτων δημιουργούνται κατά το loading του κώδικα

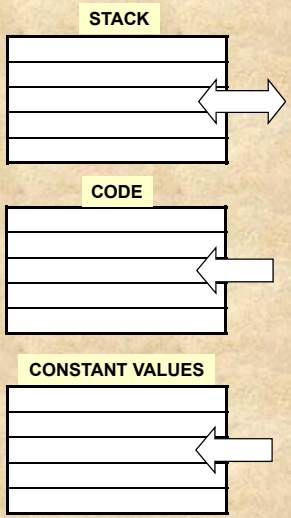
Περιεχόμενα

- Αρχιτεκτονική της εικονικής μηχανής alpha
- Ρεπερτόριο εντολών
- *Διαχείριση μνήμης*
- Δυναμικοί συσχετιστικοί πίνακες

Διαχείριση μνήμης (1/3)

- Η μηχανή alpha έχει τα εξής τμήματα μνήμης
  - Τμήμα κώδικα (που είναι δυναμικός πίνακας) και βοηθητικοί πίνακες σταθερών τιμών
  - Τμήμα στοίβας για καθολικές μεταβλητές και καταχωρήσεις ενεργών κλήσεων (activation records)
- Το μόνο τμήμα στο οποίο αποθηκεύονται δυναμικά τιμές των τύπων της γλώσσας alpha είναι η στοίβα
  - Σε κάθε κελί της στοίβας πρέπει να μπορεί να αποθηκεύεται τιμή οποιουδήποτε alpha τύπου
  - Σε μία υλοποίηση της στοίβας στη C/C++ κάτι τέτοιο είναι εφικτό πολύ εύκολα, ορίζοντας ως τύπο δεδομένων της στοίβας έναν ενοποιημένο τύπο – *union*
    - Η ανάγκη αυτή είναι παρόμοια με την περίπτωση της στοίβας ενός S/R parser, όπου έπρεπε να αποθηκεύονται τιμές όλων των διαφορετικών τύπων των γνωρισμάτων για τα γραμματικά σύμβολα

Διαχείριση μνήμης (2/3)



•Η στοίβα είναι η μνήμη δεδομένων ενώ το μέγεθός της είναι γενικά προβλέψιμο. Σε περίπτωση που το πρόγραμμα κατά την εκτέλεση εξαντλήσει όλο το χώρο της στοίβας, έχουμε το λεγόμενο *stack overflow*.

•Υπάρχει η δυνατότητα δυναμικής αύξησης του χώρου της στοίβας, κάτι που απαιτεί προσωρινή παύση της εκτέλεσης, ωστόσο αυτό δεν θα μας απασχολήσει.

•Θα θεωρήσουμε ένα σταθερό αρχικό μέγεθος στοίβας 4096 κελιών – θέσεων μνήμης.

•Το τμήμα κώδικα καθώς και οι πίνακες σταθερών τιμών δεν είναι απλώς read only αλλά το μέγεθος τους καθορίζεται κατά τη διάρκεια φόρτωσης του κώδικα στην μηχανή *avm*.

•Μένει απλώς να ορίσουμε τον ενοποιημένο τύπο δεδομένων της στοίβας ο οποίος να καλύπτει όλους τους τύπους τιμών της γλώσσας alpha.

Διαχείριση μνήμης (3/3)

```
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
};
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;
};

#define AVM_STACKSIZE   4096
#define AVM_WIPEOUT(m)  memset(&(m), 0, sizeof(m))

avm_memcell stack[AVM_STACKSIZE];
static void avm_initstack (void) {
    for (unsigned i=0; i<AVM_STACKSIZE; ++i)
        { AVM_WIPEOUT(stack[i]); stack[i].type = undef_m; }
}
```

- Η τιμή κάθε κελιού περιέχει όλη την πληροφορία για το περιεχόμενο της, δηλ. δεν έχουμε αναφορές σε πίνακες σταθερών τιμών.
- Έτσι, η τιμή string είναι πάντα ένα δυναμικό copy είτε αναπαριστά σταθερά τιμή είτε όχι. Οι σταθερές τιμές εμφανίζονται με έμμεσες αναφορές μόνο στον κώδικα και ποτέ στη στοίβα.
- Η τιμή μίας συνάρτησης χρήστη είναι απευθείας η αντίστοιχη διεύθυνση κώδικα ενώ μίας συνάρτησης βιβλιοθήκης το μοναδικό όνομά της (αντίγραφο).
- Η στοίβα μπορεί να υλοποιηθεί είτε ως στατικός πίνακας σταθερού μεγέθους η ως δυναμικός πίνακας σε περίπτωση δυναμικής αύξησης.
- Λύσεις όπως υλοποίηση της στοίβας με δείκτες συνδεσμολογίας (π.χ. next και previous) δεν ενδείκνυνται καθώς δεν είναι κατάλληλες για τυχαία πρόσβαση στη στοίβα (π.χ. stack[i]).

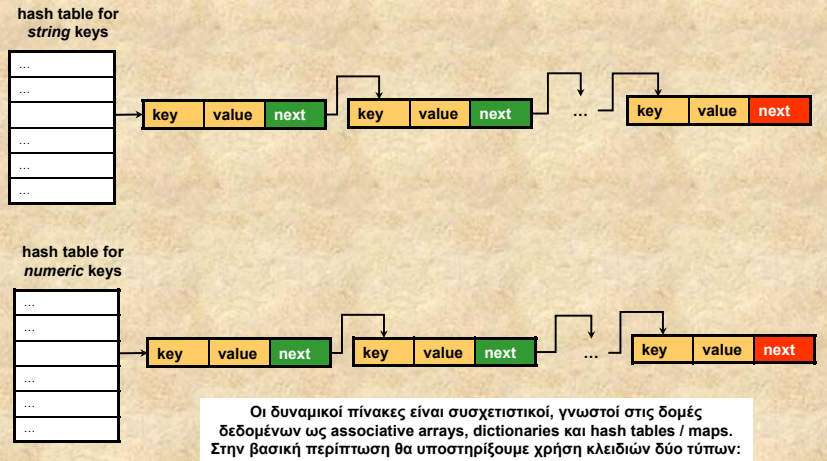
Περιεχόμενα

- Αρχιτεκτονική της εικονικής μηχανής alpha
- Ρεπερτόριο εντολών
- Διαχείριση μνήμης
- *Δυναμικοί συσχετιστικοί πίνακες*

Δυναμικοί συσχετιστικοί πίνακες (1/13)

- Οι δυναμικοί συσχετιστικοί πίνακες αποτελούν τον βασικότερο τύπο δεδομένων της γλώσσας alpha
- Όπως έχουμε ήδη αναφέρει υποστηρίζουν αυτόματη απελευθέρωση μνήμης όταν «δεν χρησιμοποιούνται πλέον» από το πρόγραμμα - *automatic garbage collection*
- Η τακτική για garbage collection που θα υλοποιήσουμε είναι μία απλής μορφής, αλλά αρκετά ικανοποιητική και βασίζεται σε καταμέτρηση αναφορών - *reference counting*
  - Η μόνη προϋπόθεση για να λειτουργήσει είναι η έλλειψη κυκλικών αναφορών - θα δούμε ακριβώς τι σημαίνει αυτό
- Πρώτα θα αναφέρουμε τα κύρια τμήματα της υλοποίησης των δυναμικών πινάκων και έπειτα θα αναφερθούμε στην μέθοδο για garbage collection

Δυναμικοί συσχετιστικοί πίνακες (2/13)



Δυναμικοί συσχετιστικοί πίνακες (3/13)

Οι τρεις από τις πρώτες συναρτήσεις είναι αυτές που θα χρησιμοποιηθούν στην υλοποίηση των αντίστοιχων εντολών της εικονικής μηχανής. Συνήθως η παράμετρος και το επιστρεφόμενο αποτέλεσμα είναι τύπου «κελί μνήμης».

```
avm_table*      avm_tablenew (void);
void            avm_tabledestroy (avm_table* t);
avm_memcell*    avm_tablegetelem (avm_memcell* key);
void            avm_tablesetelem (avm_memcell* key, avm_memcell* value);

#define AVM_TABLE_HASHSIZE  211

struct avm_table_bucket {
    avm_memcell         key;
    avm_memcell         value;
    avm_table_bucket*   next;
};

/*  For simplicity, we only show support for numeric and
    string keys. Bonus for teams  imlpementing keys
    for user functions, library functions and booleans.
*/
struct avm_table {
    unsigned            refCounter;
    avm_table_bucket*   strIndexed[AVM_TABLE_HASHSIZE];
    avm_table_bucket*   numIndexed[AVM_TABLE_HASHSIZE];
    unsigned            total;
};
```

Σε κάθε στοιχείο του πίνακα αποθηκεύεται τόσο το «κλειδί» όσο και η αντίστοιχη «τιμή» με τύπο ίδιο με τα κελιά της στοίβας. Φαίνεται λοιπόν ξεκάθαρα ότι οι <u>δυναμικοί πίνακες συνιστούν δυναμική μνήμη στη γλώσσα *alpha*</u>.

Οι δυναμικοί *alpha* πίνακες ενσωματώνουν τόσους διαφορετικούς hash tables όσοι και οι διαφορετικοί τύποι κλειδιών που υποστηρίζουν.

Αντί της χρήσης ενός total μετρητή των συνολικών στοιχείων μπορούν να χρησιμοποιηθούν ξεχωριστές μεταβλητές total για τους επιμέρους hash tables.

Δυναμικοί συσχετιστικοί πίνακες (4/13)

```
void    avm_tableincrefcounter (avm_table* t)
    { ++t->refCounter; }

/*  Automatic garbage collection for tables when
    reference counter gets zero.
*/
void    avm_tabledecrefcounter (avm_table* t) {
    assert(t->refCounter > 0);
    if (!--t->refCounter)
        avm_tabledestroy(t);
}

void avm_tablebucketsinit (avm_table_bucket** p) {
    for (unsigned i=0; i<AVM_TABLE_HASHSIZE; ++i)
        p[i] = (avm_table_bucket*) 0;
}
```

•Η αύξηση του reference counter γίνεται κάθε φορά που ένα κελί μνήμης (είτε στη στοίβα, είτε σε άλλο δυναμικό πίνακα) εκχωρείται την διεύθυνση ενός πίνακα.

•Η μείωση γίνεται αντίστοιχα όταν ένα κελί μνήμης το οποίο είχε τιμή τύπου *table_m* (δηλ. διεύθυνση κάποιου δυναμικού πίνακα) είτε αλλάζει τιμή ή καταστρέφεται.

•Η μείωση μπορεί να καταστρέψει τον πίνακα εάν το reference counter γίνει 0, δηλ. όταν δεν υπάρχει άλλο κελί μνήμης που να αναφέρεται σε αυτόν τον πίνακα.

```
/* The reference counter is initially zero.
*/
avm_table* avm_tablenew (void) {
    avm_table* t = (avm_table*) malloc(sizeof(avm_table));
    AVM_WIPEOUT(*t);

    t->refCounter = t->total =  0;
    avm_tablebucketsinit(t->numIndexed);
    avm_tablebucketsinit(t->strIndexed);

    return t;
}
```

•Κατά τη δημιουργία ενός δυναμικού πίνακα ο reference counter είναι αρχικά πάντα 0.

Δυναμικοί συσχετιστικοί πίνακες (5/13)

```
/* When a cell is cleared, it has to destroy all
   dynamic data content or reset its reference to a table.
*/
void avm_memcellclear (avm_memcell* m);

void avm_tablebucketsdestroy (avm_table_bucket** p) {
    for (unsigned i=0; i<AVM_TABLE_HASHSIZE; ++i, ++p) {
        for (avm_table_bucket* b = *p; b;) {
            avm_table_bucket* del = b;
            b = b->next;
            avm_memcellclear(&del->key);
            avm_memcellclear(&del->value);
            free(del);
        }
        p[i] = (avm_table_bucket*) 0;
    }
}

void avm_tabledestroy (avm_table* t) {
    avm_tablebucketsdestroy(t->strIndexed);
    avm_tablebucketsdestroy(t->numIndexed);
    free(t);
}
```

•Η καταστροφή ενός πίνακα πίνακα σημαίνει περισσότερα από απλή απελευθέρωση της καταλαμβανόμενης μνήμης.

•Πρέπει να γίνει «αποκομιδή» όλων των κελιών μνήμης τα οποία έχουν δημιουργηθεί και αποθηκεύονται στις δομές του πίνακα.

•Η διαδικασία αυτή «καθαρισμού» κελιών μνήμης υποδεικνύεται με τη συνάρτηση *avm_memcellclear* της οποίας η υλοποίηση θα δοθεί αργότερα στην κατασκευή της εικονικής μηχανής. Ο καθαρισμός απαιτείται τόσο για το κλειδί όσο και για την τιμή κάθε στοιχείου του πίνακα.

•Γενικά η συνάρτηση αυτή πρέπει να απελευθερώσει είτε τα όποια δυναμικά δεδομένα ενός κελιού (π.χ. για strings) ή σε περίπτωση που η τιμή του κελιού είναι πίνακας, να μειώσει τον μετρητή αναφορών του πίνακα (αυτό ενδέχεται να προκαλέσει και καταστροφή του πίνακα).

Δυναμικοί συσχετιστικοί πίνακες (6/13)

- Η τακτική αυτή για garbage collection είναι πολύ απλή στην υλοποίηση, ωστόσο δεν καλύπτει την περίπτωση κυκλικών αναφορών
  - Θεωρούμε ότι όταν μία μεταβλητή είναι τύπου πίνακα προκύπτει μία ακμή από την μεταβλητή στον πίνακα
  - Κάθε κύκλος στον γράφο που ορίζεται από αυτές τις ακμές είναι περίπτωση κυκλικών αναφορών
- Υπάρχουν περιπτώσεις στις οποίες οι κυκλικές αναφορές είναι αναπόφευκτες - σε αυτές θα πρέπει ο προγραμματιστής να φροντίζει «χειροκίνητα» την διάσπαση του κύκλου κάνοντας nil κάποια αναφορά
- Εναλλακτικά μπορούν να χρησιμοποιηθούν άλλοι αλγόριθμοι
  - http://en.wikipedia.org/wiki/Automatic_garbage_collection
  - http://www3.interscience.wiley.com/cgi-bin/fulltext/108566782/PDFSTART
    - reference counting με εξάλειψη κύκλων
- Στην υλοποίηση της εικονικής μηχανής για τη γλώσσα alpha θα υλοποιήσουμε την απλή περίπτωση
  - η επέκταση ή αντικατάσταση με πιο προηγμένη μέθοδο δεν είναι απλό επιχείρημα

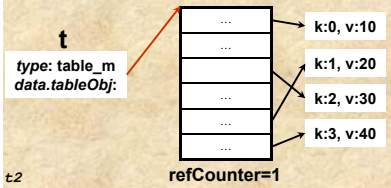
Δυναμικοί συσχετιστικοί πίνακες (7/13)

```
t  =  [10, 20, 30, t = 40];

// Ισοδύναμο με το παρακάτω τμήμα.
t=40;
t=[];
t[0]=10;
t[1]=20;
t[2]=30;
t[3]=40;

t2=[{"t" : t}];
t=nil;

t2.t2 = t2;      // Κύκλος αναφορών για το t2
t2.t = nil;      // Το t γίνεται collected
t2 = nil;        // Αλλά το t2 δεν μπορεί
```



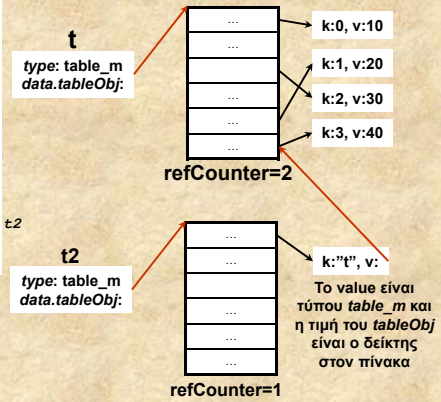
Δυναμικοί συσχετιστικοί πίνακες (8/13)

```
t  =  [10, 20, 30, t = 40];

// Ισοδύναμο με το παρακάτω τμήμα.
t=40;
t=[];
t[0]=10;
t[1]=20;
t[2]=30;
t[3]=40;

t2=[{"t" : t}];
t=nil;

t2.t2 = t2;      // Κύκλος αναφορών για το t2
t2.t = nil;      // Το t γίνεται collected
t2 = nil;        // Αλλά το t2 δεν μπορεί
```



To value είναι τύπου *table_m* και η τιμή του *tableObj* είναι ο δείκτης στον πίνακα

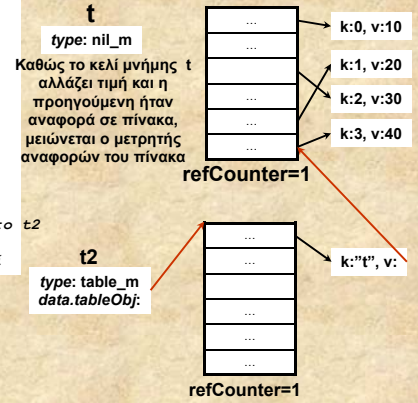
Δυναμικοί συσχετιστικοί πίνακες (9/13)

```
t  =  [10, 20, 30, t = 40];

// Ισοδύναμο με το παρακάτω τμήμα.
t=40;
t=[];
t[0]=10;
t[1]=20;
t[2]=30;
t[3]=40;

t2=[{"t" : t}];
t=nil;

t2.t2 = t2;      // Κύκλος αναφορών για το t2
t2.t = nil;      // Το t γίνεται collected
t2 = nil;        // Αλλά το t2 δεν μπορεί
```



Καθώς το κελί μνήμης t αλλάζει τιμή και η προηγούμενη ήταν αναφορά σε πίνακα, μειώνεται ο μετρητής αναφορών του πίνακα

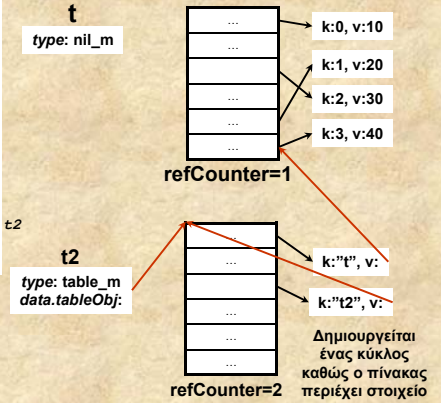
Δυναμικοί συσχετιστικοί πίνακες (10/13)

```
t  =  [10, 20, 30, t = 40];

// Ισοδύναμο με το παρακάτω τμήμα.
t=40;
t=[];
t[0]=10;
t[1]=20;
t[2]=30;
t[3]=40;

t2=[{"t" : t}];
t=nil;

t2.t2 = t2;      // Κύκλος αναφορών για το t2
t2.t = nil;      // Το t γίνεται collected
t2 = nil;        // Αλλά το t2 δεν μπορεί
```



Δημιουργείται ένας κύκλος καθώς ο πίνακας περιέχει στοιχείο που αναφέρεται σε αυτόν.

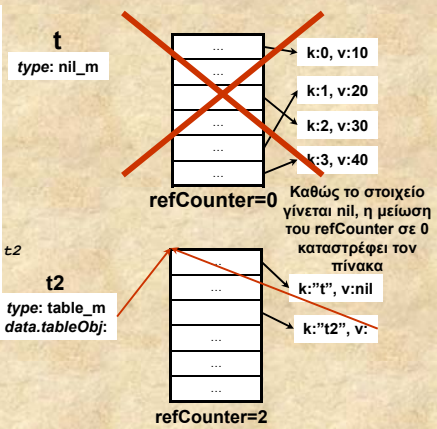
Δυναμικοί συσχετιστικοί πίνακες (11/13)

```
t  =  [10, 20, 30, t = 40];

// Ισοδύναμο με το παρακάτω τμήμα.
t=40;
t=[];
t[0]=10;
t[1]=20;
t[2]=30;
t[3]=40;

t2=[{"t" : t}];
t=nil;

t2.t2 = t2;      // Κύκλος αναφορών για το t2
t2.t = nil;      // Το t γίνεται collected
t2 = nil;        // Αλλά το t2 δεν μπορεί
```



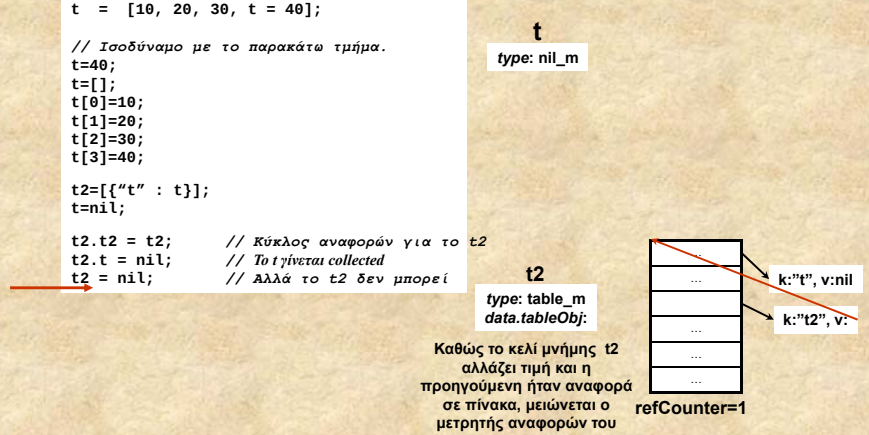
Δυναμικοί συσχετιστικοί πίνακες (12/13)

```
t  =  [10, 20, 30, t = 40];

// Ισοδύναμο με το παρακάτω τμήμα.
t=40;
t=[];
t[0]=10;
t[1]=20;
t[2]=30;
t[3]=40;

t2=[{"t" : t}];
t=nil;

t2.t2 = t2;      // Κύκλος αναφορών για το t2
t2.t = nil;      // Το t γίνεται collected
t2 = nil;        // Αλλά το t2 δεν μπορεί
```



Καθώς το κελί μνήμης t2 αλλάζει τιμή και η προηγούμενη ήταν αναφορά σε πίνακα, μειώνεται ο μετρητής αναφορών του πίνακα

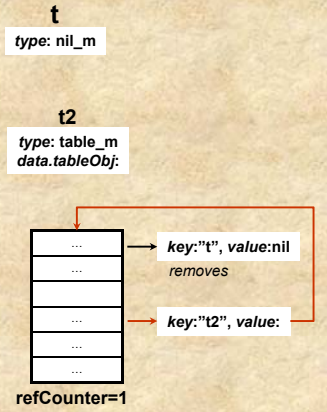
Δυναμικοί συσχετιστικοί πίνακες (13/13)

```
t  =  [10, 20, 30, t = 40];

// Ισοδύναμο με το παρακάτω τμήμα.
t=40;
t=[];
t[0]=10;
t[1]=20;
t[2]=30;
t[3]=40;

t2=[{"t" : t}];
t=nil;

t2.t2 = t2;      // Κύκλος αναφορών για το t2
t2.t = nil;      // Το t γίνεται collected
t2 = nil;        // Αλλά το t2 δεν μπορεί
```



Ενώ έχει τελειώσει η εκτέλεση του κώδικα, υπάρχει κάποιο memory leak καθώς ο διπλανός πίνακας λόγω της κυκλικής αναφοράς έχει *refCounter > 0*. Είναι memory leak γιατί δεν υπάρχει καμία διαθέσιμη μεταβλητή του προγράμματος με την οποία να μπορεί να χρησιμοποιηθεί ο πίνακας.