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<
```

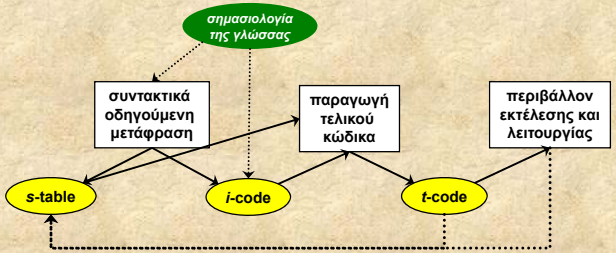
ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης

Διάλεξη 9η
ΠΑΡΑΓΩΓΗ ΕΝΔΙΑΜΕΣΟΥ ΚΩΔΙΚΑ - Ι

Περιεχόμενα

Ρόλος και εξαρτήσεις (1/3)

- Γενικά, ο ενδιάμεσος κώδικας είναι μία εσωτερική αναπαράσταση του αρχικού πηγαίου κώδικα αρκετά κοντά στον τελικό κώδικα, χωρίς ωστόσο να εξαρτάται από αυτόν
- Η δομή του ενδιάμεσου κώδικα και ο τρόπος παραγωγής εξαρτάται πολύ από τα χαρακτηριστικά και τη σημασιολογία της γλώσσας
- Τμήμα της πληροφορίας που κρατείται στον πίνακα συμβόλων επηρεάζεται από τον τελικό κώδικα και τα χαρακτηριστικά του περιβάλλοντος εκτέλεσης του τελικού κώδικα



Ρόλος και εξαρτήσεις (2/3)

- Απαιτείται πλήρης γνώση της σημασιολογίας της γλώσσας (δηλ. τι ακριβώς σημαίνει λειτουργικά κάθε γλωσσική δομή)
- Θα παρουσιάσουμε μία τυπολογία ενδιάμεσου κώδικα, η οποία θα περιέχει και ειδικές εντολές με σκοπό
  - τη διευκόλυνση παραγωγής τελικού κώδικα για τη μηχανή *avm* (alpha virtual machine)
- Ταυτόχρονα θα δούμε πως μπορούμε να αποφύγουμε τις ειδικές εντολές κάνοντας ελαφρώς «εξυπνότερο» τον παραγωγό τελικού κώδικα
- Η μέθοδος παραγωγής ενδιάμεσου κώδικα που θα μελετήσουμε και θα εφαρμόσουμε υιοθετεί την κατασκευή μεταφραστή δύο περασμάτων
  - Στην πρώτη φάση παράγουμε τον ενδιάμεσο κώδικα για το πρόγραμμα εισόδου
  - Στη δεύτερη ανεξάρτητη φάση επιτελείται η παραγωγή του τελικού κώδικα

Ρόλος και εξαρτήσεις (3/3)

Σημασιολογία της γλώσσας *alpha*
Τυπολογία ενδιάμεσου κώδικα
- Με ή χωρίς ειδικές εντολές για το *avm*
- Τρόπος υλοποίησης

Επιπλέον πληροφορία στον πίνακα συμβόλων
Συντακτικά οδηγούμενη μετάφραση
- Δηλώσεις
  - Μεταβλητές
  - Συναρτήσεις
- Εκφράσεις
  - Αποθήκευσης (l-values)
  - Βασικές (primary)
    - Κλήση συνάρτησης
  - Αριθμητικές
  - Λογικές
    - Πλήρης αποτίμηση
    - Γρήγορη μερική αποτίμηση
  - Δημιουργίας πινάκων
  - Διεύθυνση συνάρτησης
- Εντολές
  - Εκχώρησης
  - Διακλάδωσης
  - Ανακύκλωσης
  - Ειδικές εντολές

•Θα ασχοληθούμε κυρίως με τη συντακτικά οδηγούμενη παραγωγή ενδιάμεσου κώδικα βασισμένη σε συντιθέμενα γνωρίσματα, επιτρέποντας τους σημασιολογικούς κανόνες να προκαλούν πλάγια αποτελέσματα.

•Σε ορισμένες περιπτώσεις για να έχουμε πληροφορία εξαρτημένη από τα συμφραζόμενα (όπως, π.χ., η παρούσα συνάρτηση), θα χρησιμοποιήσουμε κληρονομημένα γνωρίσματα τα οποία θα κρατούνται σε βοηθητικές μεταβλητές (απλός και γενικός τρόπος, παρά να μπλέκουμε με διαχείριση κληρονομημένων γνωρισμάτων χωρίς πλάγια αποτελέσματα).

•Τα μεταφραστικά σχήματα που θα παρουσιάσουμε βασίζονται στην αυθεντική γραμματική της γλώσσας *alpha* και είναι κατάλληλα για την κατασκευή LR / LL parser. Στην δεύτερη περίπτωση πρέπει οι κανόνες να ενσωματωθούν ως γραμματικά σύμβολα πριν τους LL μετασχηματισμούς.

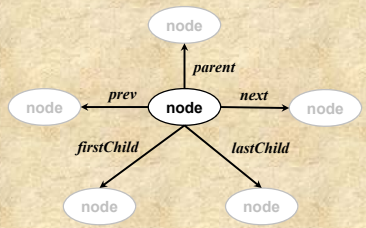
Περιεχόμενα

Μορφές ενδιάμεσου κώδικα (1/11)

- Δύο είναι οι επικρατέστερες μέθοδοι εσωτερικής αναπαράστασης ενδιάμεσου κώδικα
  - Αφηρημένα συντακτικά δέντρα – *abstract syntax trees (AST)*, τα οποία δεν θυμίζουν κάποιου είδους κώδικα
    - πρακτικά είναι επιπλέον αναπαράσταση, καθώς το παρακάτω δε θα το αποφύγουμε
  - Κώδικας τριών διευθύνσεων – *three address code (quads)*, που μοιάζει με το σύνολο εντολών κάποιας αφηρημένης εικονικής μηχανής
    - μπορεί να παραχθεί είτε με επεξεργασία του AST που έχει παραχθεί πρώτο
    - ή *απευθείας με συντακτικά οδηγούμενους κανόνες* (η μέθοδος που θα διδαχθείτε)

Μορφές ενδιάμεσου κώδικα (2/11)

- *Abstract syntax trees (1/4)*
  - Μεγάλη ελευθερία ως προς τον τρόπο αναπαράστασης, αρκεί να «βολεύει» για την παραγωγή τελικού κώδικα
  - Ότι δεν χρειαζόμαστε από το συντακτικό δεν το αποθηκεύουμε στο δέντρο
  - Προγραμματιστικά τα ASTs είναι *n*-ary trees, με κάθε κόμβο να είναι *N(parent, prev, next, firstChild, lastChild)*



•Σε ένα AST μπορούμε να έχουμε κόμβους που περιέχουν σύνθετους τύπους. Π.χ., ένα expression list μπορεί να βολεύει να αναπαρίσταται μέσω κόμβου λίστας, παρά με ανεξάρτητους κόμβους στο ίδιο επίπεδο.

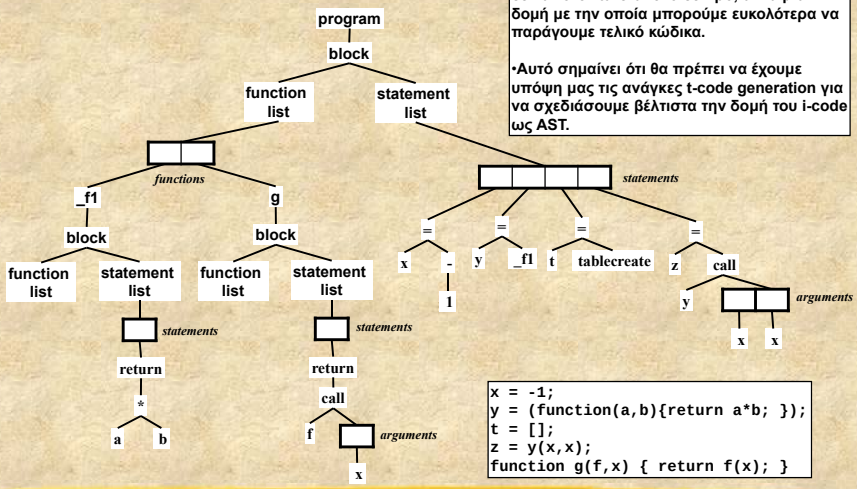
•Είναι συνηθισμένο στα φύλλα ενός AST να εμφανίζονται: αναγνωριστικά ονόματα και σταθερές τιμές. Στην περίπτωση των αναγνωριστικών ονομάτων το φύλλο περιέχει μία αναφορά στο αντίστοιχο στοιχείο του πίνακα συμβόλων.

Μορφές ενδιάμεσου κώδικα (3/11)

- *Abstract syntax trees (2/4)*

•Όπως φαίνεται, η αυστηρή συντακτική δομή δεν αποτυπώνεται στο δέντρο, αλλά μία δομή με την οποία μπορούμε ευκολότερα να παράγουμε τελικό κώδικα.

•Αυτό σημαίνει ότι θα πρέπει να έχουμε υπόψη μας τις ανάγκες t-code generation για να σχεδιάσουμε βέλτιστα την δομή του i-code ως AST.




```
x = -1;
y = (function(a,b){return a*b; });
t = [];
z = y(x,x);
function g(f,x) { return f(x); }
```


Μορφές ενδιάμεσου κώδικα (4/11)

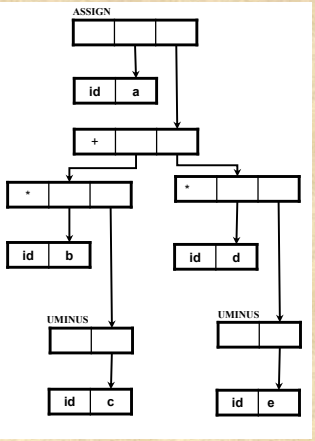
- *Abstract syntax trees (3/4)*

S→ **id** = *expr*;
  {*S.sval* = *node*(*ASSIGN*, *leaf*(*VAR*, **id.***tval*), *expr.sval*); }
expr→**expr**$_1$ + *expr*$_2$
  {*expr.sval* = *node*(*ADD*, *expr*$_1$*.sval*, *expr*$_2$*.sval*); }
expr→**expr**$_1$ - *expr*$_2$
  {*expr.sval* = *node*(*SUB*, *expr*$_1$*.sval*, *expr*$_2$*.sval*); }
expr→ - *expr*$_1$
  {*expr.sval* = *node*(*UMINUS*, *expr*$_1$*.sval*, *NULL*); }
expr→ - *expr*$_1$
  {*expr.sval* = *expr*$_1$*.sval*; }
expr→ id
  {*expr.sval* = *leaf*(*VAR*, **id.***tval*) }

```
union value {
        expr*           e;
        symbol*         s;
};
```

typeof<expr> as *expr.sval* is *expr** field *value.e*
typeof<id> as *id.tval* is *symbol** field *value.s*

Ιδιαίτερα εύκολη η κατασκευή των ASTs με συντακτικά οδηγούμενη μετάφραση



a = b*-c + d * -e

Μορφές ενδιάμεσου κώδικα (5/11)

- *Abstract syntax trees (4/4)*
  - Πλεονεκτήματα
    - Αποτυπώνει συνολικά όλη τη δομή του προγράμματος
    - Είναι εύκολη η αντιμετώπιση των εντολών διακλάδωσης και ανακύκλωσης
    - Μπορεί να χρησιμοποιηθεί και ως είσοδος σε εργαλείο γραφικής αναπαράστασης του κώδικα
  - Μειονεκτήματα
    - Η παραγωγή τελικού κώδικα θέλει περισσότερη δουλειά καθώς το AST είναι αρκετά «μακριά» του t-code
    - Δύσχρηστο για τεχνικές όπως μερική αποτίμηση λογικών εκφράσεων
    - Οι προσωρινές μεταβλητές εμφανίζονται μόνο στον τελικό κώδικα
    - Δεν είναι αναγνώσιμη μορφή παρά μόνο εάν χρησιμοποιηθεί επιπλέον εργαλείο γραφικής παρουσίασης

Μορφές ενδιάμεσου κώδικα (6/11)

- *Three address code – quads* (τετράδες)
  - Ακολουθία από αρχέτυπες εντολές της μορφής
    - x = y *op* z
    - x, y, z είναι τα τρία ορίσματα (μπορεί να είναι και λιγότερα) τα οποία αντιστοιχούν σε αναγνωριστικά ονόματα ή σταθερές
    - *op* είναι δυαδικός ή μοναδιαίος τελεστής, όπως αριθμητική ή λογική πράξη, κλήση συνάρτησης, κλπ.
  - Καθώς πρόκειται για πολύ απλές εντολές, οι πολύπλοκες εκφράσεις πρέπει να τεμαχίζονται κατάλληλα σε αρχέτυπες
    - με αποθήκευση ενδιάμεσων αποτελεσμάτων σε προσωρινές «κρυφές» μεταβλητές, δηλ. όχι σε αυτές που δηλώνει ο ίδιος ο προγραμματιστής,
    - εξασφαλίζοντας ταυτόχρονα ότι δεν πρόκειται ποτέ να υπάρξει πρόβλημα σύγκρουσης ονομάτων
      - αυτό μπορεί να επιτευχθεί ονομάζοντας τις προσωρινές μεταβλητές με αναγνωριστικά ονόματα που δεν επιτρέπονται στη γλώσσα, π.χ. *$2*

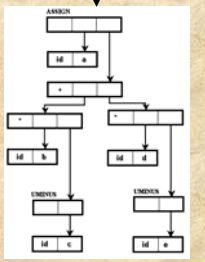
Μορφές ενδιάμεσου κώδικα (7/11)

```
a = b*-c + d * -e
```

three address code

```
$1 = -c;
$2 = b * $1;
$3 = -e;
$4 = d * $3;
$5 = $2 + $4;
a  = $5;
```

abstract syntax tree



•Όταν έχουμε αναγνωριστικά ονόματα ως ορίσματα (π.χ. μεταβλητές προγραμματιστή, προσωρινές μεταβλητές, ονόματα συναρτήσεων χρήστη, εσωτερικά ονόματα ανώνυμων συναρτήσεων, ονόματα συναρτήσεων βιβλιοθήκης), τότε στη θέση του ορίσματος υπάρχει αναφορά (δείκτης) στο αντίστοιχο στοιχείο του πίνακα συμβόλων.

•Όταν έχουμε σταθερά τιμή (π.χ. αριθμοί, string literals, boolean constants), τότε εμφανίζεται απευθείας η ίδια η σταθερά τιμή ως όρισμα.

•Η εσωτερική αποθήκευση και διαχείριση των quads είναι πολύ εύκολη καθώς αρκεί μία απλή σειριακή μέθοδος αποθήκευσης σε λίστα. Επίσης, είναι αρκετά απλό να γράψουμε τον ενδιάμεσο κώδικα σε κάποιο αρχείο κειμένου, ενώ ο κώδικας αυτός είναι συνήθως αρκετά ευανάγνωστος (για τον κατασκευαστή του compiler).

Μορφές ενδιάμεσου κώδικα (8/11)

- *Ρεπερτόριο εντολών για quads (1/3)*

Γενικές εντολές (για τις περισσότερες προστακτικές γλώσσες)

- Εκχώρησης (assignments)
  - Δυαδικού τελεστή, **x = y *op* z**, όπου το αποτέλεσμα της εφαρμογής της πράξης *op* στα ορίσματα *x* και *y* εκχωρείται στο *x*
  - Μοναδιαίου τελεστή, **x = *op* y**, όπου το αποτέλεσμα του εφαρμογής τελεστή *op* στο όρισμα *y* εκχωρείται στο *x*
  - Αντιγραφής, **x = y**, όπου η τιμή του *y* εκχωρείται στο *x*
  - Από στοιχείο πίνακα, **x = y[i]**, όπου η τιμή του στοιχείου του πίνακα *y* η θέση του οποίου προσδιορίζεται από την τιμή του *i* εκχωρείται στο *x*
  - Σε στοιχείο πίνακα, **y[i] = x**, όπου η τιμή του *x* εκχωρείται στο στοιχείου του πίνακα *y* η θέση του οποίου προσδιορίζεται από την τιμή του *i*
- Μετάβασης ελέγχου (jumps)
  - Χωρίς συνθήκη, **goto n**, δηλ. η επόμενη εντολή προς εκτέλεση είναι αυτή στη θέση *n*
  - Με συνθήκη, **if x relop y goto n**, δηλ. εάν ισχύει η συνθήκη *x relop y*, για *relop* συσχετιστικό τελεστή, τότε η επόμενη εντολή προς εκτέλεση είναι αυτή στη θέση *n*

Θα μπορούσαμε εναλλακτικά να ενσωματώσουμε τους συσχετιστικούς τελεστές στα assignments παραπάνω και όχι στα jumps

Μορφές ενδιάμεσου κώδικα (9/11)

- *Ρεπερτόριο εντολών για quads (2/3)*

- Συναρτήσεων (functions)
  - Πραγματικό όρισμα, param x, δηλ. η τιμή του x θεωρείται ως πραγματικό όρισμα συνάρτησης που έπεται.
  - Κλήση, call x n, που σημαίνει κλήση της συνάρτησης της οποίας η διεύθυνση είναι η τιμή του x με συνολικό αριθμό πραγματικών ορισμάτων n, όπου n είναι σταθερά
  - Αρχή ορισμού, funcstart x, μη εκτελέσιμη εντολή που απλώς σηματοδοτεί ότι ακολουθούν οι εντολές για τον ορισμό της συνάρτησης x
  - Τέλος ορισμού, funcend x, μη εκτελέσιμη εντολή που απλώς σηματοδοτεί ότι το πέρας των εντολών για τον ορισμό της συνάρτησης x
  - Επιστροφής με προαιρετικό αποτέλεσμα, return [x], που σημαίνει επιστροφή συνάρτησης με αποτέλεσμα (προαιρετικά) την τιμή του x.
- Απραξία, **nop**, η οποία δεν κάνει απολύτως τίποτε, αλλά χρησιμεύει σε ορισμένα σημεία της παραγωγής ενδιάμεσου κώδικα.

Μορφές ενδιάμεσου κώδικα (10/11)

- *Ρεπερτόριο εντολών για quads (3/3)*

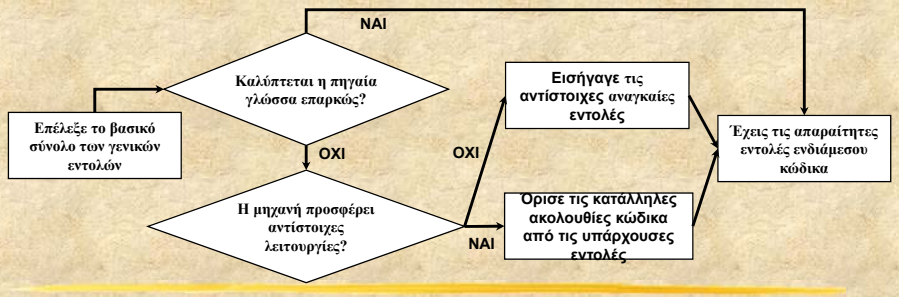
Ειδικές εντολές (λόγω συγκεκριμένων χαρακτηριστικών της γλώσσας)

- Π.χ., για τη διαχείριση δεικτών στη γλώσσα C
  - Εκχώρησης
    - Από διεύθυνσης μεταβλητής, **x = &y**, όπου η διεύθυνση της μεταβλητής *y* εκχωρείται στο *x*
    - Από περιεχόμενο διεύθυνσης, **x = *y**, όπου το περιεχόμενο της διεύθυνσης της μεταβλητής *y* εκχωρείται στο *x*
    - Σε περιεχόμενο διεύθυνσης, ***x = y**, όπου το περιεχόμενο της διεύθυνσης της μεταβλητής *x* εκχωρείται την τιμή του *y*
- Για τη διαχείριση συσχετιστικών πινάκων στη γλώσσα *alpha*
  - Δημιουργία, **newtable t**, δημιουργία ενός νέου πίνακα και εκχώρηση της τιμής του στο *t*

Μορφές ενδιάμεσου κώδικα (11/11)

Η επιλογή των υποστηριζόμενων εντολών είναι μία ιδιαίτερα κρίσιμη απόφαση στη σχεδίαση της ενδιάμεσης αναπαράστασης. Προφανώς το σύνολο των εντολών πρέπει να είναι επαρκές για την υλοποίηση όλων των εντολών της πηγαίας γλώσσας. Πρέπει να λάβετε υπόψη σας τα εξής:

- Ένα μικρό και γενικό σύνολο εντολών είναι ευκολότερο να υλοποιηθεί (παραγωγή τελικού κώδικα) σε κάποια νέα μηχανή
- Ένα περιορισμένο σύνολο εντολών μπορεί να οδηγήσει στην παραγωγή σχετικά μεγάλων ακολουθιών από εντολές ενδιάμεσου κώδικα για αντίστοιχες λειτουργίες της πηγαίας γλώσσας. Η υλοποίηση της μεθόδου βελτιστοποίησης, αλλά και της παραγωγής τελικού κώδικα, γίνεται αρκετά πιο δύσκολη.



Περιεχόμενα

- Ρόλος και εξαρτήσεις
- Μορφές ενδιάμεσου κώδικα
- *Σημασιολογία γλώσσας alpha*
- Εντολές του *alpha* i-code
- Χρήση κρυφών μεταβλητών
- Δηλώσεις και εκχωρήσεις

Σημασιολογία της γλώσσας alpha (1/11)

R-values. Στη γλώσσα *alpha* πληροφορία τύπου δεδομένων κατά τη μεταγλώττιση αποδίδεται μόνο σε r-values (τιμές) του πηγαίου προγράμματος ως εξής:

- Όλες οι σταθερές εκφράσεις φέρουν τον τύπο της αντίστοιχης σταθερής τιμής. Ειδικότερα υπάρχουν οι εξής τύποι τιμών:
  - **Number** (για όλους τους αριθμούς)
  - **String**
  - **Boolean**
  - **Nil** (το μονοσύνολο nil)
- Αναγνωριστικό όνομα συνάρτησης που ορίζει ο χρήστης φέρει τη διεύθυνση της συνάρτησης, εσωτερικά ως ακέραιο αριθμό χωρίς πρόσημο, με τον τύπο **FunctionAddress**.
- Ορισμός συνάρτησης μέσα σε παρενθέσεις είναι τιμή με τη διεύθυνση της συνάρτησης και τύπο **FunctionAddress**.
- Αναγνωριστικό όνομα συνάρτησης βιβλιοθήκης φέρει ως σταθερά string το όνομα της συνάρτησης βιβλιοθήκης, με τύπο **LibraryFunction**.

Σημασιολογία της γλώσσας alpha (2/11)

```
x = 10.12;
y = "hello";
z = (function(){return x; });
function g(a,b){return z(a,b); }
print(x,y,z);
t = [];
t.x = y;
```

R-values (με compile-time value)		L-values (χωρίς compile-time type)	
10.12,	Number	x,	Variable, 0
"hello",	String	y,	Variable, 0
_f1,	FunctionAddress, *?*	z,	Variable, 0
g,	FunctionAddress, *?*	a,	Variable, 1, argument
print,	LibraryFunction, "print"	b,	Variable, 1, argument
		t,	Variable, 0
		t.x,	Table element t[x]

Προσοχή, μόνο αυτά που έχουν scope ουσιαστικά αντιπροσωπεύουν και entries στον symbol table

Σημασιολογία της γλώσσας alpha (3/11)

Λειτουργική σημασιολογία εκφράσεων (runtime semantics)	
Εκχώρηση *l-value* = *r-value*	Ο τύπος και το περιεχόμενο της μεταβλητής l-value αλλάζει ανάλογα με το r-value. Εάν το r-value είναι undefined, τότε έχουμε προειδοποίηση (runtime warning).
Αριθμητικές εκφράσεις **+ - * / % ++ -- -**	Επιτρέπονται μόνο όταν και τα δύο ορίσματα (ή το μοναδικό όρισμα για μοναδιαίους τελεστές) είναι τύπου Number. Το αποτέλεσμα είναι τύπου Number.
Λογικές εκφράσεις με λογικούς τελεστές **&& || !**	Επιτρέπονται μόνο όταν και τα δύο ορίσματα (ή το μοναδικό όρισμα για μοναδιαίους τελεστές) είναι τύπου Boolean, ή μετατρέψιμα σε boolean. Το αποτέλεσμα είναι τύπου Boolean.
Λογικές εκφράσεις με συσχετιστικούς τελεστές διάταξης **> >= < <=**	Επιτρέπονται μόνο όταν και τα δύο ορίσματα είναι τύπου Number. Το αποτέλεσμα είναι τύπου Boolean.
Λογικές εκφράσεις με συσχετιστικούς τελεστές ισότητας **== !=**	Επιτρέπονται μόνο όταν και τα δύο ορίσματα είναι του ιδίου τύπου εκτός από undefined. Το αποτέλεσμα είναι τύπου Boolean. Ως εξαίρεση επιτρέπεται και η σύγκριση *table* με *nil* που επιστρέφει πάντα *false* σε ισότητα και *true* σε ανισότητα.

Εναλλακτικά η σύγκριση είναι πιο ευέλικτη επιστρέφοντας true εάν ισχύει ένα από τα παρακάτω (ομοίως για ανισότητα), πλέον των κανόνων για Nil-Table:

***typeof*(x) == *typeof*(y) ∧ x == y**
***typeof*(x) == Bool ∧ *TrueTest*(y) == x**
***typeof*(y) == Bool ∧ *TrueTest*(x) == x**

Σημασιολογία της γλώσσας alpha (4/11)

Τύπος	***Τρόπος αυτόματης μετατροπής σε Boolean***
Number	`x != 0`
FunctionAddress	`true`
LibraryFunction	`true`
Table	`true`
Nil	`false`
String	`x!= ""`

`x = (function(a,b){return x+y; });`	*Undefined*(x) ← *FunctionAddress*(*K*); Το x γίνεται τύπου *FunctionAddress* με τιμή *K*.
`x="hello";`	*Undefined*(x) ← *String*, "hello"; Το x γίνεται τύπου *String* με τιμή "hello".
`a = b = true;` `c = a > b;`	Η αποτίμηση της έκφρασης **a > b** παράγει λάθος εκτέλεσης (runtime error) καθώς υπάρχει όρισμα που δεν είναι τύπου *Number*.
`x=input();` `y = !x;`	Η αποτίμηση της έκφρασης **!y** εξαρτάται από την τιμή εισόδου στο *x*.
`f(){} g(){}` `b = f>=g;`	Η αποτίμηση της έκφρασης **f >= g** παράγει λάθος εκτέλεσης (runtime error) καθώς υπάρχει όρισμα που δεν είναι τύπου *Number*.

Σημασιολογία της γλώσσας alpha (5/11)

Λειτουργική σημασιολογία εκφράσεων (runtime semantics)	
[]	Το αποτέλεσμα είναι η κατασκευή ενός άδειου συσχετιστικού πίνακα, η διεύθυνση του οποίου συνιστά την αποτίμηση της όλης έκφρασης. Αυτή η τιμή είναι αναγνωρίσιμη ως ο τύπος *Table*.
[*elist* **]**	Το αποτέλεσμα είναι η κατασκευή ενός νέου συσχετιστικού πίνακα, η διεύθυνση του οποίου συνιστά την αποτίμηση της όλης έκφρασης, με στοιχεία στις θέσεις *0,…,n* αντίστοιχα τις τιμές των εκφράσεων της λίστας *elist*.
[*indexed* **]**	Το αποτέλεσμα είναι η κατασκευή ενός νέου συσχετιστικού πίνακα, η διεύθυνση του οποίου συνιστά την αποτίμηση της όλης έκφρασης. Τα στοιχεία του πίνακα είναι οι αντίστοιχες indexed τιμές.
lvalue **. id**	Το *lvalue* πρέπει να είναι τύπου *Table*, αλλιώς runtime error. Η τιμή της έκφρασης είναι το περιεχόμενο του στοιχείου με index **id.val**, όπου **val** η τιμή string του token **id**.
lvalue **[** *expr* **]**	Το *lvalue* πρέπει να είναι τύπου *Table*, αλλιώς runtime error. Η τιμή της έκφρασης είναι το περιεχόμενο του στοιχείου με index την τιμή του *expr*.
func **(** *elist* **)**	Το *func* πρέπει να είναι τύπου *FunctionAddress* ή *LibraryFunction* ή *Functor* (ακολουθεί έπειτα) αλλιώς runtime error.
*lvalue***..id (** *elist* **)**	Το *lvalue* πρέπει να είναι τύπου *Table*, αλλιώς runtime error. Το *lvalue***.id** πρέπει να είναι τύπου *FunctionAddress* ή *LibraryFunction* ή *Functor* αλλιώς runtime error. Η κλήση πραγματοποιείται ισοδύναμα με: *lvalue***.id (** *lvalue, elist* **)**. Η τιμή της έκφρασης είναι το αποτέλεσμα της κλήσης.

Σημασιολογία της γλώσσας alpha (6/11)

Δυναμικοί συσχετιστικοί πίνακες - παραδείγματα

```
t=[];
t.x=10;          // Ίδιο με t["x"] = 10;
t.x.y=20;        // Runtime error: t.x not a Table
t.x.y=[20];
t=[10,20,30];    // Ίδιο με t=[]; t[0]=10; t[1]=20; t[2]=30;
t=[{"x": (function(s){print(s);})}];  // t=[]; t.x = (function(s){print(s);});
t.x("hello");    // Ίδιο τελικά με print("hello");
t.y("world");    // Runtime error: t.y not a function.
t.y=input;       // Εκχώρηση τιμής LibraryFunction "input"
t.y = t.y();     // Ίδιο τελικά με t.y=input();
t.y=nil;         // Διαγραφή του t.y, το nil σβήνει στοιχεία (μη αποθηκεύσιμο)

// Εδώ διαφαίνονται μερικές δυνατότητες οντοκεντρικού προγραμματισμού
function Point(x,y) {
     return [
          {"x" :x},
          {"y" : y"},
          {"move" : (function(this,dx,dy){
               this.x += dx;
               this.y += dy;
          })}
     ]
}
pt = Point(20,30);
pt.move(pt, -1, -2);
pt..move(-1, -2);  // Συντακτική συντομία για το προηγούμενο!
```


Σημασιολογία της γλώσσας alpha (7/11)

Δυναμικοί συσχετιστικοί πίνακες - παραδείγματα

```
// Και επαναχρησιμοποίηση!
function Square(p1,p2) {
     return [
          {"p1" :p1},
          {"p2" :p2},
          {"move" : (function(this,dx,dy){
               this.p1..move(dx, dy);
               this.p2..move(dx, dy);
          })}
     ]
}
sq = Square(Point(10,20), Point(30, 40));
sq..Move(-4, -8);

// Και δυναμική κληρονομικότητα! με merging / concatenation 'd' derived, 'b' base.
function inherit(d, b) {
     for (t = tableindices(b), n=tablength(t)-1; n>0; --n) {
          if (d[t[n]] == nil)
               d[t[n]] = b[t[n]];
     }
}
```


Functors - ένθετο (1/3)

FunctorCall(*t*) {
  if there exists element *x* = *t*["()"] **then**
    if ***x*** is of *FunctionAddress* type **then**
      Call *x* with *t* as an extra argument that is always put first
    else
    if *x* is of *Table* type **then**
      FunctorCall(*x*)
    else *error*()
  else *error*()
}

Τέτοιου είδους πίνακες λέγονται functors και ουσιαστικά υπερφορτώνουν τον function call operator. Συντακτικά χρησιμοποιούνται ως συναρτήσεις.

Functors - ένθετο (2/3)

```
t1 = [ {"()" : (function(t){ print(t); })}];
t1();    // Prints the contents of t
function constmaker(c) {
    return [ {"c" : c},
             {"()" : (function(t){return t.c;})}];
}

c10 = constmaker(10);
print(c10());    // Prints '10'
chello = constmaker("hello");
print(cello());  // Prints 'hello'

f=[{"i" : 0}, {"()" : (function(t){
            ++t.i;
            if (t.i % 2 == 0)
                print("f()");
            return t; // Still a functor.
            })}];
f()()()();   // Prints 'f()f()'
```

Ένα functor είναι ουσιαστικά συνάρτηση με persistent state information, το οποίο λέγεται και closure. Τέτοιου είδους συναρτήσεις υλοποιούνται μερικώς στη C με local static data (δεν μπορούμε όμως να έχουμε για την ίδια συνάρτηση ξεχωριστά states) και στη C++ ως κλάσεις που υλοποιούν τον operator () (με παρόμοιες δυνατότητες όπως η alpha)

Functors - ένθετο (3/3)

```
function delegate(obj, method) {
    return [
        {"x"    : obj },
        {"f"    : method },
        {"()"   : (function(t) { return t.f(t.x); })}
    ];
}

function Attacker(sprite, <other args>) {
    return [ {"sprite" : sprite},
             {"ai" : (function(self) { // essentially ai(void)
                        // AI logic goes here
                      })}
           ];
}

// Somewhere in the code we produce attackers.
a = Attacker(...);
ai_f = delegate(a, a.ai);
// Put all ai_f in a list and call massively
ai_f(); // Equivalent to calling a.ai()
```

Ένα delegate είναι ένα proxy σε method ενός object το οποίο είναι callable, δηλαδή μπορεί να κληθεί ως συνάρτηση. Ενδιαφέρον έχουν delegates για methods που έχουν signature *void (void)* καθώς μέσω delegates επιτρέπουν μαζικές κλήσεις. Π.χ. μπορεί κάποιος να δημιουργήσει delegates για πολλά objects και να τα καλέσει αργότερα μαζικά χωρίς να χρειάζεται πλέον ούτε το object ως μεταβλητή αλλά ούτε και το όνομα του method. Delegates υπάρχουν στη C# αλλά όχι στη C++.

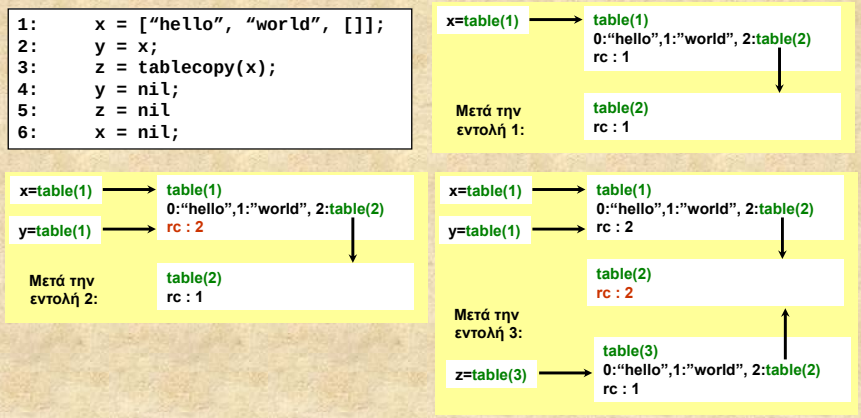
Σημασιολογία της γλώσσας alpha (8/11)

Ειδικοί κανόνες για τους δυναμικούς πίνακες

- Οι πίνακες εκχωρούνται by reference και όχι by value, καθώς δημιουργούνται μόνο μέσω των εκφράσεων δημιουργίας πινάκων όπως αναφέρθηκε προηγουμένως.
- Μπορεί να γίνει copy ενός πίνακα με την tablecopy, αλλά αυτή δεν θα κάνει tablecopy αναδρομικά κάποιο στοιχείο εάν και αυτό είναι πίνακας.
- Επιπλέον οι πίνακες έχουν έναν αριθμό εσωτερικά που κρατάει πόσες μεταβλητές αναφέρονται σε αυτόν, λέγεται reference counter, σύντομα *rc*.
- Όταν μία μεταβλητή που αναφέρεται σε έναν πίνακα καταστραφεί ή αλλάξει περιεχόμενο, τότε αυτόματα μειώνεται και το rc του πίνακα κατά 1.
- Όταν μία μεταβλητή εκχωρηθεί την τιμή αναφοράς ενός πίνακα, τότε αυξάνεται το *rc* κατά 1.
- Όταν το *rc* γίνει 0, αυτόματα καταστρέφεται ο πίνακας (garbage collection).

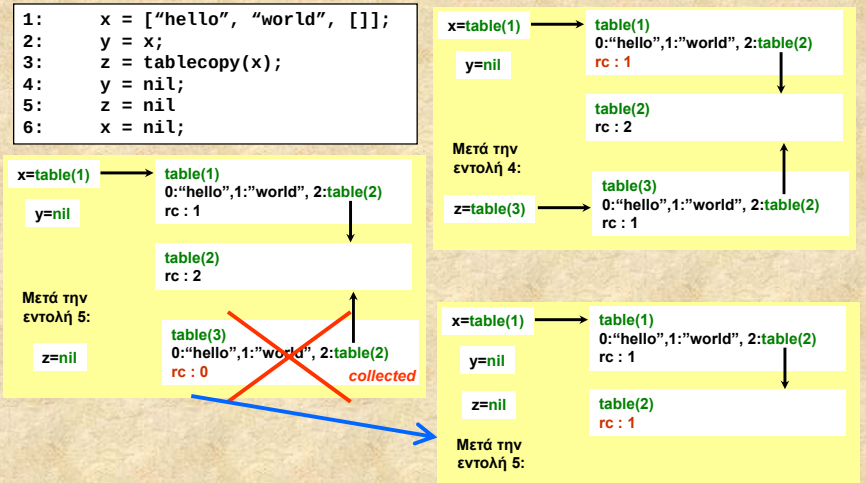
Σημασιολογία της γλώσσας alpha (9/11)

```
1:      x = [“hello”, “world”, []];
2:      y = x;
3:      z = tablecopy(x);
4:      y = nil;
5:      z = nil
6:      x = nil;
```



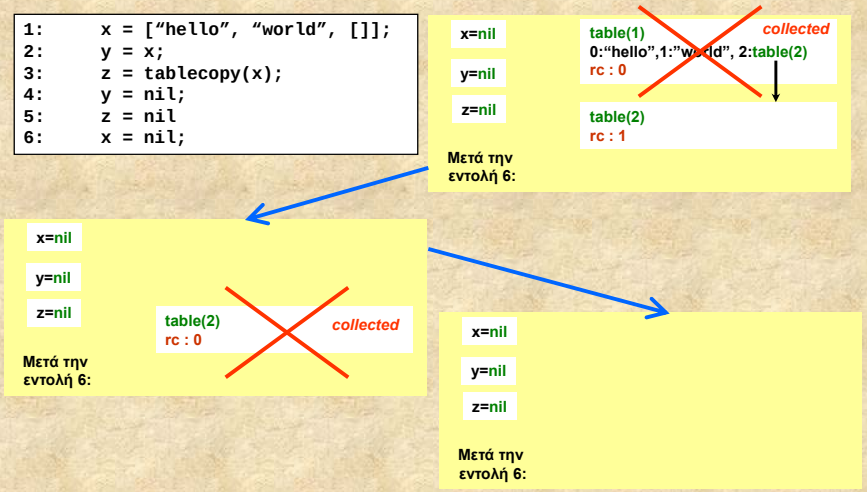
Σημασιολογία της γλώσσας alpha (10/11)

```
1:      x = [“hello”, “world”, []];
2:      y = x;
3:      z = tablecopy(x);
4:      y = nil;
5:      z = nil
6:      x = nil;
```



Σημασιολογία της γλώσσας alpha (11/11)

```
1:      x = "hello", "world", [];
2:      y = x;
3:      z = tablecopy(x);
4:      y = nil;
5:      z = nil
6:      x = nil;
```



Περιεχόμενα

- Ρόλος και εξαρτήσεις
- Μορφές ενδιάμεσου κώδικα
- Σημασιολογία γλώσσας alpha
- *Εντολές του alpha i-code*
- Χρήση κρυφών μεταβλητών
- Δηλώσεις και εκχωρήσεις

Εντολές του alpha i-code (1/4)

Κωδικός εντολής	Επεξήγηση
assign	Η εντολή εκχώρησης με αντιγραφή
add	Οι εντολές εκχώρησης με δυαδικούς αριθμητικούς τελεστές
sub	
mul	
div	
mod	
uminus	Η εντολή εκχώρησης με το μοναδιαίο μείον
and	Οι εντολές εκχώρησης με δυαδικούς λογικούς τελεστές
or	
not	Η εντολή εκχώρησης με τη λογική άρνηση
if_eq	Εντολές αλλαγής ροής ελέγχου (goto) με συσχετιστικούς τελεστές ισότητας
if_noteq	
if_lesseq	Εντολές αλλαγής ροής ελέγχου (goto) με συσχετιστικούς τελεστές διάταξης
if_geatereq	
if_less	
if_greater	
jump	Αλλαγή ροής ελέγχου χωρίς συνθήκη

Εντολές του alpha i-code (2/4)

Κωδικός εντολής	Επεξήγηση
call	Εντολές για συναρτήσεις (κλήση, τροφοδότηση πραγματικού ορίσματος, επιστροφή).
param	
return	
getretval	Εντολή για τη λήψη αποτελέσματος αμέσως μετά από κλήση συνάρτησης.
funcstart	Ψευδο-εντολές για αρχή και τέλος συνάρτησης.
funcend	
tablecreate	Εντολές για τη διαχείριση πινάκων (δημιουργία, εξαγωγή στοιχείου και μεταβολή στοιχείου).
tablegetelem	
tablesetelem	

Εντολές του alpha i-code (3/4)

- Τα quads μπορεί να αντιστοιχούν σε τύπους δεδομένων που φέρουν πλέον των τεσσάρων πεδία
- Κάθε όρισμα του three address code είναι αντίστοιχο με έναν κόμβο ενός abstract syntax tree
  - Έτσι στις περισσότερες των περιπτώσεων φαίνεται να αρκεί κάποιος τύπος έκφρασης
  - Εκτός από την περίπτωση στην οποία έχουμε *goto* εντολές και κάποιο όρισμα πρέπει να είναι quad label (φυσικός αριθμός)

```
enum iopcode {
    assign,          add,             sub,
    mul,             div,             mod,
    uminus,          and,             or,
    not,             if_eq,           if_noteq,
    if_lesseq,       if_geatereq,     if_less,
    if_greater,      call,            param,
    ret,             getretval,       funcstart,
    funcend,         tablecreate,
    tablegetelem,    tablesetelem
};

struct expr;
struct quad {
    iopcode      op;
    expr*        result;
    expr*        arg1;
    expr*        arg2;
    unsigned     label;
    unsigned     line;
};

quad*            quads = (quad*) 0;
unsigned         total = 0;
unsigned int     currQuad = 0;

#define EXPAND_SIZE 1024
#define CURR_SIZE    (total*sizeof(quad))
#define NEW_SIZE     (EXPAND_SIZE*sizeof(quad)+CURR_SIZE)
```


Εντολές του alpha i-code (4/4)

- Βασική λειτουργία στη διαχείριση ενδιάμεσου κώδικα είναι η παραγωγή μίας νέας εντολής, μέσω μίας συνάρτησης που είθισται να λέγεται *emit*
- Επίσης, βολεύει ιδιαίτερα η αποθήκευση σε πίνακα καθώς έτσι το κάθε quad μπορεί να έχει label ίδιο με τη θέση του στον πίνακα γεγονός, που κάνει πολύ γρήγορη την πρόσβαση
- Ο πίνακας αυτός είναι δυναμικός και πρέπει να μπορούμε να τον επεκτείνουμε δυναμικά. *Αυτό σημαίνει ότι δείκτες σε quads απαγορεύονται – μόνος ακέραια indices επιτρέπονται.*

```
void expand (void) {
    assert(total==currQuad);
    quad* p = (quad*) malloc(NEW_SIZE);
    if (quads) {
        memcpy(p, quads, CURR_SIZE);
        free(quads);
    }
    quads = p;
    total += EXPAND_SIZE;
}

void emit (
        iopcode      op,
        expr*        arg1,
        expr*        arg2,
        expr*        result,
        unsigned     label,
        unsigned     line
        ) {

    if (currQuad == total)
        expand();

    quad* p      = quads+currQuad++;
    p->arg1      = arg1;
    p->arg2      = arg2;
    p->result    = result;
    p->label     = label;
    p->line      = line;
}
```


Περιεχόμενα

- Ρόλος και εξαρτήσεις
- Μορφές ενδιάμεσου κώδικα
- Σημασιολογία γλώσσας alpha
- Εντολές του alpha i-code
- *Χρήση κρυφών μεταβλητών*
- Δηλώσεις και εκχωρήσεις

Χρήση κρυφών μεταβλητών (1/6)

- Καθώς ο ενδιάμεσος κώδικας έχει πολύ απλές εντολές είναι αναμενόμενο πολύπλοκες εκφράσεις να τεμαχίζονται σε επιμέρους υπολογισμούς με ενδιάμεσα αποτελέσματα
- Αυτά τα ενδιάμεσα αποτελέσματα θα πρέπει να αποθηκεύονται σε κρυφές μεταβλητές οι οποίες υφίστανται (δηλώνονται) στο scope στο οποίο και είναι χρήσιμες
- Η ονομασία των κρυφών μεταβλητών επιλέγεται να είναι τέτοια ώστε να μη συγκρούεται με αναγνωριστικά ονόματα της γλώσσας
- Οι κρυφές μεταβλητές είναι κανονικές μεταβλητές και η δημιουργία τους απαιτεί δημιουργία νέου συμβόλου στον πίνακα συμβόλων, ενώ απενεργοποιούνται κανονικά εκτός της εμβέλειας δήλωσής τους
- Γενικά χρησιμοποιούμε έναν απλό αλγόριθμο γεννήτριας ονομάτων όπως *$i* ή *_ti* ή *@i*, π.χ. *$1*, *$2*
- Πιο πολλές φορές θα ακούσετε τον όρο temporary variables (προσωρινές μεταβλητές) παρά hidden variables

Χρήση κρυφών μεταβλητών (2/6)

Φαίνεται το πόσο συχνά χρησιμοποιούνται οι κρυφές μεταβλητές. Ο ενδιάμεσος κώδικας που βλέπετε έχει απλοποιήσεις.

Πηγαίος κώδικας	Ενδιάμεσος κώδικας - Quads			
	op	*arg1*	*arg2*	*result/label*
`x = y = 10;`	1: ASSIGN	10		y
	2: ASSIGN	y		x
`z =(x + y) * (x-y);`	3: ADD	x	y	_t0
	4: SUB	x	y	_t1
	5: MUL	_t0	_t1	_t0
	6: ASSIGN	_t0		z
`a = x>=y or y>=z;`	7: IF_GREATEREQ	x	y	10
	8: ASSIGN	false		_t0
	9: JUMP			11
	10:ASSIGN	true		_t0
	11:IF_GREATEREQ	y	z	14
	12: ASSIGN	false		_t1
	13: JUMP			15
	14:ASSIGN	true		_t1
	15:OR	_t0	_t1	_t2
	16:ASSIGN	_t2		a

Χρήση κρυφών μεταβλητών (3/6)

- Θα δούμε αργότερα στους σημασιολογικούς κανόνες παραγωγής ενδιάμεσου κώδικα πότε ακριβώς χρειαζόμαστε τις κρυφές μεταβλητές
- Ένα από τα βασικά θέματα είναι πότε μία κρυφή μεταβλητή που φέρει ένα ενδιάμεσο αποτέλεσμα δύναται να ξαναχρησιμοποιηθεί για την αποθήκευση νέου αποτελέσματος
  - Η επίλυση του προβλήματος αυτού δεν είναι καθόλου προφανής, αφού πρέπει να είμαστε σίγουροι ότι το προηγούμενο ενδιάμεσο αποτέλεσμα δεν υπάρχει περίπτωση να ξαναγίνει απαραίτητο.
  - Εάν δεν επαναχρησιμοποιήσουμε τις κρυφές μεταβλητές, υπερφορτώνουμε τον πίνακα συμβόλων αλλά επιπλέον και το πρόγραμμα με πολλές μεταβλητές, γεγονός που κάνει τον τελικό κώδικα πιο «αργό»

Χρήση κρυφών μεταβλητών (4/6)

- Γενικά υπάρχουν διάφορες ευρεστικές τεχνικές οι οποίες δουλεύουν σωστά υπό συνθήκη (δηλ. όχι σε όλες τις περιπτώσεις)
- Εμάς δεν θα μας απασχολήσει τόσο η βέλτιστη επαναχρησιμοποίηση, όσο το να πετύχουμε ικανοποιητική επαναχρησιμοποίηση κρυφών μεταβλητών.
- Μία τέτοια απλή τακτική που θα εφαρμόσουμε βασίζεται στον εξής κανόνα:
  - *Στο τέλος κάθε statement θα θεωρούμε όλες τις κρυφές μεταβλητές διαθέσιμες.*

Χρήση κρυφών μεταβλητών (5/6)

- Για την παραγωγή μίας προσωρινής μεταβλητής θα εισαγάγουμε τρεις νέες συναρτήσεις:
  - ***newtempname()***, η οποία κάθε φορά που καλείται παράγει έναν νέο όνομα χρησιμοποιώντας μία εσωτερική μεταβλητή tempcounter.
  - ***newtemp()***, η οποία επιστρέφει είτε μία νέα κρυφή μεταβλητή στο παρόν scope ή μία ήδη διαθέσιμη κρυφή μεταβλητή με το όνομα nwetempname
  - ***resettemp()***, η οποία καλείται πάντα με την αναγωγή ενός statement, και κάνει τον tempcounter μηδέν.
- Προϋποθέτουμε την ύπαρξη των συναρτήσεων
  - ***currscope()***, που επιστρέφει την αριθμητική τιμή που χαρακτηρίζει την παρούσα εμβέλεια
  - ***lookup(name, scope)***, που επιστρέφει σύμβολο με συγκεκριμένο όνομα σε συγκεκριμένο scope

Χρήση κρυφών μεταβλητών (6/6)

Οι απλές συναρτήσεις διαχείρισης κρυφών μεταβλητών

```
integer tempcounter = 0;
newtempname()  { return "_t" + tempcounter; }
resettemp()    { tempcounter = 0; }
newtemp() {
        name =  newtempname();
        sym = lookup(name, currscope());
        if sym = nil then
                return newsymbol(name);
        else
                return sym;
}
```


Περιεχόμενα

- Ρόλος και εξαρτήσεις
- Μορφές ενδιάμεσου κώδικα
- Σημασιολογία γλώσσας alpha
- Εντολές του alpha i-code
- Χρήση κρυφών μεταβλητών
- *Δηλώσεις μεταβλητών*

Δηλώσεις μεταβλητών (1/8)

- Θα μας απασχολήσουν μόνο οι δηλώσεις των μεταβλητών. Αναφερόμαστε πάντα στις φανερές μεταβλητές του πηγαίου προγράμματος.
- Οι μεταβλητές απαιτούν μνήμη, επομένως πρέπει να καταγράφουμε για κάθε μεταβλητή το «που ανήκει» και το σειριακό αριθμό της.
- Έχουμε γενικά μεταβλητές τριών κατηγοριών
  - Αυτές που ορίζονται εκτός συναρτήσεων, σε οποιαδήποτε εμβέλεια - ακόμη και σε block: *program variables*
  - Αυτές που ορίζονται στο σώμα μίας συνάρτησης: *function locals*
  - Αυτές που ορίζονται ως τυπικά ορίσματα μίας συνάρτησης: *formal arguments*
- Αυτές τις τρεις διαφορετικές κατηγορίες τις λέμε χώρους εμβέλειας - *scope spaces*

Δηλώσεις μεταβλητών (2/8)

```
lvalue → id
        {       sym = lookup(id.name);
                if sym = nil then {
                        sym = newsymbol(id.name);
                        sym.space = currscopespace();
                        sym.offset = currscopeoffset();
                        inccurrscopeoffset();
                }
                lvalue.sval = lvalue_expr(sym);
        }                       ....για αργότερα

lvalue → local id
        {       sym = lookup(id.name, currscope());
                if sym = nil then {
                        sym = newsymbol(id.name);
                        sym.space = currscopespace();
                        sym.offset = currscopeoffset();
                        inccurrscopeoffset();
                } else { …warning if sym is a function … }
                lvalue.sval = lvalue_expr(sym);
        }                       ....για αργότερα
```

- **Το offset είναι η σειρά εμφάνισης μίας μεταβλητής στο εκάστοτε active scope.**
- **Ξέρουμε ότι οι δηλώσεις μεταβλητών «προκαλούνται» αποκλειστικά από τον κανόνα για *lvalue*, και μάλιστα σε δύο μόνο περιπτώσεις.**
- **Όπως φαίνεται οι δηλώσεις δεν παράγουν ενδιάμεσο κώδικα, αλλά κυρίως προκαλούν εισαγωγές στον πίνακα συμβόλων.**

Δηλώσεις μεταβλητών (3/8)

```
enum scopespace_t {
    programvar,
    functionlocal,
    formalarg
};

enum symbol_t { var_s, programfunc_s, libraryfunc_s };

struct symbol {
    symbol_t        type;
    char*           name;   // Dynamic string.
    scopespace_t    space;  // Originating scope space.
    unsigned        offset; // Offset in scope space.
    unsigned        scope;  // Scope value.
    unsigned        line;   // Source line of declaration.
};

unsigned programVarOffset    = 0;
unsigned functionLocalOffset = 0;
unsigned formalArgOffset     = 0;
unsigned scopeSpaceCounter   = 1;

scopespace_t currscopespace(void) {
    if (scopeSpaceCounter == 1)
        return programvar;
    else
    if (scopeSpaceCounter % 2 == 0)
        return formalarg;
    else
        return functionlocal;
}
```

- Προφανώς η δομή δεδομένων που δίδεται για το symbol δεν είναι πλήρης (π.χ. λείπει το field για τον τύπο του συμβόλου – μεταβλητή, συνάρτηση βιβλιοθήκης, συνάρτηση προγράμματος).
- Κάθε σύμβολο αποθηκεύει το scope space στο οποίο γεννήθηκε.
- Επιπλέον αποθηκεύει και έναν αριθμό σχετικής θέσης εμφάνισης στο εκάστοτε scope space που λέγεται offset.
- Ο τρόπος με τον οποίο αναγνωρίζουμε το παρόν scope space είναι μέσω ενός αριθμού ο οποίος για global scope space είναι πάντα 1, κατά την είσοδο σε formal arguments (ακόμη και κενά) ή σε συνάρτηση αυξάνεται κατά ένα, κατά την έξοδο από συνάρτηση μειώνεται κατά 2.

Δηλώσεις μεταβλητών (4/8)

```
unsigned currscopeoffset (void) {
    switch (currscopespace()) {
        case programvar     : return programVarOffset;
        case functionlocal  : return functionLocalOffset;
        case formalarg      : return formalArgOffset;
        default: assert(0);
    }
}

void inccurrscopeoffset (void) {
    switch (currscopespace()) {
        case programvar     : ++programVarOffset; break;
        case functionlocal  : ++functionLocalOffset; break;
        case formalarg      : ++formalArgOffset; break;
        default: assert(0);
    }
}

void enterscopespace (void)
    { ++scopeSpaceCounter; }

void exitscopespace (void)
    { assert(scopeSpaceCounter>1); --scopeSpaceCounter; }
```

- Για κάθε scope space διατηρώ και ξεχωριστό offset. Προσοχή θέλει το γεγονός ότι εάν «μπαίνουμε» σε ορισμό συνάρτησης μέσα σε άλλη συνάρτηση, το offset της περιέχουσας συνάρτησης πρέπει να «σωθεί» κάπου, ενώ όταν τελειώσει ο ορισμός της περιεχόμενης συνάρτησης να εκχωρηθεί και πάλι στην μεταβλητή του current offset για function locals.
- Με κάθε νέα δήλωση μεταβλητής αυξάνεται το offset του εκάστοτε scope space. Η πληροφορία αυτή μας βοηθάει να ξέρουμε: (α) τον συνολικό αριθμό των δηλωμένων μεταβλητών του προγράμματος, (β) τον αριθμό των τυπικών ορισμάτων της εκάστοτε συνάρτησης, και (γ) τον συνολικό αριθμό των δηλωμένων τοπικών μεταβλητών της εκάστοτε συνάρτησης.

Δηλώσεις μεταβλητών (5/8)

- Η ιδιαίτερη λιτότητα των δηλώσεων οφείλεται στο γεγονός ότι η alpha είναι δυναμική γλώσσα με
  - αυτόματες δηλώσεις μεταβλητών βάσει χρήσης
  - χωρίς ανάγκη στατικής εκχώρησης τύπου δεδομένων στις μεταβλητές
- Αυτό σημαίνει ότι στον πίνακα συμβόλων οι μεταβλητές δεν θα έχουν
  - κάποιο ειδικό πεδίο για τον τύπο δεδομένων
  - κάποιο πεδίο για τον απαιτούμενο χώρο αποθήκευσης σε bytes
    - καθώς ο τύπος μίας μεταβλητής αλλάζει δυναμικά, θεωρούμε ότι μία μεταβλητή γενικά λαμβάνει «μία θέση μνήμης» της εικονικής μηχανής
    - σε κάθε τέτοια θέση μνήμης της εικονικής μηχανής δύναται να αποθηκευτεί τιμή οποιουδήποτε τύπου

→ *Ας δούμε όμως τι θα χρειαζόταν να κάνουμε στην περίπτωση μίας γλώσσας με στατική αντιστοίχηση τύπων και δήλωση μεταβλητών πριν τη χρήση όπως η Pascal, C, C++ ή Java.*

Δηλώσεις μεταβλητών (6/8)

- Έστω οι δηλώσεις μεταβλητών στη γλώσσα C. Θα δούμε μία μικρή απλούστευση του συντακτικού δηλώσεων, ενώ δεν θα ασχοληθούμε με τον ορισμό τύπου δεδομένων

Για λόγους ευκολίας για ένα μη τερματικό σύμβολο α με γνώρισμα α.sval θα χρησιμοποιούμε τον συμβολισμό $α αντί του α.sval

vardecl → *type varlist* ;
{ *distributetype*($*varlist*, $*type*); }
varlist → *varlist* , *var*
{ *append*($*varlist*, $*var*); }
varlist → *var*
{ $*varlist* = *newlist*(); *append*($*varlist*, $*var*); }

var → **id**
{ $*var* = *newsymbol*(**id.name**); $*var.class* = *normalvar*; }
var → *stars* **id**
{ $*var* = *newsymbol*(**id.name**); $*var.class* = *pointervar*; $*var.nest* = $*stars*; }
stars → $stars_1$ *
{ $*stars* = $$stars_1$ + 1; }
stars → *
{ $*stars* = 1; }

Δηλώσεις μεταβλητών (7/8)

```
type → int
    { $type.type = int_t; $type.size = 4; }
type → unsigned int
    { $type.type = unsigned_int_t; $type.size = 4; }
type → float
    { $type.type = float_t; $type.size = 4; }
type → double
    { $type.type = double_t; $type.size = 8; }
type → id
    { sym = lookup(id.name); $type.type = userdefined_t; $type.size = sym.size; }
```

```
var → id array
    { $var = newsymbol(id.name); $var.class = arrayvar; $var.dims= $array; }
array → array [ intconst ]
    { append($array, intconst.value); }
array → [ intconst ]
    { $array = newlist(); append($array, intconst.value); }
var → stars id array
    { $var = newsymbol(id.name);
      $var.class = arraypointervar;
      $var.dims = $array;
      $var.nest = $stars; }
```


Δηλώσεις μεταβλητών (8/8)

```
distributetype(vars, type) {
    for each v in vars do {
        if v.class = pointervar then
            v.size = 4;
        else if v.class = arrayvar then
            v.size = arraytotal(v.dims)*type.size;
        else if v.class = arraypointervar then
            v.size = arraytotal(v.dims)*4;
        else
            v.size = type.size;
        v.offset =getcurroffset();
        inccurroffset(v.size);
    }
}
```

- Είναι προφανές ότι απαιτείται αρκετός κώδικας τόσο για τη διαχείριση τύπων στον πίνακα συμβόλων όσο και για τους σημασιολογικούς κανόνες μετάφρασης.
- Επιπλέον απαιτείται κώδικας για τον στατικό έλεγχο συμφωνίας τύπων (γνωστό ως static ή compile-time type checking).