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 3ο
Παραγωγή Ενδιάμεσου Κώδικα

Type Checking

- Η γλώσσα alpha είναι μια dynamically typed γλώσσα δηλαδή ο τύπος μιας μεταβλητής αλλάζει ακολουθώντας τον τύπο της τιμής που κάθε φορά αποθηκεύεται σε αυτήν
- Συνεπώς δεν μπορούμε να κάνουμε πλήρη έλεγχο τύπων κατά την μεταγλώττιση
  - Αυτό θα γίνει κατά την εκτέλεση

```
if (x > 12)
        a = false;
else
        a = 3.33;
b = (a and c); // ?
```

- Για τις σταθερές εκφράσεις μπορείτε προαιρετικά να κάνετε τους ελέγχους και σε compile time
  - a = true + 12; // Boolean + Number → Error
  - b = “str1” + false; // String + Boolean → Error

Σημασιολογία της γλώσσας alpha (1/7)

- ***Rvalues*** και οι τύποι τους
  - Τα constants έχουν τον τύπο της αντίστοιχης τιμής τους και έχουμε του εξής τύπους
    - Number – όλοι οι αριθμοί
    - String – όλα τα αλφαριθμητικά
    - Boolean – τα ***true*** και ***false***
    - Nil – το ***nil***
  - Τα id των συναρτήσεων χρήστη έχουν ως τιμή τη διεύθυνση της συνάρτησης (θετικός ακέραιος) και έχουν τύπο ***FunctionAddress***
  - Ορισμός συνάρτηση μέσα σε παρενθέσεις είναι τιμή με τη διεύθυνση της συνάρτησης και τύπο ***FunctionAddress***
  - Τα id των library functions έχουν ως τιμή ένα string με το όνομα της συνάρτησης και τύπο ***LibraryFunction***

Σημασιολογία της γλώσσας alpha (2/7)

- Παράδειγμα

```
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]

Σημασιολογία της γλώσσας alpha (3/7)

- ***Εκφράσεις***
  - Εκχώρηση (*lvalue = rvalue)*
    - Ανάθεση τύπου και περιεχομένου από το rvalue στο lvalue
    - Αν το rvalue είναι undefined έχουμε runtime warning
  - Αριθμητικές εκφράσεις (+, -, *, /, %, ++, --, -)
    - Παίρνουν ορίσματα αριθμούς και επιστέφουν αριθμό
  - Συσχετιστικές εκφράσεις διάταξης (>, >=, <, <=)
    - Παίρνουν ορίσματα αριθμούς και επιστέφουν boolean
  - Συσχετιστικές εκφράσεις ισότητας (==, !=)
    - Επιτρέπεται μεταξύ ορισμάτων ίδιου τύπου (όχι όμως undefined) και επιστρέφει boolean
    - Συγκρίνουν και table με το nil επιστρέφοντας false σε ισότητα και true σε ανισότητα
  - Λογικές εκφράσεις (and, or, not)
    - Παίρνουν boolean (ή μετατρέψιμα σε boolean) ορίσματα και επιστέφουν boolean
    - Όλες οι τιμές στη γλώσσα μετατρέπονται σε boolean

Σημασιολογία της γλώσσας alpha (4/7)

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 (5/7)

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*.
lvalue **(** *elist* **)** | *call* **(** *elist* **)**	Το *lvalue* ή *call* πρέπει να είναι τύπου *FunctionAddress ή LibraryFunction*, αλλιώς runtime error. Η τιμή της έκφρασης είναι το αποτέλεσμα της κλήσης.

Σημασιολογία της γλώσσας alpha (6/7)

- Παραδείγματα

```
t = [];              //empty table
t.x = 1;             //insert item at index "x" with value 1
print(t.x, t.y); //prints 1 and nil as t.y does not exists
t2 = [1, true, "lala", print]; //table with numeric indices
t3 = [{0:1},{1:true},{2:"lala"},{3:print}]; //same as above
t4 = [{true:1},{"c":cos},{t2:t3}]; //all key types valid
t2[3](1, 2);  //t2[3] is print, so the call prints 1 and 2
t4[t2][3]("hi"); //t4[t2] is t3, t3[3] is print: prints "hi"

function f(x) { print(x); return f; }
f(1)(2)(3);//f(1) prints 1 and returns f that can be called
           //again:[f(1)](2) prints 2, [f(1)(2)](3) prints 3
t = [ (function f(x){ print(x); return t; }) ];
t[0](1)[0](2)[0](3);//Complex but valid! t[0] gets f and the
//call returns table t for further indexing: prints 1 2 3
```

Σημασιολογία της γλώσσας alpha (7/7)

- Δυναμικοί πίνακες
  - Δημιουργούνται μόνο μέσω της έκφρασης δημιουργίας πίνακα ([], [elist], [indexed])
  - Εκχωρούνται by reference και όχι by value
    - Αντίγραφο ενός πίνακα μπορούμε να πάρουμε με την tablecopy αλλά δεν είναι “deep copy”
  - Γίνονται reference counted και διαγράφονται αυτόματα όταν δεν υπάρχουν πλέον αναφορές σε αυτούς (***garbage collection***)
    - Κατά το runtime θα χρειαστεί να κρατάτε μέσα σε κάθε πίνακα ένα αριθμό που θα κρατάει το πλήθος των μεταβλητών που αναφέρονται σε αυτόν
  - Μπορούν να κληθούν ως συναρτήσεις (***functor tables***) αν υπάρχει στο ειδικό index “()”
    - *Συνάρτηση, οπότε* καλείται η συνάρτηση με πρώτο όρισμα τον πίνακα
    - Αναδρομικά, ένας πίνακας που μπορεί να κληθεί ως συνάρτηση

Κώδικας τριών διευθύνσεων

- Χρησιμοποιείται για την περιγραφή ενός πολύπλοκου προγράμματος με μια ακολουθία απλών εντολών
  - Οι εντολές έχουν όμοια μορφή και χρησιμοποιούν ***τρία*** ορίσματα
    - ***Εντολή: αποτέλεσμα, τελεστής1, τελεστής2***
    - Τα ορίσματα μαζί με την εντολή δημιουργούν μια τετράδα, για αυτό χρησιμοποιείται και ο όρος ***quads***
  - Συνήθως έχουμε μικρό αριθμό εντολών
  - Οι εντολές είναι πολύ κοντά στη γλώσσα μηχανής
    - Σε αρκετές εντολές ενδιάμεσου κώδικα η αντιστοίχηση με τις εντολές μηχανής είναι μια προς μια

Εντολές του alpha i-code (1/2)

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 (2/2)

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	

*Οι εντολές αυτές θα πρέπει να υπάρχουν στον κώδικα ως **enumerated values**, όπως φαίνεται δίπλα.*

```
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
};
```

Δομές δεδομένων για τα quads (1/2)

- Τα ορίσματα που μπορούν να δοθούν σε ένα quad είναι:
  - σταθερές τιμές (constants) συγκεκριμένου τύπου (string, number, boolean)
  - τιμή συνάρτησης βιβλιοθήκης (όνομα)
  - τιμής συνάρτησης προγράμματος (διεύθυνση)
  - κρυφή μεταβλητή (μπορεί να είναι result)
  - μεταβλητή προγράμματος (μπορεί να είναι result)
- Στο result field ενός quad μπορεί να είναι *μόνο μεταβλητή*, καθώς πρέπει να προσφέρει αποθηκευτικό χώρο για αποτέλεσμα

Δομές δεδομένων για τα quads (2/2)

- Τα quads αντιστοιχίζονται σε τύπους δεδομένων με 4 πεδία
- Τα ορίσματα των εντολών αντιστοιχούν σε κόμβους του AST
  - Συνήθως χρησιμοποιούμε κάποιο τύπο έκφρασης
  - Για τις goto εντολές χρειαζόμαστε και ένα όρισμα που να είναι quad label (φυσικός αριθμός)
- Τα quads αποθηκεύονται σε ένα δυναμικό πίνακα
  - Επιτρέπονται μόνο ακέραια indices και όχι δείκτες σε quads!

```
enum expr_t {

    var_e,
    tableitem_e,

    programfunc_e,
    libraryfunc_e,

    arithexpr_e,
    boolexpr_e,
    assignexpr_e,
    newtable_e,

    costnum_e,
    constbool_e,
    conststring_e,

    nil_e,
};
```

```
struct expr {
    expr_t          type;
    symbol*         sym;
    expr*           index;
    double          numConst;
    char*           strConst;
    unsigned char   boolConst;
    expr*           next;
};
```

```
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)
```

Κρυφές Μεταβλητές (1/2)

- Ο ενδιάμεσος κώδικας έχει πολύ απλές εντολές, οπότε πολύπλοκες εκφράσεις θα πρέπει να «τεμαχιστούν» σε επιμέρους υπολογισμούς με ενδιάμεσα αποτελέσματα
- Τα αποτελέσματα αυτά αποθηκεύονται σε κρυφές μεταβλητές
  - Τις ονομάζουμε με τέτοιο τρόπο ώστε να μη συγκρούονται με ids της γλώσσας, π.χ. _t1, _t2
  - Είναι ***κανονικές*** μεταβλητές που απαιτούν δημιουργία νέου συμβόλου στο SymbolTable και απενεργοποιούνται κανονικά όταν βγουν εκτός εμβέλειας

Κρυφές Μεταβλητές (2/2)

- Για παράδειγμα, το x = 1 + 2 * 3 / 4 θα γίνει
  - MUL tmp1, 2, 3 (tmp1 = 2 * 3)
  - DIV tmp2, tmp1, 4 (tmp2 = 2 * 3 / 4)
  - ADD tmp3, 1, tmp2 (tmp3 = 1 + 2 * 3 / 4)
  - ASSIGIN x, tmp3 (x = 1 + 2 * 3 / 4)
- Μπορούμε να ελαχιστοποιήσουμε και τη χρήση προσωρινών μεταβλητών
  - MUL tmp1, 2, 3 (tmp1 = 2 * 3)
  - DIV tmp1, tmp1, 4 (tmp1 = 2 * 3 / 4)
  - ADD tmp1, 1, tmp1 (tmp1 = 1 + 2 * 3 / 4)
  - ASSIGN x, tmp1 (x = 1 + 2 * 3 / 4)
- Στην πράξη δεν μπορούμε πάντα να πετύχουμε τη βέλτιστη επαναχρησιμοποίηση κρυφών μεταβλητών
  - *Θα θεωρήσουμε απλά ότι στο τέλος κάθε statement όλες οι κρυφές μεταβλητές είναι διαθέσιμες*
  - Προαιρετικά, μπορείτε να επαναχρησιμοποιήσετε κρυφές μεταβλητές και σε αρκετά άλλα σημεία:
    - Κλήση συναρτήσεων, κατασκευή πίνακα, μοναδιαίο μείον, λογική άρνηση, αριθμητικές και λογικές εκφράσεις

Μεταβλητές (1/2)

- Έχουμε 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***
- Για κάθε μεταβλητή θα πρέπει να κρατάμε πληροφορία σχετικά με:
  - Το scope space στον οποίο ανήκει
  - Τη σειρά εμφάνισής του μέσα στο scope space (offset)

Μεταβλητές (2/2)

- Το offset θα αυξάνεται μετά από την δήλωση κάθε μεταβλητής κατά 1
  - Ο τύπος κάθε μεταβλητής μπορεί να αλλάζει δυναμικά οπότε θεωρούμε ότι μια μεταβλητή χρειάζεται «μια θέση μνήμης» για αποθήκευση στην εικονική μηχανή
    - Σε statically typed γλώσσες το offset θα αυξανόταν ανάλογα με τον τύπο της κάθε μεταβλητής
- Για κάθε scope space θα πρέπει να διατηρούμε ξεχωριστά offsets
- Μπαίνοντας σε ορισμό συνάρτησης θα πρέπει να ξεκινήσει καινούριο scope space για τις τοπικές μεταβλητές της
  - Θα πρέπει να σώνουμε το τρέχον scope space μπαίνοντας στον ορισμό της και να το επαναφέρουμε βγαίνοντας από αυτή

Παράδειγμα quads με δομές (1/3)

- Είσοδος: ***y = x + 1;***

```
lvalue → id
{
    symbol = ...; //lookup $id or create new
    $lvalue = newexpr(var_e);
    $lvalue->sym = symbol;
}
```

```
const → number
{
    $const = newexpr(constnum_e);
    $const->numConst= $number;
}
```

y: *lvalue → id*

type	var_e
symbol	→

type	var_s
name	"y"
space	program_var
offset	0

x: *lvalue → id*

type	var_e
symbol	→

type	var_s
name	"x"
space	program_var
offset	1

1: *const → number*

type	constnum_e
numConst	1

Παράδειγμα quads με δομές (2/3)

- Είσοδος: ***y = x + 1;***

x+1: *expr → expr + expr*

```
expr → expr1 + expr2 {
    $expr = newexpr(arithexpr_e);
    $expr->sym = newtemp();
    emit(add, $expr1, $expr2, $expr);
}
```

Emitted instructions

```
1 : ADD x 1 _t0
```

quad * for instruction

op	add
arg1	
arg2	
result	

expr * for x

type	***var_e***
symbol	

type	***var_s***
name	***"x"***
space	***program_var***
offset	***1***

type	***constnum_e***
numConst	1

expr * for 1

expr * for x + 1 (result)

type	***arithexpr_e***
symbol	

type	***var_s***
name	***"_t0"***
space	***program_var***
offset	***2***

Παράδειγμα quads με δομές (3/3)

- Είσοδος: ***y = x + 1;***

y=x+1: *assignxpr → lvalue=expr*

```
assignexpr → lvalue = expr {
  emit(assign, $expr, NULL, $lvalue);
  //other checks and emits here as well
}
```

Emitted instructions

```
2 : ASSIGN _t0 y
```

quad * for instruction

op	assign
arg1	
arg2	-
result	

expr * for x + 1 (expr)

type	***arithexpr_e***
symbol	

type	***var_s***
name	***"_t0"***
space	***program_var***
offset	***2***

expr * for y (lvalue)

type	***var_e***
symbol	

type	***var_s***
name	***"y"***
space	***program_var***
offset	***0***

Δυναμικοί πίνακες (1/3)

- Οι δυναμικοί πίνακες δεν είναι «δηλωμένες» μεταβλητές, αλλά εκφράσεις που *δυναμικά* αντιστοιχούν σε χώρο αποθήκευσης, οπότε θέλουν ειδική μεταχείριση

```
...
x = t.a.b.c;
```

```
1: TABLEGETELEM t   "a" _t1   // t["a"]
2: TABLEGETELEM _t1 "b" _t2   // t["a"]["b"]
3: TABLEGETELEM _t2 "c" _t3   // t["a"]["b"]["c"]
4: ASSIGN _t3 x
```

Όταν είναι r-value, μπορώ να παράγω TABLEGETELEM εντολές για το κάθε Index και να χρησιμοποιώ το αποτέλεσμα σαν το νέο στοιχείο.

```
...
t.a.b = x;
```

```
1: TABLEGETELEM t   "a" _t1   // t["a"]
2: TABLESETELEM _t1 "b" x     // t["a"]["b"]
```

Όταν όμως είναι l-value σε εκχώρηση, πρέπει να παράγω μία TABLESETELEM εντολή για το τελευταίο index.

- Πώς ξέρουμε όμως αν πρέπει να παράγουμε την εντολή SET αντί για την εντολή GET έχοντας δει μόνο το t.a.b;

Δυναμικοί πίνακες (2/3)

- Στα στοιχεία πίνακα, δεν παράγουμε επιτόπου κώδικα, αλλά απλά τα καταγράφουμε ως νέα expression τύπου tableElement αποθηκεύοντας τα επιμέρους table και index expressions
- Όταν χρησιμοποιείται ένα lvalue τύπου tableElement παράγουμε κώδικα ανάλογα με τον γραμματικό κανόνα
  - Ως μερική έκφραση σε άλλο κανόνα (π.χ. lvalue.id ή lvalue[expr]) ή για αναγωγή σε r-value), τότε παράγουμε μία εντολή ***TABLEGETELEM***
  - Ως το αριστερό τμήμα μίας εκχώρησης (lvalue=expr), τότε παράγουμε μία εντολή ***TABLESETELEM***

Δυναμικοί πίνακες (3/3)

Στην περίπτωση του έχω *lvalue*.id, και το *lvalue* είναι στοιχείο πίνακα, πρέπει να παράγουμε την εντολή που λαμβάνει αυτό το στοιχείο, καθώς το *lvalue* δεν χρησιμοποιείται το ίδιο για αποθήκευση. Αυτό γίνεται από την emit_iftableitem που φαίνεται δίπλα.

```
expr* emit_iftableitem(expr* e) {
    if (e->type != tableitem_e)
        return e;
    else {
        expr* result = newexpr(var_e);
        result->sym  = newtemp();
        emit(
            tablegetelem,
            e,
            e->index,
            result
        );
        return result;
    }
}
```

tableitem → *lvalue* **. id**

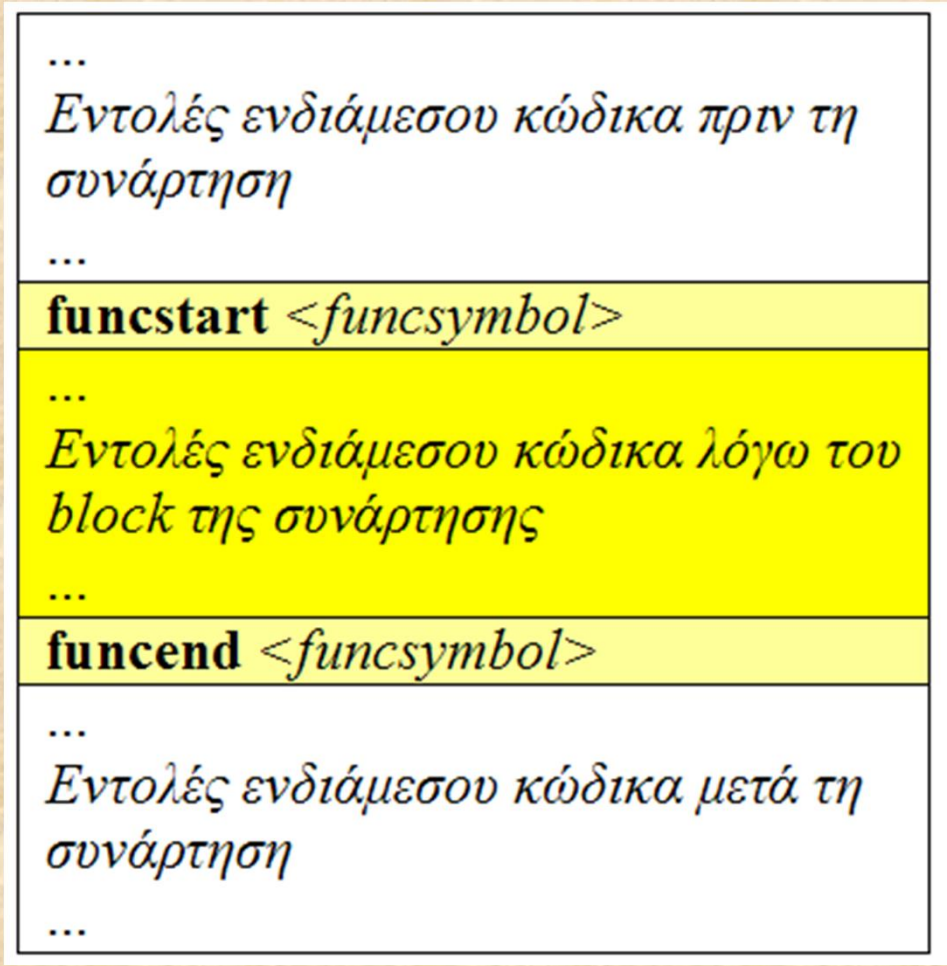
{ *$tableitem = member_item($lvalue,* **id.name**); }

```
expr* member_item (lvalue, name) {
    lvalue = emit_iftableitem(lvalue); // Emit code if r-value use of table item.
    expr*item = newexpr (tableitem_e);  // Make a new expression.
    item->sym = lvalue->sym;
    item->index = newexpr_conststring(name); // Const string index.
}
```

Συναρτήσεις (1/2)

- Για κάθε συνάρτηση που ορίζεται πρέπει να αποθηκεύεται
  - Ο αριθμός των συνολικών τοπικών μεταβλητών
  - Η διεύθυνση της (αριθμός) στον ενδιάμεσο αλλά και τελικό κώδικα (αυτό σε επόμενη φάση)
  - *Δεν* χρειάζεται να κρατήσουμε τον αριθμό των formals
- Μια συνάρτηση μπορεί να βρεθεί σε οποιοδήποτε scope
  - Θα πρέπει μπαίνοντας σε αυτή να κρατάμε τον αριθμό των τοπικών μεταβλητών του τρέχοντος block ώστε να την επαναφέρουμε βγαίνοντας από αυτή
- Ο ενδιάμεσος κώδικας μιας συνάρτησης μπορεί να παραχθεί ανάμεσα σε άλλες εντολές
  - Θεωρούμε ότι δεν είναι εκτελέσιμη εντολή και δε δημιουργεί πρόβλημα (παρακάμπτεται με κάποιο τρόπο)
  - Θα επιστρέψουμε σε αυτό στην επόμενη φάση

Συναρτήσεις (2/2)

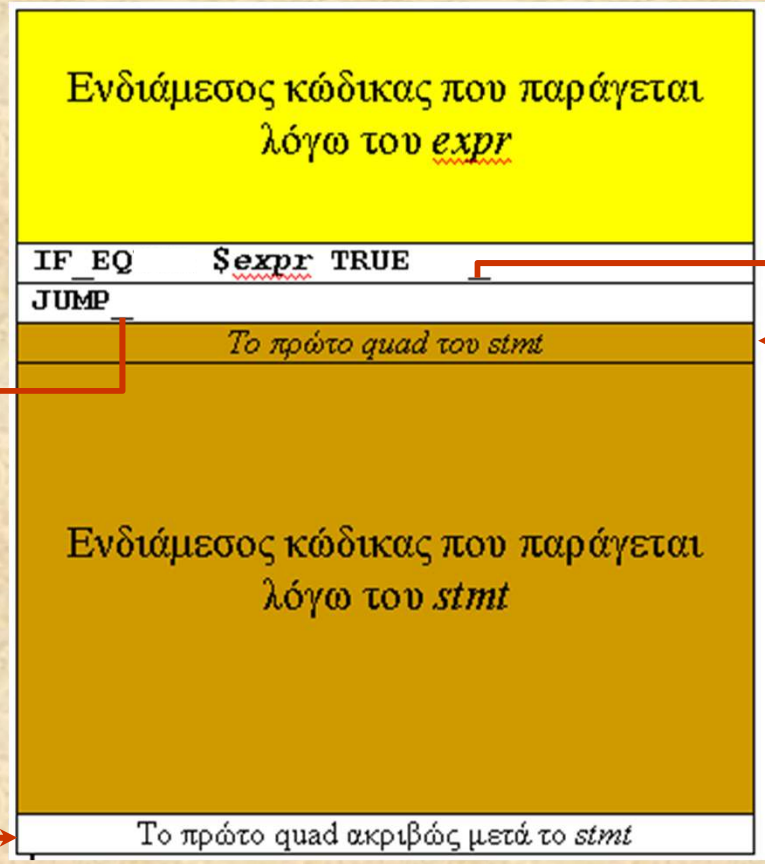



```
z = x + y;
function f(a,b) { return a+b; }
function g() { function h(){} }
x = g;
x =(function(){});
```

```
1:   ADD x y _t1
2:   ASSIGN _t1 z

3:   FUNCSTART f
4:   ADD a b _t1
5:   RETURN _t1
6:   FUNCEND f

7:   FUNCSTART g
8:   FUNCSTART h
9:   FUNCEND h
10:  FUNCEND g

11:  ASSIGN g x
12:  FUNCSTART _f1
13:  FUNCEND _f1

14:  ASSIGN _f1 x
```

if statement

if (*expr*) *stmt*




```
if (x+y >= z-w)
    f(x,y,z);
```

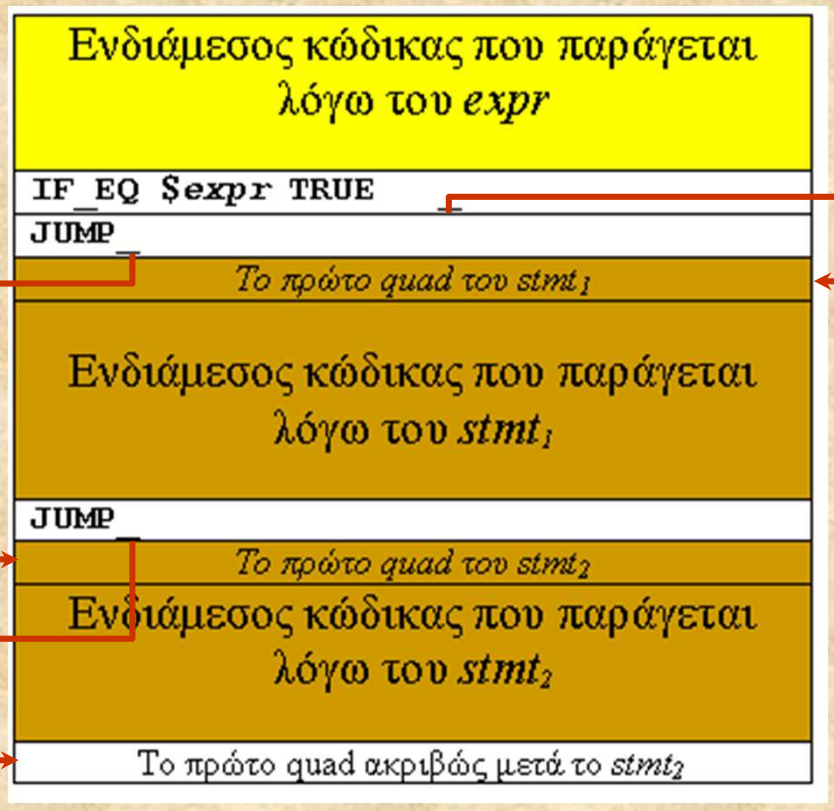
```
1: ADD x y _t1
2: SUB z w _t2
3: IF_GREATEREQ _t1 _t2 6
4: ASSIGN FALSE _t3
5: JUMP 7
6: ASSIGN TRUE _t3
7: IF_EQ TRUE _t3 9
8: JUMP 14

9: PARAM x
10:PARAM y
11:PARAM Z
12:CALL f
13:GETRETVAL _t4

14:
```

go if

skip if

if-else statement

if (*expr*) *stmt* else *stmt*




```
if (a || b)
    x = y;
else
    y = z;
```

```
1: OR a b  t1
2: IF_EQ TRUE _t1 4
3: JUMP 6
4: ASSIGN y x
5: JUMP 7
6: ASSIGN z_y
7:
```

while statement

while (*expr*) *stmt*


```
while (x-y > z-w)
      if (b)
          break;
      else
        x = x-z;
```

```
1:  SUB x y _t1
2:  SUB z w _t2
3:  IF_GREATER _t1 _t2 6
4:  ASSIGN FALSE _t3
5:  JUMP 7
6:  ASSIGN TRUE _t3

7:  IF_EQ TRUE _t3 9
8:  JUMP 16

9:   IF_EQ TRUE b 11
10:  JUMP 13
11:  JUMP 16
12:  JUMP 15
13:  SUB x z _t4
14:  ASSIGN _t4 x

15:  JUMP 1
16:
```

for statement

for (*elist1; expr; elist2*) *stmt*


```
for (i=0; i<N; ++i)
     print("*)
```

```
1: ASSIGN 0 i

2: IF_LESS i N 5
3: ASSIGN FALSE _t1
4: JUMP 6
5: ASSIGN TRUE _t1

6: IF_EQ TRUE _t1 10
7: JUMP 14

8: ADD i 1 i
9: JUMP 2

10: PARAM "*"
11: CALL "print"
12: GETRETVAL _t2
13: JUMP 8

14:
```

Ειδικές εντολές για loops (1/3)

- ***Break*** – Αλλάζει τη ροή ελέγχου ώστε να έχουμε άμεση έξοδο από το τρέχον loop
- ***Continue*** – Αλλάζει τη ροή ελέγχου ώστε να σταματήσει η τρέχουσα ανακύκλωση και να ξεκινήσει αμέσως η επόμενη στο τρέχον loop
- Και οι δύο αυτές επιτρέπονται μόνο μέσα σε κάποιο loop
  - Θα χρειαστεί ένας μετρητής για τα loops
    - Θα πρέπει να σώνεται μπαίνοντας σε συναρτήσεις
    - Και να επαναφέρεται βγαίνοντας από αυτές

Ειδικές εντολές για loops (2/3)

- Τα *break* και *continue* κάνουν jump σε σημεία που δεν είναι διαθέσιμα μέσα στον ίδιο γραμματικό κανόνα (incomplete jumps)
- Πρέπει λοιπόν να τα κρατήσουμε κάπου και τα κάνουμε patch όταν είναι διαθέσιμα τα targets τους
  - Εισάγουμε δύο λίστες, τις ***breaklist*** και ***continuelist*** που θα περιέχουν τους αριθμούς των quads που αντιστοιχούν σε unfinished jumps λόγω break και continue αντίστοιχα
  - Κάνουμε patch στους γραμματικούς κανόνες των loops

```
break → break ;
        { $break.breaklist = newlist(nextquad()); emit(jump, _); }
continue → continue;
        { $ continue.contlist = newlist(nextquad()); emit(jump, _); }

stmts → stmt { $stmts = $stmt; }
stmts → stmts1 stmt
        {
                $stmts.breaklist = merge($stmts1.breaklist, $stmt.breaklist);
                $stmts. contlist  = merge($stmts1. contlist, $stmt. contlist);
        }
```

Ειδικές εντολές για loops (3/3)

- Παράδειγμα

```
while (a) {                 1:  IF_EQ TRUE a 3
                            2:  JUMP 15        // go exit

   a = f(a);                3:  PARAM a
                            4:  CALL f
                            5:  GETRETVAL _t1
                            6:  ASSIGN _t1 a

   if (a)                   7:  IF_EQ TRUE 9   // go if
      break;                8:  JUMP 11        // go else
   else                     9:  JUMP 15        // break
      continue;             10: JUMP 12        // skip else
                            11: JUMP 1         // continue

   continue;                12: JUMP 1         // continue
   break;                   13: JUMP 15        // break

                            14: JUMP 1         // go loop
}                           15:
```

Αποτίμηση εκφράσεων - Backpatching

- Για την αποτίμηση των λογικών εκφράσεων υπάρχουν δύο τεχνικές
- **Ολική αποτίμηση**
  - Αποτιμάται ολόκληρη η έκφραση
  - Π.χ. ο ενδιάμεσος κώδικας για το ***a < b or c < d*** με ολική αποτίμηση φαίνεται δίπλα

```
1: IF_LESS    a       b       4
2: ASSIGN     FALSE   _t1
3: JUMP       5
4: ASSIGN     TRUE    _t1
5: IF_LESS    c       d       8
6: ASSIGN     FALSE   _t2
7: JUMP       9
8: ASSIGN     TRUE    _t2
9: OR         _t1     _t2     _t3
```

- **Μερική αποτίμηση** (short circuit evaluation)
  - Αποτιμάται μόνο το ελάχιστο αναγκαίο τμήμα της έκφρασης από το οποίο μπορεί να εξαχθεί συμπέρασμα για την τιμή της
  - Αυτή είναι η τεχνική ακολουθείται σε γλώσσες όπως C/C++/Java και ***είναι η default σημασιολογία στην alpha***
  - Απαιτεί ειδική παραγωγή κώδικα γνωστή ως ***backpatching***

Μερική αποτίμηση έκφρασης (1/5)

- Εισάγουμε δύο λίστες *truelist* και *falselist* που θα περιέχουν πληροφορία για τα αποτελέσματα των μερικών εκφράσεων
  - Θα περιέχουν labels σε εντολές που πρέπει να κάνουμε jump για να αποτιμήσουμε το ελάχιστο αναγκαίο τμήμα της έκφρασης
  - Τα labels που αντιστοιχούν σε αυτές τις εντολές προστίθενται στην κατάλληλη από τις δύο λίστες
- Καθώς παράγεται ο ενδιάμεσος κώδικας για την έκφραση οι JUMP εντολές δεν συμπληρώνονται (*incomplete jumps*)
  - Δε γνωρίζουμε από πριν που πρέπει να κάνουμε jump
  - Τα jumps συμπληρώνονται αργότερα από τη *backpatch*
    - backpatch(list, quad) – κάθε incomplete jump που περιέχεται στη λίστα θα συμπληρωθεί με το quad που δίνεται

Μερική αποτίμηση έκφρασης (2/5)

- Γραμματική

Στον 1ο κανόνα, αν κάποιο από τα E1 και E2 είναι true τότε και το E θα είναι true, άρα η συνολική truelist προκύπτει συνδυάζοντας τις επιμέρους. Αν το E1 είναι false, τότε πρέπει να εξεταστεί το E2, άρα η συνολική falselist προκύπτει από το E2. Όμοια και τα άλλα.

```
E ->    E1 or M E2      {backpatch (E1.falselist, M.quad);
                          E.truelist = merge (E1.truelist,E.2truelist);
                          E.falselist = E2.falselist;
                        }

E ->    E1 and M E2     {backpatch (E1.truelist, M.quad);
                          E.truelist = E2.truelist;
                          E.falselist = merge (E1.falselist,E2.falselist);
                        }

E ->    not E1          {E.truelist = E1.falselist;
                          E.falselist = E1.truelist;
                        }

E->     id1 relop id2   {E.truelist = makelist(nextquad);
                          E.falselist = makelist(nextquad+1);
                          emit(IF_RELOP, id1.val, id2.val, _);
                          emit(JUMP, _);
                        }

M ->    ε               { M.quad = nextquad; }
```

Μερική αποτίμηση έκφρασης (3/5)

- Παράδειγμα: ***a < b or c < d and e < f***
  - Γίνεται reduced το *a* < *b* από το E→ id1 relop id2
  - Γίνεται reduced το *c* < *d* από το E→ id1 relop id2
  - Γίνεται reduced το *e* < *f* από το E→ id1 relop id2
  - Γίνεται reduce από το E→ E1 and M E2 (M= 5)
  - Γίνεται reduce από το E→ E1 or M E2 (M=3)

```
1: IF_LESS     a      b      _
2: JUMP        3
3: IF_LESS     c      d      5
4: JUMP        _
5: IF_LESS     e      f      _
6: JUMP        _
```

*Η αποκοπή είναι λανθασμένη, αν εκτελεστούν οι JUMP εντολές με label **1** ή **5**, ενώ είναι ψευδής αν εκτελεστούν οι JUMP εντολές με label **4** ή **6***

Backpatch({2}, 6)

True list: {3,5}

False list: {4,6}

Μερική αποτίμηση έκφρασης (4/5)

- Όταν μια λογική έκφραση χρησιμοποιηθεί θα πρέπει να προστεθούν ***extra εντολές*** για να δώσουν τελικά ένα boolean αποτέλεσμα
- Επίσης θα πρέπει να γίνουν κατάλληλα patched τα υπόλοιπα incomplete jumps που βρίσκονται στις truelist και falselist

x = a < b or c < d and e < f

*Η έκφραση είναι αληθής, αν εκτελεστούν οι JUMP εντολές με label **1** ή **5** ενώ είναι ψευδής αν εκτελεστούν οι JUMP εντολές με label **4** ή **6***
*Τα jumps με label **1** και **5** θα γίνουν patched στο **ASSIGN true** και αυτά με label **4** και **6** στο **ASSIGN false.***

```
1 : IF_LESS     a       b       7
2 : JUMP        3
3 : IF_LESS     c       d       5
4 : JUMP        9
5 : IF_LESS     e       f       7
6 : JUMP        9
7 : ASSIGN      TRUE    _t0
8 : JUMP        10
9 : ASSIGN      FALSE   _t0
10: ASSIGN      _t0     x
```

Μερική αποτίμηση έκφρασης (5/5)

- Extra εντολές χρειάζονται επίσης όταν τα operands της λογικής έκφρασης δεν είναι ήδη boolean
  - Π.χ. x = a and b; y = f() or g() or h(); z = not x;
- Κάθε non-boolean operand θα πρέπει να μετατραπεί σε boolean με ένα “true-test” φτιάχνοντας παράλληλα τις truelist και falselist που του αντιστοιχούν

x = a and not b

Το a είναι αληθές αν είναι ίσο με TRUE, αλλιώς είναι ψευδές. Αρα παράγουμε το IF_EQ (1) που ανήκει στην truelist, και το JUMP (2) που ανήκει στη falselist. Ομοίως για το b, απλά το not αλλάζει τις λίστες και τελικά το label 3 ανήκει στην falselist και το 4 στην truelist.

```
1 : IF_EQ       a       TRUE    3
2 : JUMP        7
3 : IF_EQ       b       TRUE    7
4 : JUMP        5
5 : ASSIGN      TRUE    _t0
6 : JUMP        8
7 : ASSIGN      FALSE   _t0
8 : ASSIGN      _t0     x
```