



## HY340 : 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



## HY340 : ΓΛΩΣΣΕΣ ΚΑΙ ΜΕΤΑΦΡΑΣΤΕΣ

### Διάλεξη 10η ΠΑΡΑΓΩΓΗ ΕΝΔΙΑΜΕΣΟΥ ΚΩΔΙΚΑ - II

HY340

A. Σαββίδης

Slide 2 / 38



## Περιεχόμενα

- Δηλώσεις συναρτήσεων
- Εκφράσεις αποθήκευσης
- Βασικές εκφράσεις
- Επαναχρησιμοποίηση κρυφών μεταβλητών

HY340

A. Σαββίδης

Slide 3 / 38



## Δηλώσεις συναρτήσεων (1/9)

- Ο ορισμός συναρτήσεων επιτρέπεται σε οποιοδήποτε σημείο του κώδικα
  - είτε ως μη εκτελέσιμη εντολή
  - ή ως έκφραση σε παρενθετική μορφή που φέρει τη διεύθυνση της συνάρτησης
- Κάθε τέτοιος σωστός ορισμός οδηγεί και στη δήλωση ενός νέου συμβόλου κατηγορίας συνάρτησης
  - είτε με εξωτερικό αναγνωριστικό όνομα
  - ή με κρυφό όνομα συνάρτησης
- Για κάθε ορισμένη συνάρτηση πρέπει να αποθηκεύεται
  - ο αριθμός των συνολικών τοπικών μεταβλητών
  - η διεύθυνση της (αριθμός) στον ενδιάμεσο αλλά και τελικό κώδικα (η τελευταία θα λάβει τιμή μόνο όταν εφαρμοστεί η παραγωγή τελικού κώδικα)

HY340

A. Σαββίδης

Slide 4 / 38



## Δηλώσεις συναρτήσεων (2/9)

Η διαχείριση του scope παραλείπεται καθώς είναι πολύ απλή. Παρατηρήστε ότι τεμαχίζουμε τον αρχικό γραμματικό κανόνα σε περισσότερους, καθώς θέλουμε επιμέρους σημασιολογικούς κανόνες με αντίστοιχο τύπο για το αριστερό τερματικό σύμβολο. Π.χ., πριν τα formal arguments, αυξάνουμε το scope counter με την κλήση της `enterscopespace` και μηδενίζουμε το offset για τα formal arguments με την `resetformalargsoffset`.

```
funcname → id
        { $funcname = id.name; }

funcname → ε
        { $funcname = newtempfuncname(); }

funprefix → function funcname
        {
            $funprefix = newsymbol($funcname, function_t);
            $funprefix.iaddress = nextquadlabel();
            emit(FUNCSTART, , lvalue_expr($funprefix));
            push(functionLocalsStack, functionLocalOffset); // Save current offset.
            enterscopespace(); // Entering function arguments scope space.
            resetformalargsoffset(); // Start formals from zero.
        }
```

Ο ενδιάμεσος κώδικας για μία συνάρτηση μπορεί να παραχθεί «στη μέση» άλλων εντολών, χωρίς να συνιστά εκτελέσιμη εντολή (θεωρούμε ότι παρακάμπτεται με κάποιο τρόπο).



## Δηλώσεις συναρτήσεων (3/9)

Και πάλι εισάγουμε ειδικό κανόνα για τα formal arguments καθώς θέλουμε να σηματοδοτήσουμε την είσοδο στο scope space των function locals, ενώ ταυτόχρονα θα μηδενίσουμε το αντίστοιχο offset. Επιπλέον, επειδή θα χρειαστεί να κρατήσουμε κάπου τον αριθμό των locals της συνάρτησης, χρησιμοποιούμε ένα γραμματικό κανόνα για το block της συνάρτησης (funcbody) και αποθηκεύουμε στο γνώρισμά του τον αριθμό αυτό. Επίσης, με την έξοδο από το σώμα μίας συνάρτησης πρέπει να μειώσουμε τον μετρητή για το παρόν scope space με την `exitscopespace`.

```
funcargs → (idlist)
        {
            enterscopespace(); // Now entering function locals space.
            resetfunctionlocalsoffset; // Need to start locals from zero.
        }

funcbody → block
        {
            existscopespace(); // Exiting function locals space.
        }
```

Προσοχή: η τιμή του scope space counter θα συμβαδίζει με την τιμή στο τρέχων scope μόνο εάν τα formal arguments είναι σε υψηλότερο scope (μικρότερη τιμή) κατά ένα από ότι τα function locals. Εάν δε συμβαίνει αυτό τότε χρειαζόμαστε ξεχωριστή μεταβλητή για scope.



## Δηλώσεις συναρτήσεων (4/9)

Ο ολοκληρωμένος συντακτικά οδηγούμενος ορισμός για τις συναρτήσεις, ο οποίος τελειώνει με την συμπλήρωση του αριθμού των function locals στο σύμβολο της συνάρτησης. Με το πέρας μίας συνάρτησης θα έχει ολοκληρωθεί και η παραγωγή ενδιάμεσου κώδικα για το block, και τελειώνουμε με την εντολή `FUNCEND`.

```
funcdef → funcprefix funcargs funcbody
        {
            existscopespace(); // Exiting function definition space.
            $funcprefix.totallocals = functionLocalOffset; // Store #locals in symbol entry.
            functionLocalOffset = pop(functionLocalsStack); // Restore saved scope offset.
            $funcdef = $funcprefix; // The function definition returns the symbol.
            emit(FUNCEND, lvalue_expr($funcprefix));
        }

typeof<funcname>:      string
typeof<funcbody>:      integer
typeof<funcprefix, funcdef>: symbol*
```



## Δηλώσεις συναρτήσεων (5/9)

...  
Εντολές ενδιάμεσου κώδικα πριν τη συνάρτηση  
...

**funcstart <funcsymbol>**

...  
Εντολές ενδιάμεσου κώδικα λόγω του block της συνάρτησης  
...

**funcend <funcsymbol>**

...  
Εντολές ενδιάμεσου κώδικα μετά τη συνάρτηση  
...

Εδώ είναι απολύτως φυσιολογικό να έχουμε εσωτερικά παραγωγή ενδιάμεσου κώδικα για ορισμό συναρτήσεων



## Δηλώσεις συναρτήσεων (6/9)

```
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
```

Τα δύο αυτά `_t1`  
είναι διαφορετικά  
(σε διαφορετική  
εμβέλεια)



## Δηλώσεις συναρτήσεων (7/9)

Συμπλήρωση μερικών από τις βοηθητικές συναρτήσεις. Προσθέτουμε και μία συνάρτηση `patchlabel` για να συμπληρώσουμε ένα αρχικά undefined label για κάποιο quad, αρκεί να είχαμε «σώσει» κάπου το index του.

```
void resetformalargoffset(void)
{ formalArgOffset = 0; }

void resetfunctionlocaloffset(void)
{ functionLocalOffset = 0; }

void restorecurrscopeoffset(unsigned n) {
    switch (currscopeSpace()) {
        case programvar : programVarOffset = n; break;
        case functionlocal : functionLocalOffset = n; break;
        case formalarg : formalArgOffset = n; break;
        default : assert(0);
    }
}

unsigned nextquadlabel(void)
{ return currQuad; }

void patchlabel(unsigned quadNo, unsigned label) {
    assert(quadNo < currQuad);
    quads[quadNo].label = label;
}
```



## Δηλώσεις συναρτήσεων (8/9)

- Γιατί δεν κρατάμε πουθενά τον αριθμό των formal arguments των συναρτήσεων;
  - Γιατί ποτέ δεν απαιτείται στατικός έλεγχος ως προς τον αριθμό των πραγματικών ορισμάτων στη κλήση συνάρτησης καθώς:
    - ◆ Δεν είναι πάντα εφικτό, αφού οποιαδήποτε μεταβλητή μπορεί να φέρει διεύθυνση συνάρτησης, ανάλογα με τη λογική εκτέλεσης
    - ◆ Επιτρέπουμε να υπάρχει διαφορετικός αριθμός πραγματικών ορισμάτων από ότι τα τυπικά ορίσματα
    - ◆ Τα τυπικά ορίσματα στην alpha είναι προαιρετικά, ως συντακτική ευκολία στον προγραμματιστή της συνάρτησης, καθώς μπορεί να «βλέπει» όλα τα actual arguments μέσω των συναρτήσεων βιβλιοθήκης `totalarguments` και `argument`



## Δηλώσεις συναρτήσεων (9/9)

```
function dumparguments() {
    for (local i = 0, local n = totalarguments(); i < n; ++i)
        { print(argument(i)); print("\n"); }
}
```

```
dumparguments(23, true, "hello");
dumparguments();
dumparguments(
    print,
    dumparguments,
    (function){}),
    []
);
```

23  
TRUE  
hello

LibraryFunction("print")  
ProgramFunction("dumparguments", address n)  
ProgramFunction(anonymous, address κ)  
Table[]





## Περιεχόμενα

- Δηλώσεις συναρτήσεων
- **Εκφράσεις αποθήκευσης**
- Βασικές εκφράσεις
- Επαναχρησιμοποίηση κρυφών μεταβλητών



## Εκφράσεις αποθήκευσης (1/11)

- Πρόκειται για τα λεγόμενα *l-values*, δηλ. τις εκφράσεις που μπορούν να εμφανιστούν στο αριστερό τμήμα μίας εντολής εκχώρησης
  - αντιπροσωπεύοντας τον αποθηκευτικό χώρο του προγράμματος που έχει στη διάθεση του ο προγραμματιστής
  - γνωρίζουμε ότι υπάρχει και άλλος αποθηκευτικός χώρος, οι λεγόμενες κρυφές μεταβλητές, αλλά αυτές δεν είναι ποτέ στη διάθεση του προγραμματιστή
    - ◆ παρά το γεγονός ότι παράγουμε ενδιάμεσο κώδικα που τα χρησιμοποιεί ως *l-values* (δηλ. μπορούν να εμφανίζονται ως *result* για ASSIGN quads)



## Εκφράσεις αποθήκευσης (2/11)

- Πριν μιλήσουμε για τα *l-values* είναι σημαντικό να ξεκαθαρίσουμε τι είδους ορίσματα δίνουμε σε κάθε quad:
  - σταθερές τιμές (constants) που είναι τιμές συγκεκριμένου τύπου (string, number, boolean)
  - τιμή συνάρτησης βιβλιοθήκης (όνομα)
  - τιμές συνάρτησης προγράμματος (διεύθυνση)
  - κρυφή μεταβλητή (μπορεί να είναι result)
  - μεταβλητή προγράμματος (μπορεί να είναι result)
- Άρα το *result field* ενός quad μπορεί να είναι **μόνο μεταβλητή** (καθώς πρέπει να προσφέρει αποθηκευτικό χώρο για αποτέλεσμα).
  - Με βάση αυτή την παρατήρηση συνεχίζουμε



## Εκφράσεις αποθήκευσης (3/11)

- Τα ήδη των βασικών αποθηκευτικών εκφράσεων στη γραμματική μας είναι δύο:
  - ανεξάρτητες «δηλωμένες» μεταβλητές (ενδέχεται να μην καταλήξουν πάντα σε μεταβλητές, εάν το **id** ή το **global id** κάνουν resolve σε κάτι άλλο, π.χ. συνάρτηση)
    - ◆ *lvalue* → **id** | **local id** | **global id**
  - στοιχεία πίνακα
    - ◆ *lvalue* → *tableitem*
- Καθώς όλα αυτά αντιστοιχούν σε τύπο έκφρασης, μένει να δούμε τον τύπο δεδομένων για τις εκφράσεις, καθώς και σε ποια γραμματικά σύμβολα συνολικά αναφέρεται



## Εκφράσεις αποθήκευσης (4/11)

// Expression types. You use only the types you really need for i-code generation, so you may drop some entries.

```
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,
};
```

Γραμματικά σύμβολα με τύπο *expr*

- *lvalue*
- *member*
- *primary*
- *assignexpr*
- *call*
- *term*
- *objectdef*
- *const*

// For simplicity this is a superset type, but you may hack around with more clever storage (if you like).

```
struct expr {
    expr_t      type;
    symbol*     sym;
    expr*       index;
    double      numConst;
    char*       strConst;
    unsigned char boolConst;
    expr*       next; // Just to make trivial s-lists.
};
```

HY340

A. Σαββίδης

Slide 17 / 38



## Εκφράσεις αποθήκευσης (5/11)

*lvalue* → *id*  
 { ... \$lvalue = *lvalue\_expr*(sym); }  
*lvalue* → *local id*  
 { ... \$lvalue = *lvalue\_expr*(sym); }  
*lvalue* → *global id*  
 { ... \$lvalue = *lvalue\_expr*(sym); }

// Making an l-value expression out of a symbol is simple, since the expression inherits the symbol type. Also, getting information like 'library function' name or the 'program function' address (after code generation), is straightforward through the symbol 'sym' field.

```
expr* lvalue_expr (symbol* sym) {
    assert(sym);
    expr* e = (expr*) malloc(sizeof(expr));
    memset(e, 0, sizeof(expr));
    e->next = (expr*) 0;
    e->sym = sym;

    switch (sym->type) {
        case var_s      : e->type = var_e; break;
        case programfunc_s : e->type = programfunc_e; break;
        case libraryfunc_s : e->type = libraryfunc_e; break;
        default: assert(0);
    }

    return e;
}
```

HY340

A. Σαββίδης

Slide 18 / 38



## Εκφράσεις αποθήκευσης (6/11)

- Εκτός από τις μεταβλητές, χώρο αποθήκευσης λαμβάνουν και τα στοιχεία των δυναμικών πινάκων.
- Καθώς δεν αποτελούν «δηλωμένες» μεταβλητές, αλλά εκφράσεις που δυναμικά αντιστοιχούν σε χώρο αποθήκευσης, θέλουν ειδική μεταχείριση. Αυτό φαίνεται καλύτερα με δύο παραδείγματα

...	1: TABLEGETELEM t "a" _t1 // t["a"]
x = t.a.b.c;	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, μπορεί να παράγω TABLEGETELEM εντολές για το κάθε Index και να χρησιμοποιώ το αποτέλεσμα σαν το νέο στοιχείο.

...	1: TABLEGETELEM t "a" _t1 // t["a"]
t.a.b = x;	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.

HY340

A. Σαββίδης

Slide 19 / 38



## Εκφράσεις αποθήκευσης (7/11)

- *Η λύση*
  - Όταν έχω στοιχείο πίνακα, δεν παράγω κώδικα για το συγκεκριμένο στοιχείο, αλλά το καταγράφω ως νέο expression «στοιχείο πίνακα» - *tableitem\_e*, αποθηκεύοντας τόσο το *table* όσο και το *index* expression
  - Όταν χρησιμοποιείται ένα l-value που είναι τύπου «στοιχείο πίνακα»
    - ♦ ως μερική έκφραση σε άλλο κανόνα (π.χ. *lvalue.id* ή *lvalue[expr]*) ή για αναγωγή σε r-value (δηλ. *primary*), τότε παράγουμε μία εντολή TABLEGETELEM
    - ♦ ως το αριστερό τμήμα μίας εκχώρησης (*lvalue=expr*), τότε παράγουμε μία εντολή TABLESETELEM

HY340

A. Σαββίδης

Slide 20 / 38



## Εκφράσεις αποθήκευσης (8/11)

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. Αυτή η δουλειά γίνεται από τη συνάρτηση `emit_iftableitem` που θα δούμε αργότερα.

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

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



## Εκφράσεις αποθήκευσης (9/11)

```
tableitem → lvalue [ expr ]
{
    $lvalue = emit_iftableitem($lvalue);
    $tableitem = newexpr(tableitem_e);
    $tableitem->sym = $lvalue->sym;
    $tableitem e->index = $expr; // The index is the expression.
}

primary → lvalue
{ $primary = emit_iftableitem($lvalue); }
```



## Εκφράσεις αποθήκευσης (10/11)

```
assignexpr → lvalue = expr {
    if $lvalue->type = tableitem_e then {
        emit( // that is: lvalue[index] = expr
            tablesetelem,
            $lvalue,
            $lvalue->index,
            $expr
        );
        // The value of the assignment expression should be gained.
        $assignexpr = emit_iftableitem($lvalue); // Will always emit.
        $assignexpr->type = assignexpr_e;
    }
    else {
        emit( // that is: lvalue = expr
            assign,
            $expr,
            (expr*) 0,
            $lvalue
        );
        $assignexpr = newexpr(assignexpr_e);
        $assignexpr->sym = newtemp();
        emit(assign, $lvalue, (expr*) 0, $assignexpr);
    }
}
```

Χρειάζομαστε νέα κρυφή μεταβλητή καθώς το *l-value* μπορεί να αλλάξει



## Εκφράσεις αποθήκευσης (11/11)

```
expr* newexpr(expr_t t) {
    expr* e = (expr*) malloc(sizeof(expr));
    memset(e, 0, sizeof(expr));
    e->type = t;
    return e;
}

expr* newexpr_conststring(char* s) {
    expr* e = newexpr(conststring_e);
    e->strConst = strdup(s);
    return e;
}

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;
    }
}
```

<code>x = p[a+b];</code>	1: ADD a b _t1 2: TABLEGETELEM p _t1 _t2 3: ASSIGN _t2 x
<code>y = q.a = x;</code>	4: TABLESETELEM q "a" x 5: TABLEGETELEM q "a" _t3 6: ASSIGN _t3 y 7: ASSIGN y _t4





## Περιεχόμενα

- Δηλώσεις συναρτήσεων
- Εκφράσεις αποθήκευσης
- **Βασικές εκφράσεις**
- Επαναχρησιμοποίηση κρυφών μεταβλητών

HY340

A. Σαββίδης

Slide 25 / 38



## Βασικές εκφράσεις (1/9)

- Κλήση συναρτήσεων
- Δημιουργία πινάκων
- Ορισμός συνάρτησης σε παρένθεση
- Μοναδιαίοι τελεστές

HY340

A. Σαββίδης

Slide 26 / 38



## Βασικές εκφράσεις (2/9)

*methodcall* → .. *id* ( *elist* )

```
{
    $methodcall.elist = $elist;
    $methodcall.method = true;
    $methodcall.name = $id.name
}
```

*expr*\* *make\_call*(*lvalue*, *elist*)

```
{
    expr* func = emit_if_tableitem($lvalue);
    for each arg in reversed elist do
        emit(param, arg);
    emit(call, func);
    result = newexpr(var_e);
    result->sym = newtemp();
    emit(getretval, result);
    return result;
}
```

*call* → *call* ( *elist* )

```
{ $call = make_call( $call , $elist ); }
```

*call* → ( *funcdef* ) ( *elist* )

```
{
    func = newexpr(programfunc_e);
    func->sym = $funcdef;
    $call = make_call(func, elist)
}
```

### Κλήση συναρτήσεων (1/2)

<i>x</i> = <i>f</i> ( <i>y</i> ) ( <i>a</i> , <i>b</i> ) ;	1: PARAM <i>y</i> 2: CALL <i>f</i> 3: GETRETVAL <i>t1</i> 4: PARAM <i>b</i> 5: PARAM <i>a</i> 6: CALL <i>t1</i> 7: GETRETVAL <i>t2</i> 8: ASSIGN <i>t2</i> <i>x</i>
--	--

HY340

A. Σαββίδης

Slide 27 / 38



## Βασικές εκφράσεις (3/9)

*call* → *lvalue* *callsuffix*

```
{
    if callsuffix.method then {
        expr* self = lvalue;
        lvalue = emit_if_tableitem(member_item(self, callsuffix.name));
        $callsuffix.elist.add_front(self);
    }
    $call = make_call($lvalue, $callsuffix.elist);
}
```

*callsuffix* → *normcall*

```
{ $callsuffix = $normcall }
```

*callsuffix* → *methodcall*

```
{ $callsuffix = $methodcall }
```

*normcall* → ( *elist* )

```
{
    $normcall.elist = $elist;
    $normcall.method = false;
    $normcall.name = nil
}
```

### Κλήση συναρτήσεων (2/2)

<i>sprite</i> .. <i>move</i> ( <i>dx</i> , <i>dy</i> ) ;	1: TABLEGETELEM <i>sprite</i> "move" <i>t1</i> 2: PARAM <i>dy</i> 3: PARAM <i>dx</i> 4: PARAM <i>sprite</i> 5: CALL <i>t1</i> 6: GETRETVAL <i>t2</i>
--	---

HY340

A. Σαββίδης

Slide 28 / 38



## Βασικές εκφράσεις (4/9)

### Δημιουργία πινάκων (1/2)

tablemake → [ elist ]

```
t = newexpr(newtable_e);
t->sym = newtemp();
emit(tablecreate, t);
i=0;
for each x in $elist do
    emit(tablesetelem, t, newexpr_constnum(i++), x);
$stablemake = t;
```

```
expr* newexpr_constnum(double i) {
    expr* e = newexpr(constnum_e);
    e->numConst = i;
    return e;
}
```

x = [-4, 13, 12.34, a+b];	1: UMINUS 4 _t1 2: ADD a b _t2 3: TABLECREATE _t3 4: TABLESETELEM _t3 0 _t1 5: TABLESETELEM _t3 1 13 6: TABLESETELEM _t3 2 12.34 7: TABLESETELEM _t3 3 _t2 8: ASSIGN _t3 x
---------------------------	---



## Βασικές εκφράσεις (5/9)

### Δημιουργία πινάκων (2/2)

tablemake → [ indexed ]

```
t = newexpr(newtable_e);
t->sym = newtemp();
emit(tablecreate, t);
for each <x,y> in $indexeddo
    emit(tablesetelem, t, x, y);
$stablemake = t;
```

Το γραμματικό σύμβολο *indexed* αντιστοιχεί σε τύπο δεδομένων λίστας από ζευγάρια εκφράσεων.

t = [ {"x": 0}, {"y": a+b}, {-1: 0} 1];	1: ADD a b _t1 2: UMINUS 1 _t2 3: TABLECREATE _t3 4: TABLESETELEM _t3 "x" 0 5: TABLESETELEM _t3 "y" _t1 6: TABLESETELEM _t3 _t2 0 7: ASSIGN _t3 t
---	---



## Βασικές εκφράσεις (6/9)

### Ορισμός συνάρτησης σε παρένθεση

primary → ( funcdef )

```
$primary = newexpr(programfunc_e);
$primary->sym = $funcdef;
```

```
x = (function(a,b){ return a+b; });
```

Πως μπορούμε να υποστηρίξουμε κάτι τέτοιο?  
Με τους κανόνες που ήδη έχουμε παρουσιάσει για call!

```
1: FUNCSTART _f1
2: ADD a b _t1
3: RETURN _t1
4: FUNCEND _f1
5: ASSIGN _f1 x

6: FUNCSTART _f2
7: FUNCEND _f2
8: CALL _f2
9: GETRETVAL _t1
```



## Βασικές εκφράσεις (7/9)

### Μοναδιαίοι τελεστές (1/3)

term → ( expr ) { \$term = \$expr; }

term → primary { \$term = \$expr; }

```
term → - expr
{
    checkuminus($expr);
    $term = newexpr(arithexpr_e);
    $term->sym = newtemp();
    emit(uminus, $expr, $term);
}
```

term → ! expr

```
{
    $term = newexpr(boolexpr_e);
    $term->sym = newtemp();
    emit(not, $expr, $term);
}
```

```
void compererror(char* format, ...);
// Since we may know at compile time a unary
// minus type conflict, we can spot it. You can
// also use negation test for implementation.
void checkuminus(expr* e) {
    if (e->type == constbool_e ||
        e->type == conststring_e ||
        e->type == nil_e ||
        e->type == newtable_e ||
        e->type == programfunc_e ||
        e->type == libraryfunc_e ||
        e->type == boolexpr_e)
        compererror("Illegal expr to unary -");
}
```





## Βασικές εκφράσεις (8/9)

### Μοναδιαίοι τελεστές (2/3)

$term \rightarrow lvalue++$   
{

$\$term = newexpr(var\_e);$

$\$term \rightarrow sym = newtemp();$

if  $\$lvalue \rightarrow type = tableitem\_e$  then {  
   $value = emit\_iftableitem(\$lvalue);$   
   $emit(assign, value, \$term);$   
   $emit(add, value, newexpr\_constnum(1, value);$   
   $emit(tablesetelem, \$lvalue, \$lvalue \rightarrow index, value);$   
}

else {  
   $emit(assign, \$lvalue, \$term);$   
   $emit(add, \$lvalue, newexpr\_constnum(1, \$lvalue);$   
}

}

$a = x--;$	1: ASSIGN $x\_t0$ 2: SUB $x\_1\ x$ 3: ASSIGN $t0\ a$
$a = t.x++;$	4: TABLEGETELEM $t\ "x"\_t2$ 5: ASSIGN $t2\ t1$ 3: ADD $t2\ 1\ t2$ 4: TABLESETELEM $t\ "x"\_t2$ 5: ASSIGN $t1\ a$

HY340

A. Σαββίδης

Slide 33 / 38



## Βασικές εκφράσεις (9/9)

### Μοναδιαίοι τελεστές (3/3)

$term \rightarrow ++lvalue$   
{

if  $\$lvalue \rightarrow type = tableitem\_e$  then {  
   $\$term = emit\_iftableitem(\$lvalue);$   
   $emit(add, \$term, newexpr\_constnum(1, \$term);$   
   $emit(tablesetelem, \$lvalue, \$lvalue \rightarrow index, \$term);$   
}

else {  
   $emit(add, \$lvalue, newexpr\_constnum(1, \$lvalue);$   
   $\$term = newexpr(arithexpr\_e);$   
   $\$term \rightarrow sym = newtemp();$   
   $emit(assign, \$lvalue, \$term);$   
}

$term \rightarrow --lvalue$  { Συμπληρώστε το! }  
 $term \rightarrow lvalue--$  { Συμπληρώστε το! }

$++x;$	1: ADD $x\_1\ x$ 2: ASSIGN $x\_t0$
$a = ++t.x;$	2: TABLEGETELEM $t\ "x"\_t0$ 3: ADD $t0\ 1\ t0$ 4: TABLESETELEM $t\ "x"\_t0$ 5: ASSIGN $t0\ a$
$a = --t.x.y;$	6: TABLEGETELEM $t\ "x"\_t1$ 7: TABLEGETELEM $t1\ "y"\_t2$ 8: SUB $t2\ 1\ t2$ 9: TABLESETELEM $t1\ "y"\_t2$ 10: ASSIGN $t2\ a$

Χρειαζόμαστε νέα κρυφή μεταβλητή καθώς το l-value μπορεί να αλλάξει μετά.

HY340

A. Σαββίδης

Slide 34 / 38



## Περιεχόμενα

- Δηλώσεις συναρτήσεων
- Εκφράσεις αποθήκευσης
- Βασικές εκφράσεις
- Επαναχρησιμοποίηση κρυφών μεταβλητών

HY340

A. Σαββίδης

Slide 35 / 38



## Επαναχρησιμοποίηση κρυφών μεταβλητών (1/3)

- Έχουμε ήδη αναφέρει την τακτική επαναφοράς του μετρητή ονομασίας κρυφών μεταβλητών στην τιμή μηδέν μετά από κάθε *stmt*
- Ωστόσο υπάρχουν και επιπλέον περιθώρια βελτίωσης, εάν παρατηρήσει κανείς ότι μετά τη χρησιμοποίηση μίας κρυφής μεταβλητής σε μία εντολή ενδιάμεσου κώδικα ως όρισμα (αφού την πρώτη φορά είναι πάντα το αποτέλεσμα), αυτή ουσιαστικά καθίσταται «διαθέσιμη».
- Αυτό σημαίνει εμβόλιμο κώδικα στους σημασιολογικούς κανόνες μετάφρασης για επαναχρησιμοποίηση ανά περίπτωση

HY340

A. Σαββίδης

Slide 36 / 38



## Επαναχρησιμοποίηση κρυφών μεταβλητών (2/3)

- Η δυνατότητα επαναχρησιμοποίησης φαίνεται και στο παρακάτω παράδειγμα
  - Όποτε έχω χρήση κρυφών μεταβλητών ως r-values, τότε μπορώ να χρησιμοποιήσω αυτή τη μεταβλητή και πάλι για προσωρινή αποθήκευση
- Η αναγνώριση των κρυφών μεταβλητών είναι εύκολη βάσει του ονόματος

```
unsigned int istempname (char* s)
{ return *s == '_'; }

unsigned int istempexpr (expr* e) {
    return e->sym &&
           e->sym->type == vars_s &&
           istempname(e->sym->name);
}
```

**a = x+y+z;**  
 ((x+y)+z)  
 με αριστερή προσηταιριστικότητα

1: ADD x y \_t1  
 2: ADD \_t1 z \_t2  
 3: ASSGIN \_t2 a

Στο 1: το \_t1 είναι αποτέλεσμα, ενώ στο 2: είναι όρισμα, άρα μπορεί να επαναχρησιμοποιηθεί

1: ADD x y \_t1  
 2: ADD \_t1 z \_t1  
 3: ASSIGN \_t1 a



## Επαναχρησιμοποίηση κρυφών μεταβλητών (3/3)

### Σημεία επαναχρησιμοποίησης (προαιρετικό)

- Κλήση συνάρτησης. Από το lvalue, call, ή κάποιο από τα elist
- Κατασκευή πίνακα. Από κάποιο από τα elist ή από κάποιο <index, expr>
- Μοναδιαίο μείον. Από το expr.
- Λογική άρνηση. Από το expr.
- Αριθμητικές εκφράσεις (δυναδικοί τελεστές). Από οποιαδήποτε εκ των δύο expr.
- Λογικές εκφράσεις (δυναδικοί τελεστές). Από οποιαδήποτε εκ των δύο expr.

### Η επέκταση ενός σημασιολογικού κανόνα (unary minus) για επαναχρησιμοποίηση κρυφής μεταβλητής

term  $\rightarrow$  - expr

```
{
    checkuminus($expr);
    $term = newexpr(arithexpr_e);
    $term->sym = istempexpr($expr)? $expr->sym : newtemp();
    emit(uminus, $expr, $term);
}
```

Μόνο αυτό προσθέσαμε