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 14η
ΠΑΡΑΓΩΓΗ ΤΕΛΙΚΟΥ ΚΩΔΙΚΑ – *μία διάλεξη*

Περιεχόμενα

Οι ευκολίες της μηχανής alpha (1/4)

- Θα φανεί σύντομα ότι η παραγωγή τελικού κώδικα για τη μηχανή alpha είναι ιδιαίτερα εύκολη υπόθεση.
- Αυτό οφείλεται στο γεγονός ότι, καθώς πρόκειται για εικονική μηχανή, το ρεπερτόριο των εντολών της έχει σχεδιαστεί ώστε να συνδυάζει
  - μικρότερο τελικό κώδικα
  - γρήγορο τελικό κώδικα
  - εύκολη μετάβαση από τον ενδιάμεσο στον τελικό κώδικα
- Αυτό δεν είναι εν γένει εφικτό σε πραγματικές μηχανές. Π.χ. στους επεξεργαστές RISC έμφαση δίνεται περισσότερο στη δημιουργία ενός ρεπερτορίου από ταχύτατες γενικές εντολές, παρά σε σύνθετες ειδικές εντολές
  - Ενώ συνήθως το πάντρεμα και των δύο χαρακτηριστικών δεν οδηγεί σε βέλτιστα αποτελέσματα
  - Οι μηχανές RISC χαρακτηρίζονται από απλότητα εντολών, που σημαίνει μεν μεγαλύτερο τελικό κώδικα, αλλά ωστόσο και πολύ μεγάλη ταχύτητα εκτέλεσης

Οι ευκολίες της μηχανής alpha (2/4)

- Η ιδιαιτερότητα αυτή στις πραγματικές μηχανές οφείλεται στις μεγάλες αποκλίσεις ταχύτητας ανάλογα με το είδος της μνήμης που χρησιμοποιείται
  - *Main memory* ~20-100 κύκλοι
  - *Cache memory* ~3 κύκλοι
  - *Registers* 1 κύκλος
- Η προτίμηση μίας εντολής με πρόσβαση σε καταχωρητές σε σχέση με μία που έχει πρόσβαση στην κεντρική μνήμη είναι προφανής
  - Επιπλέον σε επεξεργαστές RISC οι περισσότερες εντολές πλην των load / store εμπλέκουν αποκλειστικά καταχωρητές
  - Αυτό σημαίνει ότι ο τελικός κώδικας πρέπει να αναγκαστικά να χρησιμοποιεί καταχωρητές και όχι θέσεις μνήμης
  - Όμως εν γένει οι διαθέσιμοι καταχωρητές είναι λίγοι, αρκετές φορές λιγότεροι από τις συνολικές μεταβλητές σε κάποιο τμήμα κώδικα
  - Επομένως απαιτείται βελτιστοποίηση χρήσης, γεγονός που σημαίνει σημαντικά πιο πολύπλοκη παραγωγή τελικού κώδικα

Οι ευκολίες της μηχανής alpha (3/4)

- Γιατί αυτή η περιπλοκή δεν εμφανίζεται στην εικονική μηχανή *alpha*;
  - Οι καταχωρητές είναι απλές μεταβλητές, ενώ η μνήμη στοίβας είναι απλώς «κελιά» σε κάποιον δυναμικό πίνακα του προγράμματος εξομοίωσης
  - Το κόστος πρόσβασης σε έναν καταχωρητή στην πράξη ισοδυναμεί με το κόστος πρόσβασης σε μία μεταβλητή του προγράμματος εξομοίωσης
  - Ενώ το κόστος πρόσβασης σε μία θέση μνήμης (στοίβα) ισοδυναμεί παρόμοια με το κόστος πρόσβασης σε στοιχείο πίνακα (της στοίβας)
  - Πρακτικά, οι σχετικές διαφορές ταχύτητας πρόσβασης σε λογισμική εξομοίωση δεν είναι σημαντικές ώστε να αξίζει να ασχοληθούμε με βελτιστοποίηση χρήσης καταχωρητών
  - Για το λόγο αυτό οι προδιαγραφές της μηχανής *alpha* ορίζουν μόνο δύο βοηθητικούς καταχωρητές, ενώ και η ίδια η έννοια του καταχωρητή είναι προφανώς «μεταφορική» παρά πραγματική σε μία εικονική μηχανή.
  - Επιπλέον, όλες οι εντολές της γλώσσας μηχανής *alpha* εμπλέκουν σχετικές διευθύνσεις μνήμης, όπου μία σχετική διεύθυνση είναι της μορφής |*type*|*offset*|, όπου και τα δύο είναι σταθερές αριθμητικές τιμές, δηλ. δεν υφίσταται έμμεση αναφορά – *no indirect addressing*

Οι ευκολίες της μηχανής alpha (4/4)

- Στη σχεδίαση της μηχανής alpha έχει γίνει αυτοματισμός κοινών ακολουθιών εντολών για την κλήση συναρτήσεων
  - push argument
  - function enter
  - function exit
- Θα μπορούσε να μη γίνει μία τέτοια επιλογή, αλλά τότε θα έπρεπε να είχαμε εντολές οι οποίες θα επέτρεπαν καταχωρητές να είναι ορίσματα, και όχι απλώς διευθύνσεις μνήμης
- Υπάρχει υποστήριξη χρήσης θέσεων μνήμης ως ορίσματα σε όλες τις εντολές της μηχανής
- Ορίζεται έναν σταθερό και κοινό μέγεθος μνήμης για όλους του τύπους δεδομένων
- Υπάρχει απαγόρευση χρήσης των τυπικών ορισμάτων και τοπικών μεταβλητών μίας συνάρτησης από περιεχόμενες συναρτήσεις
  - Αυτό μας γλιτώνει από την υλοποίηση τεχνικής πρόσβασης σε μεταβλητές περιέχουσας συνάρτησης – *access links*
- Υποστηρίζεται η διαχείριση συσχετιστικών δυναμικών πινάκων με ειδικό ρεπερτόριο εντολών
  - Αλλιώς θα χρειαζόμασταν κάποιο ειδικό runtime library

Περιεχόμενα

Από εκφράσεις σε ορίσματα εντολών (1/4)

- Ο ενδιάμεσος κώδικας αποτελείται από εντολές οι οποίες εμπλέκουν τύπους εκφράσεων ***expr****
- Κατά την μετατροπή σε τελικό κώδικα, πέρα από την αντιστοίχηση εντολών ενδιάμεσου κώδικα σε εντολές μηχανής, είναι απαραίτητο να μετατραπούν οι εκφράσεις σε κατάλληλα ορίσματα των εντολών μηχανής
  - Σύμφωνα με την τυπολογία που έχουμε περιγράψει δηλαδή
    | type bits | integer value|
- Κάτι τέτοιο μπορεί να αυτοματοποιηθεί μέσω μίας συνάρτησης η οποία λαμβάνει ***expr**** και θέτει κατάλληλα ένα ***vmarg****
- Η μετατροπή είναι εύκολη καθώς υπάρχει ευκολία στη χρήση διευθύνσεων μνήμης της μηχανής *alpha*, ενώ προς το παρόν δεν μας απασχολεί κάποιο θέμα βελτιστοποίησης
  - π.χ. όπως η ελαχιστοποίηση κρυφών μεταβλητών, ελαχιστοποίηση εντολών και υπολογισμών

Από εκφράσεις σε ορίσματα εντολών (2/4)

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

Πίνακες σταθερών τιμών και συναρτήσεων

input *output*

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

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

            assert(e->sym);
            arg->val = e->sym->offset;

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

        /* Constants */

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

Αυτά ξέρουμε επιπλέον ότι θα είναι και κρυφές μεταβλητές

Από εκφράσεις σε ορίσματα εντολών (3/4)

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

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

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

        /* Functions */
        case programfunc_e: {
            arg->type = userfunc_a;
            arg->val = e->sym->taddress;
            /* or alternatively ...*/
            arg->val = userfuncs_newfunc( e->sym);

            break;
        }

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

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

Για τις συναρτήσεις μπορούμε να έχουμε ως τιμή του operand απευθείας τη διεύθυνση της συνάρτησης στον τελικό κώδικα ή ένα index στον πίνακα όπου καταχωρούμε κάθε συνάρτηση του χρήστη.

Από εκφράσεις σε ορίσματα εντολών (4/4)

```
/* Helper functions to produce common arguments for
   generated instructions, like 1, 0, "true", "false"
   and funciton return values.
*/
void make_numberopearand (vmarg* arg, double val) {
    arg->val = consts_newnumber(val);
    arg->type = number_a;
}

void make_booloperand (vmarg* arg, unsigned val) {
    arg->val = val;
    arg->type = bool_a;
}

void make_retvaloperand (vmarg* arg)
    { arg->type = retval_a; }
```

•Παρατηρούμε ότι το offset των operands που είναι μεταβλητές δεν είναι «φυσικό» (στη μνήμη) αλλά «λογικό» (δηλ. από αυτό προκύπτει το memory offset), καθώς είναι ουσιαστικά ο σειριακός αριθμός κατά τη μεταγλώττιση.

•Η χρήση των λογικών offsets για τον προσδιορισμό της πραγματικής διεύθυνσης είναι ευθύνη του περιβάλλοντος εκτέλεσης.

•Επειδή γενικά μπορεί η τεχνοτροπία υλοποίησης να διαφέρει από εξομοιωτή σε εξομοιωτή, δεν θέλουμε να επηρεάζουμε τον τελικό κώδικα από τέτοιες λεπτομέρειες. Έτσι θα μπορεί ο ίδιος τελικός κώδικας να εκτελείται από διαφορετικές υλοποιήσεις της ίδιας μηχανής.

•Έχουμε ήδη παρουσιάσει τον τρόπο που η μηχανή alpha υπολογίζει τη διεύθυνση μνήμης βάση type και offset.

Περιεχόμενα

- Οι ευκολίες της μηχανής alpha
- Από εκφράσεις σε ορίσματα εντολών
- *Σχήματα παραγωγής κώδικα*

Σχήματα παραγωγής κώδικα (1/15)

- Για την παραγωγή του τελικού κώδικα χρησιμοποιούμε ανάλογη τεχνική δυναμικού πίνακα από εντολές, όπως και στον ενδιάμεσο κώδικα. Ορίζουμε την εντολή *emit*, με τη μόνη διαφορά ότι έχει ένα μοναδικό όρισμα τύπου *instruction**.
- Επειδή αναμένεται να έχουμε εντολές jump ενδιάμεσου κώδικα σε μεταγενέστερες εντολές, μπορούμε να συμπληρώσουμε σωστά τον αριθμό της εντολής τελικού κώδικα μόνο όταν έχει γίνει και η παραγωγή τελικού κώδικα για την εντολή προορισμού. Μία πολύ απλή λύση, αλλά όχι βέλτιστη σε μνήμη, ακολουθεί παρακάτω:
  - Εισάγουμε το πεδίο *taddress* στις εντολές ενδιάμεσου κώδικα, το οποίο συμπληρώνεται κατά την παραγωγή τελικού κώδικα και προσδίδει σε κάθε εντολή ενδιάμεσου κώδικα τον αριθμο της αντίστοιχης 1ης εντολής τελικού κώδικα που παράγεται.
  - Ορίζουμε ένα record *incomplete_jump* με πεδία τον αριθμό εντολής τελικού κώδικα *instrNo* του jump και τον αριθμό της εντολής ενδιάμεσου *iaddress* που αντιστοιχεί στον προορισμό ως εντολή ενδιάμεσου κώδικα.

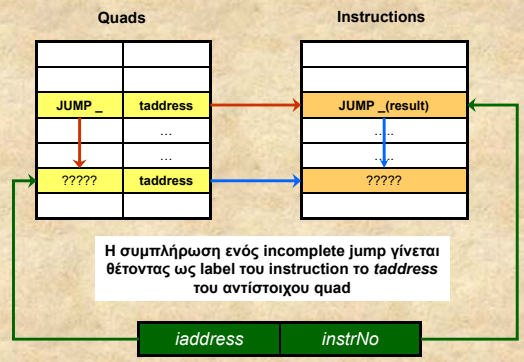
Σχήματα παραγωγής κώδικα (2/15)

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

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

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

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

Για τις εντολές jump, χρησιμοποιούμε το πεδίο *result* των εντολών τελικού κώδικα για την αποθήκευση του αριθμού της εντολής προορισμού.

Σχήματα παραγωγής κώδικα (3/15)



Η συμπλήρωση ενός incomplete jump γίνεται θέτοντας ως label του instruction το *taddress* του αντίστοιχου quad

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

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

Σχήματα παραγωγής κώδικα (4/15)

ΑΡΙΘΜΗΤΙΚΕΣ ΕΚΦΡΑΣΕΙΣ

```
generate (op, quad) {
        instruction t;
        t.opcode = op;
        make_operand(quad.arg1, &t.arg1);
        make_operand(quad.arg2, &t.arg2);
        make_operand(quad.result, &t.result);
        quad.taddress = nextinstructionlabel();
        emit(t);
}
generate_ADD (quad)     { generate(add, quad); }
generate_SUB (quad)     { generate(sub, quad); }
generate_MUL (quad)     { generate(mul, quad); }
generate_DIV (quad)     { generate(div, quad); }
generate_MOD (quad)     { generate(mod, quad); }
```

Όπως είναι προφανές, η παραγωγή τελικού κώδικα για τις αριθμητικές εντολές είναι πολύ απλή, καθώς ο κύριος φόρτος που είναι ο προσδιορισμός των ορισμάτων αυτοματοποιείται με την ***make_operand***. Η εντολή UMINUS μετατρέπεται σε πολλαπλασιασμό με το –1 και η υλοποίησή της είναι τετριμμένη.

Σχήματα παραγωγής κώδικα (5/15)

ΣΥΣΧΕΤΙΣΤΙΚΟΙ ΠΙΝΑΚΕΣ

```
generate_NEWTABLE (quad)        { generate(newtable, quad); }
generate_TABLEGETELM (quad)     { generate(tablegetelem, quad); }
generate_TABLESETELEM (quad)    { generate(tablesetelem, quad); }
generate_ASSIGN (quad)          { generate(assign, quad); }
generate_NOP ()                 { instruction t; t.opcode=nop; emit(t); }
```

Εξίσου απλή η παραγωγή τελικού κώδικα και για τις εντολές πινάκων, την εκχώρηση και φυσικά το ***nop***.

```
generate_relational (op, quad) {
    instruction t;
    t.opcode = op;
    make_operand(quad.arg1, &t.arg1);
    make_operand(quad.arg2, &t.arg2);

    t.result.type = label_a;
    if quad.label jump target < currprocessedquad() then
            t.result.value = quads[quad.label].taddress;
    else
            add_incomplete_jump(nextinstructionlabel(), quad.label);
    quad.taddress = nextinstructionlabel();
    emit(t);
}
```

Στην παραγωγή των εντολών jump με συσχετιστικούς τελεστές ενδέχεται να χρειαστεί η εισαγωγή ενός incomplete jump. Κάτι τέτοιο συμβαίνει μόνο όταν το jump γίνεται σε εντολή ενδιάμεσου κώδικα η οποία δεν έχει ακόμη επεξεργαστεί (δηλ. δεν έχουμε παράγει τελικό κώδικα).

Σχήματα παραγωγής κώδικα (6/15)

ΣΥΣΧΕΤΙΣΤΙΚΕΣ ΕΚΦΡΑΣΕΙΣ

```
generate_JUMP (quad)            { generate_relational(jump, quad); }
generate_IF_EQ (quad)           { generate_relational(jeq, quad); }
generate_IF_NOTEQ(quad)         { generate_relational(jne, quad); }
generate_IF_GREATER (quad)      { generate_relational(jgt, quad); }
generate_IF_GREATEREQ(quad)     { generate_relational(jge, quad); }
generate_IF_LESS (quad)         { generate_relational(jlt, quad); }
generate_IF_LESSEQ (quad)       { generate_relational(jle, quad); }
```

Χρησιμοποιώντας την ***generate_ralational*** η παραγωγή κώδικα γίνεται ιδιαίτερα εύκολη.

- Η παραγωγή τελικού κώδικα για τις λογικές εκφράσεις στην περίπτωση που ενσωματώνουμε στην εικονική μηχανή τις αντίστοιχες εντολές είναι τετριμμένη. Αλλιώς πρέπει να παράγουμε κώδικα αποτίμησης των λογικών εκφράσεων.
- Εάν υλοποιήσουμε την μέθοδο μερικής αποτίμησης λογικών εκφράσεων τότε σε κανένα σημείο του παραγόμενου ενδιάμεσου κώδικα δεν θα υπάρχουν λογικοί τελεστές. Ξεκινάμε με τον τελεστή ***NOT***.

ενδιάμεσος κώδικας

…			
NOT	arg1	result	*taddress*
…			

τελικός κώδικας

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

+2 +3

ΛΟΓΙΚΕΣ ΕΚΦΡΑΣΕΙΣ

Σχήματα παραγωγής κώδικα (7/15)

```
generate_NOT (quad) {

    quad.taddress = nextinstructionlabel();
    instruction t;

    t.opcode = jeq;
    make_operand(quad.arg1, &t.arg1);
    make_booloperand(&t.arg2, false);
    t.result.type = label_a;
    t.result.value  = nextinstructionlabel()+3;
    emit(t);

    t.opcode = assign;
    make_booloperand(&t.arg1, false);
    reset_operand(&t.arg2);
    make_operand(quad.result, &t.result);
    emit(t);
    …
```

```
    …συνέχεια
    t.opcode = jump;
    reset_operand (&t.arg1);
    reset_operand(&t.arg2);
    t.result.type = label_a;
    t.result.value  = nextinstructionlabel()+2;
    emit(t);

    t.opcode = assign;
    make_booloperand(&t.arg1, true);
    reset_operand(&t.arg2);
    make_operand(quad.result, &t.result);
    emit(t);
}
```

ΛΟΓΙΚΕΣ ΕΚΦΡΑΣΕΙΣ

Εάν αποφασίσουμε να ενσωματώσουμε τις εντολές λογικών εκφράσεων στην εικονική μηχανή, πετυχαίνουμε μικρότερο κώδικα αλλά πιο πολύπλοκη εικονική μηχανή. Δεν συνηθίζεται η υποστήριξη τέτοιων εντολών σε εικονικές μηχανές.

Σχήματα παραγωγής κώδικα (8/15)

ενδιάμεσος κώδικας

…				
OR	arg1	arg2	result	taddress
…				

τελικός κώδικας

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

+2

ενδιάμεσος κώδικας

…				
AND	arg1	arg2	result	taddress
…				

τελικός κώδικας

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

+2

ΛΟΓΙΚΕΣ ΕΚΦΡΑΣΕΙΣ

Σχήματα παραγωγής κώδικα (9/15)

```
generate_OR (quad) {

    quad.taddress = nextinstructionlabel();
    instruction t;

    t.opcode = jeq;
    make_operand(quad.arg1, &t.arg1);
    make_booloperand(&t.arg2, true);
    t.result.type = label_a;
    t.result.value  = nextinstructionlabel()+4;
    emit(t);

    make_operand(quad.arg2, &t.arg1);
    t.result.value  = nextinstructionlabel()+3;
    emit(t);
    …
```

Η παραγωγή κώδικα για την εντολή ενδιάμεσου κώδικα AND είναι παρόμοια.

ΛΟΓΙΚΕΣ ΕΚΦΡΑΣΕΙΣ

```
    …συνέχεια
    t.opcode = assign;
    make_booloperand(&t.arg1, false);
    reset_operand(&t.arg2);
    make_operand(quad.result, &t.result);
    emit(t);

    t.opcode = jump;
    reset_operand (&t.arg1);
    reset_operand(&t.arg2);
    t.result.type = label_a;
    t.result.value  = nextinstructionlabel()+2;
    emit(t);

    t.opcode = assign;
    make_booloperand(&t.arg1, true);
    reset_operand(&t.arg2);
    make_operand(quad.result, &t.result);
    emit(t);
}
```


Σχήματα παραγωγής κώδικα (10/15)

```
generate_PARAM(quad) {
    quad.taddress = nextinstructionlabel();
    instruction t;
    t.opcode = pusharg;
    make_operand(quad.arg1, &t.arg1);
    emit(t);
}

generate_CALL(quad) {
    quad.taddress = nextinstructionlabel();
    instruction t;
    t.opcode = callfunc;
    make_operand(quad.arg1, &t.arg1);
    emit(t);
}

generate_GETRETVAL(quad) {
    quad.taddress = nextinstructionlabel();
    instruction t;
    t.opcode = assign;
    make_operand(quad.result, &t.result);
    make_retvaloperand(&t.arg1);
    emit(t);
}
```

ΚΛΗΣΗ ΣΥΝΑΡΤΗΣΕΩΝ

- Η παραγωγή κώδικα για τις εντολές κλήσης συνάρτησης είναι επίσης πολύ απλή.
- Σημειώνουμε ότι η *pusharg* είναι σύνθετη εντολή καθώς πρέπει να αποθηκεύει ένα πραγματικό όρισμα στη στοίβα και να αναπροσαρμόζει τους δείκτες στοίβας κατάλληλα.
- Βλέπουμε πως η συγκομιδή επιστρεφόμενου αποτελέσματος συνίσταται σε μία απλή εκχώρηση από τον ειδικό καταχωρητή που περιέχει την επιστρεφόμενη τιμή.
- Παρατηρούμε ότι δεν τροφοδοτούμε τον αριθμό των πραγματικών ορισμάτων στην εντολή κλήσης. Θα μπορούσαμε, απλώς στην προκείμενη τακτική θεωρούμε ότι η υλοποίησης της εντολής *pusharg* περιέχει και τη λογική καταμέτρησης των πραγματικών ορισμάτων.

Σχήματα παραγωγής κώδικα (11/15)

ενδιάμεσος κώδικας

…		
FUNCSTART	result	taddress
…		
RETURN	result	taddress
…		
FUNCEND	result	taddress

τελικός κώδικας

…		
enterfunc		
…	…	
assign	retval	result
jump	_	
…		
exitfunc		

ΚΩΔΙΚΑΣ ΣΥΝΑΡΤΗΣΕΩΝ

- Η εντολή return απαιτεί δύο ενέργειες: (1) την εκχώρηση του αποτελέσματος στον *retval* καταχωρητή, και (2) την άμεση μετάβαση στο τέλος της συνάρτησης.
- Καθώς δεν γνωρίζουμε το τέλος της συνάρτησης παρά μόνο όταν γίνει παραγωγή του FUNCEND, κάθε τέτοιο JUMP θα μπαίνει σε μία λίστα και θα συμπληρώνεται στο τέλος όταν παράγεται κώδικας για το FUNCEND.
- Όμως τη στιγμή που παράγουμε τελικό κώδικα για μία εντολή RETURN δεν ξέρουμε τη συνάρτηση στην οποία ανήκει. Επειδή υποστηρίζουμε συναρτήσεις μέσα σε άλλες, χρειαζόμαστε μία στοίβα, έστω *funcstack*, την οποία τη διαχειριζόμαστε ως εξής:
  - Στο FUNCSTART κάνουμε push στη στοίβα
  - Στο FUNCEND κάνουμε pop
  - Στο return βάζουμε στον αριθμό εντολής τελικού κώδικα για το incomplete jump στη λίστα του top function αυτής της στοίβας

Σχήματα παραγωγής κώδικα (12/15)

ΚΩΔΙΚΑΣ ΣΥΝΑΡΤΗΣΕΩΝ

```
generate_FUNCSTART(quad) {
    f = quad.result->sym;
    f->taddress = nextinstructionlabel();
    quad.taddress = nextinstructionlabel();

    userfunctions.add(f->id, f->taddress, f->totallocals);
    push(funcstack, f);

    instruction t;
    t.opcode = enterfunc;
    make_operand(quad.result, &t.result);
    emit(t);
}
```

```
generate_RETURN(quad) {
    quad.taddress = nextinstructionlabel();
    instruction t;
    t.opcode = assign;
    make_retvaloperand(&t.result);
    make_operand(quad.arg1, &t.arg1);
    emit(t);

    f = top(funcstack);
    append(f.returnList, nextinstructionlabel());

    t.opcode = jump;
    reset_operand (&t.arg1);
    reset_operand(&t.arg2);
    t.result.type = label_a;
    emit(t);
}
```

f
…
….
….
…..

Κάθε συνάρτηση σε αυτή τη στοίβα περιέχεται στον κώδικα μέσα στο block της συνάρτησης που βρίσκεται από κάτω. Εάν είναι στον πάτο της στοίβας, τότε είναι σε καθολική εμβέλεια.

Σχήματα παραγωγής κώδικα (13/15)

ΚΩΔΙΚΑΣ ΣΥΝΑΡΤΗΣΕΩΝ

```
generate_FUNCEND(quad) {
    f=pop(funcstack);
    backpatch(f.returnList, nextinstructionlabel());

    quad.taddress = nextinstructionlabel();
    instruction t;
    t.opcode = exitfunc;
    make_operand(quad.result, &t.result);
    emit(t);
}
```

- Η υλοποίηση των παραπάνω γίνεται με δύο τρόπους. Ο προφανής είναι ότι απαιτείται η εισαγωγή ενός επιπλέον member για τις συναρτήσεις τύπου list για τους αριθμούς των incomplete return statements.
- Ωστόσο, εάν προσέξετε θα δείτε ότι πουθενά στη συμπλήρωση των return jumps δεν χρησιμοποιείται πληροφορία πλέον της λίστας με τους αριθμούς εντολών που είναι return jumps.
- Άρα μπορούμε να έχουμε απλώς μία στοίβα με λίστες από αριθμούς εντολών χωρίς να πειράξουμε τον symbol table για τις συναρτήσεις.

- Η περίπτωση αυτή δεν μοιάζει με τα break και continue statements; θα μπορούσαμε εναλλακτικά να λύσουμε το πρόβλημα απευθείας στην παραγωγή ενδιάμεσου κώδικα; η απάντηση είναι ΝΑΙ, ως εξής:
  - Προσθέτουμε στο *stmt* μη τερματικό σύμβολο ένα *returnList*.
  - Παράγουμε στον ενδιάμεσο κώδικα το jump statement με κενό target label
  - Κάνουμε *backpatch* το return list στην παραγωγή ενδιάμεσου κώδικα για το τέλος συνάρτησης.
- Προτιμάμε την δεύτερη λύση (στον ενδιάμεσο κώδικα) καθώς μπορεί ο παραγόμενος κώδικας να περάσει από την φάση βελτιστοποίησης, κάτι που δεν συμβαίνει εάν προσθέσουμε επιπλέον εντολές στη φάση παραγωγής τελικού κώδικα.

Σχήματα παραγωγής κώδικα (14/15)

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

Με την υλοποίηση των generator συναρτήσεων, του πίνακα εντολών (όπως και στον ενδιάμεσο κώδικα) και των πινάκων σταθερών τιμών, η φάση παραγωγής τελικού κώδικα γίνεται κάτι μεταξύ 0.7 – 1 KLOC (*…μη το λάβετε και σαν κανόνα*).

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

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

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


Σχήματα παραγωγής κώδικα (15/15)

- Η ενσωμάτωση πληροφορίας για την γραμμή πηγαίου κώδικα στην οποία αντιστοιχεί κάθε εντολή γίνεται συνήθως με τη βοήθεια ενός ειδικού πίνακα αντιστοίχισης
  - Μεγέθους ίσου με τον αριθμό των εντολών
  - Με κάθε στοιχείο να είναι ο αριθμός της αντίστοιχης γραμμής πηγαίου κώδικα
- Ο πίνακας αυτός συμπεριλαμβάνεται στον τελικό κώδικα όπως και οι πίνακες σταθερών τιμών
- Εναλλακτικά μπορεί να υιοθετηθεί η αποθήκευση της γραμμής πηγαίου κώδικα ως πεδίο της εντολής
- Ο κανόνας με τον οποίο μία εντολή λαμβάνει αριθμό γραμμής κώδικα είναι απλός
  - Λαμβάνει τον αριθμό γραμμής της εντολής ενδιάμεσου κώδικα λόγω της οποίας παράγεται εάν είναι η πρώτη εντολή που παράγεται με τον τρόπο αυτό, αλλιώς λαμβάνει τον αριθμό 0.
- Στην παραγωγή ενδιάμεσου κώδικα δίνουμε αριθμό γραμμής μόνο στο πρώτο quad ενός statement, ενώ τα υπόλοιπα παίρνουν την τιμή 0.
- Η εικονική μηχανή «αλλάζει» πληροφορία γραμμής κώδικα μόνο εάν αυτή δεν είναι 0.