

Compilers II

INSTRUCTION SELECTION

ΕΠΙΜΕΛΕΙΑ ΔΙΑΦΑΝΕΙΩΝ
ΑΛΕΞΑΝΔΡΟΣ ΠΛΕΣΣΙΑΣ

Μια μικρή εισαγωγή στους Μεταγλωττιστές (1/3)

Ένας μεταγλωττιστής είναι ένα πρόγραμμα που διαβάζει ένα πρόγραμμα γραμμένο σε μία γλώσσα και το μετατρέπει σε ένα πρόγραμμα γραμμένο σε άλλη γλώσσα.

Ο πηγαίος κώδικας είναι τυπικά σε μία γλώσσα υψηλού επιπέδου (π. χ. Pascal, C, C ++, Java, Perl, C #, κ.λπ.). Ο εκτελέσιμος κώδικας μπορεί να είναι μία σειρά εντολών μηχανής που μπορούν να εκτελεστούν από την CPU άμεσα ή μπορεί να είναι μια ενδιάμεση αναπαράσταση που ερμηνεύεται από μια εικονική μηχανή (π. χ. Java byte code).

Εν ολίγοις, ο μεταγλωττιστής μετατρέπει ένα πρόγραμμα από μια μορφή αναγνώσιμη από τον άνθρωπο(human-readable) σε μια μορφή αναγνώσιμη από την μηχανή(machine-readable).

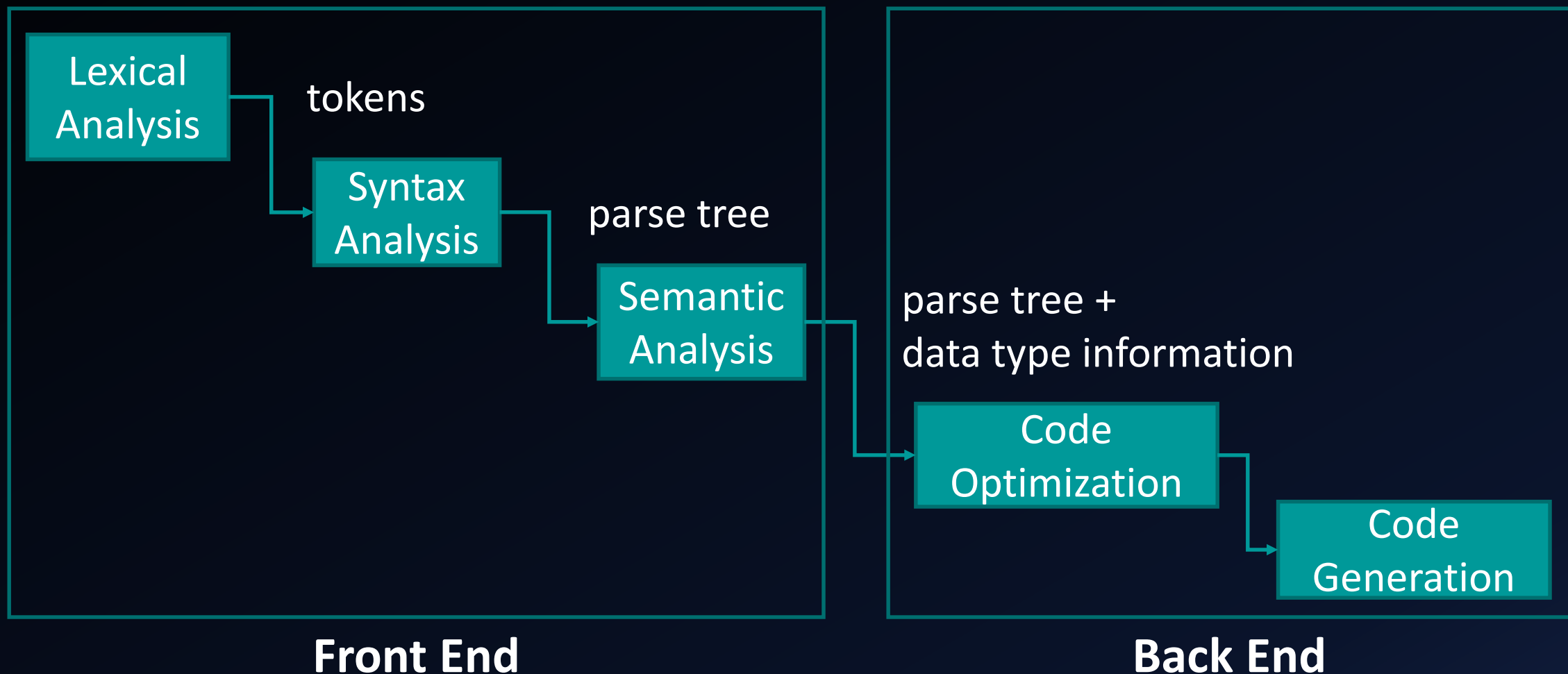
Μια μικρή εισαγωγή στους Μεταγλωττιστές (2/3)

Τώρα, το πώς λειτουργεί ένας μεταγλωττιστής? Είναι πραγματικά περίπλοκος. Θα προσπαθήσω όμως να περιγράψω εν συντομία τα κύρια στάδια της μεταγλώττισης (αλλά αυτό θα είναι μια πολύ βιαστική ανασκόπηση).

- **Lexing** - να σπάσει το κείμενο του προγράμματος σε “tokens”. Τα tokens είναι οι “λέξεις” της γλώσσας προγραμματισμού, όπως αναγνωριστικά (keywords, variable names, function names, κτλ.) ή των τελεστών (=, *, & κτλ.).
- **Parsing** - μετατρέπουν την ακολουθία των tokens σε ένα parse tree, το οποίο είναι μια δομή δεδομένων που αντιπροσωπεύει διάφορες γλωσσικές δομές όπως: type declarations, variable declarations, function definitions, loops, expressions, κτλ.
- **Βελτιστοποίηση** - αξιολογεί σταθερές εκφράσεις, βελτιστοποιεί αχρησιμοποίητες μεταβλητές ή απρόσιτο (unreachable) κώδικα, ξετυλίγει βρόχους (unroll loops) ανν είναι δυνατόν, κτλ.
- ***Μετάφραση** του συντακτικού δένδρου σε εντολές μηχανής (ή JVM byte code).

Και πάλι, τονίζω ότι αυτή είναι μια πολύ σύντομη περιγραφή. Οι σύγχρονη μεταγλωττιστές είναι πολύ έξυπνοι και κατά συνέπεια πολύ περίπλοκη.

Μια μικρή εισαγωγή στους Μεταγλωττιστές (3/3)





BACK END

CODE GENERATION

Τι είναι το Code Generation? (1/2)

Το Back End του compiler πρέπει να **λύσει τρία προβλήματα** για να παράγει εκτελέσιμο κώδικα για ένα πρόγραμμα σε IR (Intermediate Representation) αναπαράσταση:

- ***Instruction selection:** μετατρέπει την IR αναπαράσταση στο ISA (Instruction Set Architecture) ενός στοχευμένου επεξεργαστή.
- **Instruction scheduling:** επιλέγει την σειρά με την οποία πρέπει να εκτελεστούν οι λειτουργίες.
- **Register allocation:** επιλέγει ποιες τιμές πρέπει να μένουν στους καταχωρητές και ποιες στην μνήμη.

Οι περισσότεροι μεταγλωττιστές χειρίζονται τις παραπάνω διαδικασίες ξεχωριστά αλλά κατατάσσονται μαζί στον όρο “Code Generation”.

Τι είναι το Code Generation? (2/2)

Κάθε ένα από τα **τρία προβλήματα** είναι και ένα υπολογιστικά δύσκολο πρόβλημα:

- ***Instruction selection:** πρέπει να παράξει γρήγορα κώδικα για ένα CFG(Control Flow Graph) στο οποίο εμπλέκονται μερικές εκατοντάδες εναλλακτικές.
- **Instruction scheduling:** είναι NP-Complete πρόβλημα για ρεαλιστικά block κώδικα.
- **Register allocation:** είναι NP-Complete πρόβλημα λόγο της επεξεργασίας του control flow.

Στις επόμενες διαφάνειες θα καλύψουμε ενδελεχώς μόνο το **Instruction selection**.

The background is a dark navy blue. On the left side, there are several parallel teal lines that start from the top and extend downwards, with some lines turning slightly to the right. On the bottom right, there are several parallel teal lines that start from the bottom and extend towards the top right corner.

CODE GENERATION

INSTRUCTION SELECTION [IS]

Τι είναι το Instruction Selection ?

Για να μπορεί ο κώδικας να εκτελεστεί σε έναν στοχευμένο επεξεργαστή, η IR αναπαράσταση του κώδικα (πολύ σημαντικό το επίπεδο της αφαιρετικότητας της) θα πρέπει να ξαναγραφτεί στο instruction set του επεξεργαστή. Η διαδικασία του mapping (χαρτογράφησης/αντιστοίχησης) των IR λειτουργιών (operations) σε ένα στοχευμένο μηχάνημα καλείτε instruction selection (IS).

Η πολυπλοκότητα του IS βασίζεται στο ότι για μια IR λειτουργία υπάρχουν πάνω από μια υλοποιήσεις στον στοχευμένο επεξεργαστή τις περισσότερες φορές. Γενικά ένας επεξεργαστής παρέχει πολλούς τρόπους για να κάνει τον ίδιο υπολογισμό.

Παράδειγμα, register-to-register copy (1/2)

Έστω ότι θέλουμε να αντιγράψουμε την τιμή μιας μεταβλητής από τον καταχωρητή r_i στον καταχωρητή r_j . Επίσης επιλέγουμε ο στοχευμένο επεξεργαστής να χρησιμοποιεί το **ILOC*** σαν instruction set. Οπότε ποιά εντολή της ILOC πρέπει να χρησιμοποιήσουμε?

Η προφανής απάντηση είναι **i2i** $r_i \Rightarrow r_j$ (δηλαδή $r_j \leftarrow r_i$, για integers) διότι είναι η εντολή που έχει το μικρότερο κόστος στον επεξεργαστή. Βέβαια υπάρχουν και άλλες εντολές που κάνουν το ίδιο πράγμα:

Εναλλακτικές (μερικές)		
<code>addl $r_i, 0 \Rightarrow r_j$</code> (δηλαδή $r_j \leftarrow r_i + 0$)	<code>subl $r_i, 0 \Rightarrow r_j$</code> (δηλαδή $r_j \leftarrow r_i - 0$)	<code>multl $r_i, 1 \Rightarrow r_j$</code> (δηλαδή $r_j \leftarrow r_i \times 0$)
<code>divl $r_i, 1 \Rightarrow r_j$</code> (δηλαδή $r_j \leftarrow r_i \div 1$)	<code>lshifl $r_i, 0 \Rightarrow r_j$</code> (δηλαδή $r_j \leftarrow r_i \ll 0$)	<code>rshifl $r_i, 0 \Rightarrow r_j$</code> (δηλαδή $r_j \leftarrow r_i \gg 0$)
<code>and $r_i, r_i \Rightarrow r_j$</code> (δηλαδή $r_j \leftarrow r_i \& r_i$)	<code>orl $r_i, 0 \Rightarrow r_j$</code> (δηλαδή $r_j \leftarrow r_i \mid 0$)	<code>xorl $r_i, 0 \Rightarrow r_j$</code> (δηλαδή $r_j \leftarrow r_i \oplus 0$)

*Intermediate Language for an Optimizing Compiler, είναι κάτι σαν assembly για απλά RISC μηχανήματα.

Παράδειγμα, register-to-register copy (2/2)

Επίσης υπάρχουν και μερικές ακόμα εναλλακτικές, εάν ο επεξεργαστής είχε έναν καταχωρητή που είχε πάντα την τιμή 0 τότε θα μπορούσε να χρησιμοποιήσει και τις εντολές add, sub, lshift, rshift, or και xor.

Συμπέρασμα:

Ο άνθρωπος μπορεί με ευκολία να βρει την εντολή i2i αλλά το IS πρέπει να λάβει υπόψιν του όλες τις πιθανές εντολές (άρα αύξηση της πολυπλοκότητας) για να κάνει την κατάλληλη επιλογή. Ακόμα και με την χρήση της ILOC που είναι απλή, μπορεί να προκύψουν χιλιάδες τρόποι για την υλοποίηση του παράδειγμα μας.

Φανταστείτε τώρα τους πραγματικούς επεξεργαστές που έχουν ένα υψηλό επίπεδο λειτουργιών και διευθυνσιοδότησης το οποίο μεν βοηθάει έναν καλό προγραμματιστή ή έναν μεταγλωττιστή για την δημιουργία ενός αποδοτικού προγράμματος αλλά από την άλλη εκτοξεύει τον χώρο των πιθανών επιλογών για το IS.

Κάθε εναλλακτική έχει ένα κόστος

Υπάρχουν λειτουργίες (operations) οι οποίες έχουν χαμηλό κόστος όπως το `i2i`, `add` και `lshift` που έχουν **κόστος ενός κύκλου** (άρα τρέχει/εκτελείτε γρήγορα) και άλλες όπως ο πολλαπλασιασμός και η διαίρεση ακεραίων που χρειάζονται περισσότερους κύκλους.

Επίσης πέρα από τους κύκλους υπάρχουν και άλλες μετρικές που μπορεί να μας ενδιαφέρουν όπως η **ενεργειακή κατανάλωση** για κάθε λειτουργία και το **code space** δηλαδή το κόστος να εξαρτάτε μόνο από το `sequence length` (μήκος της αλληλουχία) του κώδικα και για αυτό να επιλέγονται `multioperation sequences` αντί για `single-operation sequences`.

Τέλος, το κόστος μπορεί να υπολογίζεται και από την ενεργειακή κατανάλωση και από το `code space` μαζί.

Περιορισμοί στο IS λόγω ISAs

Υπάρχουν ISAs οι οποίες έχουν επιπλέον περιορισμούς σε κάποιες λειτουργίες τους όπως:

- **Floating-point operations:** μπορεί να χρειαστεί κάποιοι operands να είναι σε άρτιο (αριθμό) καταχωρητή ή να υπάρχει ίσως κάποια operation που να υπολογίζει για παράδειγμα την ακολουθία $(r_i * r_j) + r_k$ πιο γρήγορα από τις operations του πολλαπλασιασμού και της πρόσθεσης ξεχωριστά.
- **Memory operations:** μπορεί να χρειαστεί να εκτελούνται μόνο σε μια μονάδα του επεξεργαστή.
- **Load-multiple & store-multiple operations*:** μπορεί να χρειάζονται συνεχόμενους καταχωρητές.
- **Memory system:** μπορεί να έχει καλύτερο **bandwidth** και **latency** φορτώνοντας doubleword(DWORD) ή quadword(QWORD) απ' ότι singleword(WORD).

*Αυτές οι εντολές παρέχουν ποιο αποδοτικό τρόπο μεταφοράς των περιεχομένων μερικών καταχωρητών από και προς την μνήμη σε σχέση με τις μεμονωμένες load και store. Χρησιμοποιούνται συχνά για αντιγραφή block και για stack operations στις υπορουτίνες entry και exit (των ARM επεξεργαστών).

Εργαλεία/Τεχνικές για περιορισμό του Search space

Εάν υπάρχει σημαντική διαφορά στα επίπεδα αφαίρεσης αναμεσά στο IS και στο ISA, τότε το IS αναλαμβάνει να γεφυρώσει το κενό. Όσο το κενό μεγαλώνει ανάμεσα στα IS και ISA, χρειάζονται **εργαλεία** για να βοηθήσουμε το Code Generation να παράξει ποιοτικό κώδικα.

Το σημαντικότερο είναι να έχουμε πάντα κατά νου το τεράστιο **search space** (χώρο αναζήτησης) τον οποίο πρέπει να ερευνά το IS, ειδικά όταν το search space περιέχει εκατοντάδες εκατομμύρια καταστάσεις.

Οπότε αυτό που θέλουμε είναι να βρούμε **τεχνικές** οι οποίες θα περιορίζουν τις αναζητήσεις ή που θα προ-υπολογίζουν αρκετές πληροφορίες ώστε να κάνουν την αναζήτηση σε βάθος (deep search) αποδοτική.

INSTRUCTION SELECTION [IS]

SIMPLE/EXTENDING TREEWALK SCHEME

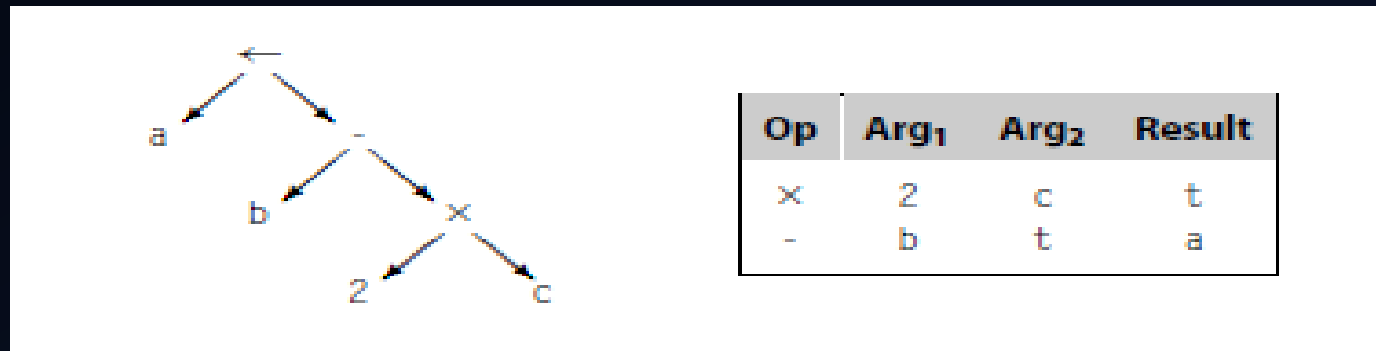
TREE-PATTERN MATCHING

PEEPHOLE OPTIMIZATION

Τι είναι το Simple Treewalk Scheme?

Το IS όταν χρησιμοποιεί το Simple Treewalk Scheme παράγει μια assembly για κάθε instance της ενδιάμεσης αναπαράστασης (IR) είτε αυτή είναι Abstract Syntax Tree (AST) είτε Quadruples(τετραπλέτες).

Η ενδιάμεση αναπαράσταση μιας ανάθεσης της μορφής: $a \leftarrow b - 2 * c$ φαίνεται στα αριστερά σαν AST και στα δεξιά σαν quadruples.



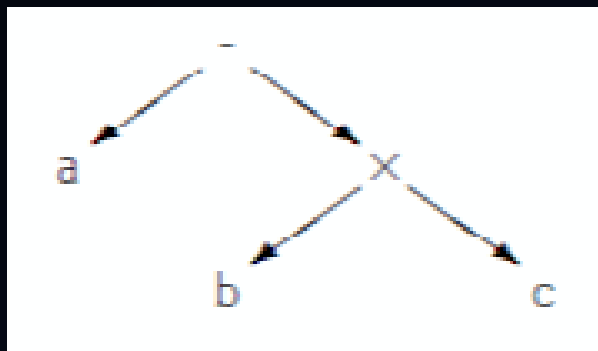
Παράδειγμα I, $a - b * c$? (1/2)

Αρχικά ας υποθέσουμε ότι θέλουμε να παράγουμε κώδικα για την ILOC και πως χρησιμοποιούμε μόνο το παρακάτω υποσύνολο εντολών:

Arithmetic Operations	Memory Operations
add $r_1, r_2 \Rightarrow r_3$ (δηλαδή $r_3 \leftarrow r_1 + r_2$)	store $r_1 \Rightarrow r_2$ (δηλαδή $WORD[r_2] \leftarrow r_1$)
addl $r_1, c_2 \Rightarrow r_3$ (δηλαδή $r_3 \leftarrow r_1 + c_2$)	storeAO $r_1 \Rightarrow r_2, r_3$ (δηλαδή $WORD[r_2 + r_3] \leftarrow r_1$)
sub $r_1, r_2 \Rightarrow r_3$ (δηλαδή $r_3 \leftarrow r_1 - r_2$)	storeAI $r_1 \Rightarrow r_2, c_3$ (δηλαδή $WORD[r_2 + c_3] \leftarrow r_1$)
subl $r_1, c_2 \Rightarrow r_3$ (δηλαδή $r_3 \leftarrow r_1 - c_2$)	loadl $c_1 \Rightarrow r_3$ (δηλαδή $r_3 \leftarrow c_1$)
rsubl $r_1, c_1 \Rightarrow r_3$ (δηλαδή $r_3 \leftarrow c_1 - r_1$)	load $r_1 \Rightarrow r_3$ (δηλαδή $r_3 \leftarrow WORD[r_1]$)
mult $r_1, r_2 \Rightarrow r_3$ (δηλαδή $r_3 \leftarrow r_1 * r_2$)	loadAO $r_1, r_2 \Rightarrow r_3$ (δηλαδή $r_3 \leftarrow WORD[r_1 + r_2]$)
multl $r_1, c_2 \Rightarrow r_3$ (δηλαδή $r_3 \leftarrow r_1 * c_2$)	loadAI $r_1, c_2 \Rightarrow r_3$ (δηλαδή $r_3 \leftarrow WORD[r_1 + c_2]$)

Παράδειγμα Ι, $a - b * c$? (2/2)

Στη συνέχεια, έχουμε το παρακάτω AST δέντρο:



Τέλος, το Simple Treewalk Scheme χρησιμοποιεί τις ρουτίνες που βλέπετε δεξιά για να χειρίζεται τις πράξεις, τις μεταβλητές και τους αριθμούς ώστε να παράξει κώδικα. Τις τιμές των base & offset πιθανός να τις συμβουλεύεται από το symbol table.

```
expr(node) {  
    int result, t1, t2;  
    switch(type(node)) {  
        case X, ÷, +, -:  
            t1 ← expr(LeftChild(node));  
            t2 ← expr(RightChild(node));  
            result ← NextRegister();  
            emit(op(node), t1, t2, result);  
            break;  
        case IDENT:  
            t1 ← base(node);  
            t2 ← offset(node);  
            result ← NextRegister();  
            emit(loadA0, t1, t2, result);  
            break;  
        case NUM:  
            result ← NextRegister();  
            emit(loadI, val(node), none,  
                result);  
            break;  
    }  
    return result;  
}
```

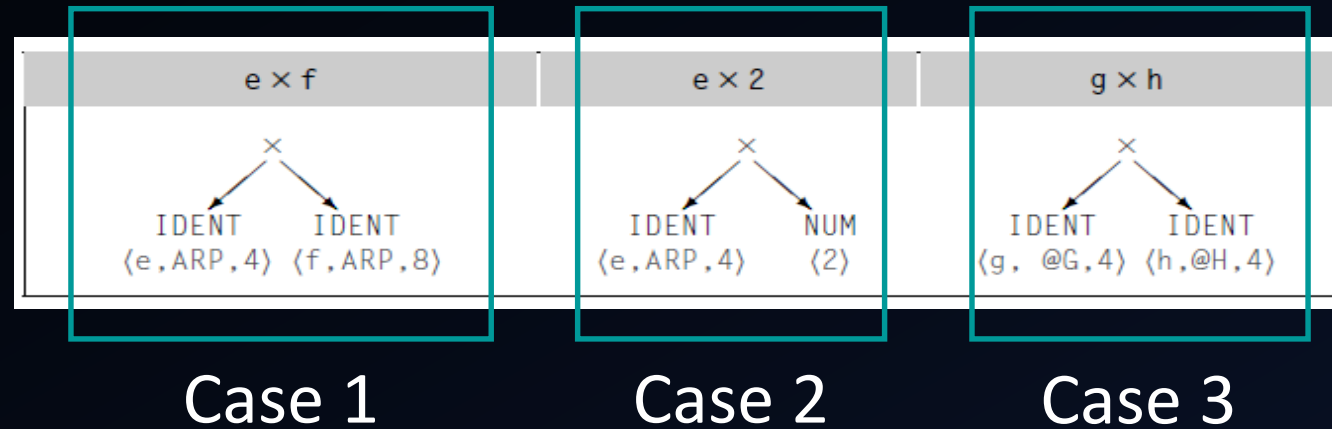
Προβλήματα του Simple Treewalk Scheme

Παρατηρούμε ότι για ρεαλιστικές περιπτώσεις δηλαδή μεταβλητές με διαφορετικά μεγέθη, κλήσεις συναρτήσεων by value ή by reference κ.α. θα πρέπει να γράφουμε κώδικα ειδικά για καθεμιά από αυτές αλλά και των υποπεριπτώσεων τους.

Ακόμα και εάν το κάναμε αυτό π.χ. για την περίπτωση του **IDENT** ο κώδικας θα ήταν πολύ μεγάλος (άρα και αργός). Επίσης, ο κώδικας θα έχανε την απλότητα του και θα ήταν αδύνατον να γραφτεί στο χέρι.

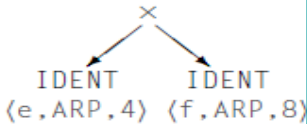
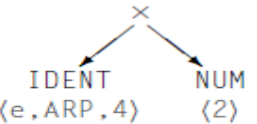
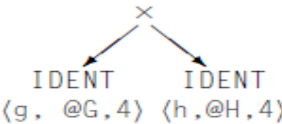
Τέλος ακόμα και ο κώδικας για την περίπτωση του **NUM** είναι πολύ απλοϊκός και υποθέτει ότι όλοι οι αριθμοί φορτώνονται σε καταχωρητές πάντα και πως η val ανακτά την τιμή της από το symbol table.

Παράδειγμα II, 3 διαφορετικοί πολ/σμοί (1/3)



Στο παραπάνω σχήμα υπάρχουν τρεις διαφορετικές περιπτώσεις πολ/σμού. Οπού βλέπουμε για καθεμία από αυτές τον κομβο του πολλαπλασιασμού με τα φύλλα του και κάτω από τα φύλλα βρίσκονται οι εγγραφές του **symbol table** οι οποίες για το **IDENT** είναι $\langle \text{όνομα}, \text{το label της base address (ή το ARP που δείχνει στο current activation record), το offset από την base address} \rangle$.

Παράδειγμα II, 3 διαφορετικοί πολ/σμοί (2/3)

$e \times f$	$e \times 2$	$g \times h$
		
	Generated Code	
<pre>loadI 4 => r5 loadAO rarp,r5 => r6 loadI 8 => r7 loadAO rarp,r7 => r8 mult r6,r8 => r9</pre>	<pre>loadI 4 => r5 loadAO rarp,r5 => r6 loadI 2 => r7 mult r6,r7 => r8</pre>	<pre>loadI @G => r5 loadI 4 => r6 loadAO r5,r6 => r7 loadI @H => r8 loadI 4 => r9 loadAO r8,r9 => r10 mult r7,r10 => r11</pre>
	Desired Code	
<pre>loadAI rarp,4 => r5 loadAI rarp,8 => r6 mult r5,r6 => r7</pre>	<pre>loadAI rarp,4 => r5 multI r5,2 => r6</pre>	<pre>loadI 4 => r5 loadAI r5,@G => r6 loadAI r5,@H => r7 mult r6,r7 => r8</pre>

Case 1

Case 2

Case 3

Παράδειγμα II, 3 διαφορετικοί πολ/σμοί (3/3)

Παρατηρούμε στην προηγούμενη διαφάνεια ότι υπάρχει μια μεγάλη διαφορά ανάμεσα στον **Generated Code** δηλαδή τον κώδικα που προέκυψε από το Simple Treewalk Scheme με τον **Desired Code** δηλαδή τον επιθυμητό κώδικα που θέλουμε από το CG αυτό γίνεται γιατί:

Case 1: στο Simple Treewalk Scheme δεν παράγεται η λειτουργία **loadAI** αν και λύνεται με ποιο περίπλοκο κώδικα στο IDENT.

Case 2: δεν μπορεί να παράξει την λειτουργία **multi** διότι για να την παράξει πρέπει να κοιτάξει σε nonlocal δηλαδή πέρα από το τοπικό context για να βρει ένα σταθερό υποδένδρο, πράγμα που παραβαίνει το Simple Treewalk Scheme.

Case 3: έχουμε 4 **loadI** για τα @G, 4, @H και 4 ενώ θα μπορούσαμε να έχουμε 1 **loadI** για τα δύο τεσσάρια και 2 **loadAI** μόνο, πράγμα που δεν γίνεται επειδή και εδώ έχουμε άλλο ένα nonlocal πρόβλημα.

Συμπέρασμα

Το Simple Treewalk Scheme δεν είναι η καλύτερη επιλογή διότι μέσα στο AST υπάρχουν κρυμμένες υποεκφράσεις.

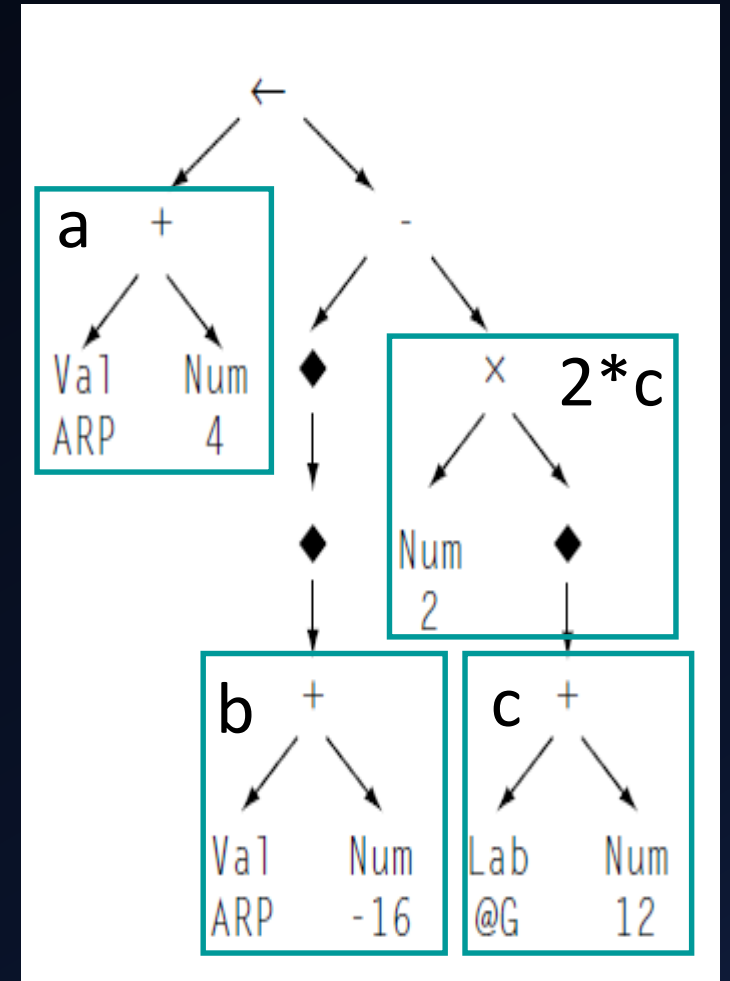
Επίσης, για την εύρεση των πλεονασμών και τον χειρισμό αυτών χρειάζεται κώδικας που να εξετάζει όλες τις περιπτώσεις, πράγμα που οδηγεί σε έναν απογοητευτικά μεγάλο extra κώδικα.

Ο καλύτερος τρόπος για να πιάσουμε τον πλεονασμό είναι να τον εκθέσουμε μέσω της πλεονάζουσας πληροφορίας που υπάρχει στην ενδιάμεση αναπαράσταση (IR) στο Front End του μεταγλωττιστή μέσω ενός **Low-Level AST** ώστε να εφαρμόσουμε μια **Extending μορφή** του Simple Treewalk Scheme.

Τι είναι το Extending Simple Treewalk Scheme? (1/2)

Πρόκειται ουσιαστικά για ένα Low-Level AST το οποίο έχει προκύψει από το FE και έχει μερικά είδη κόμβων παραπάνω. Στο σχήμα φαίνεται το **Low-Level AST της ανάθεσης** $a \leftarrow b - 2 * c$.

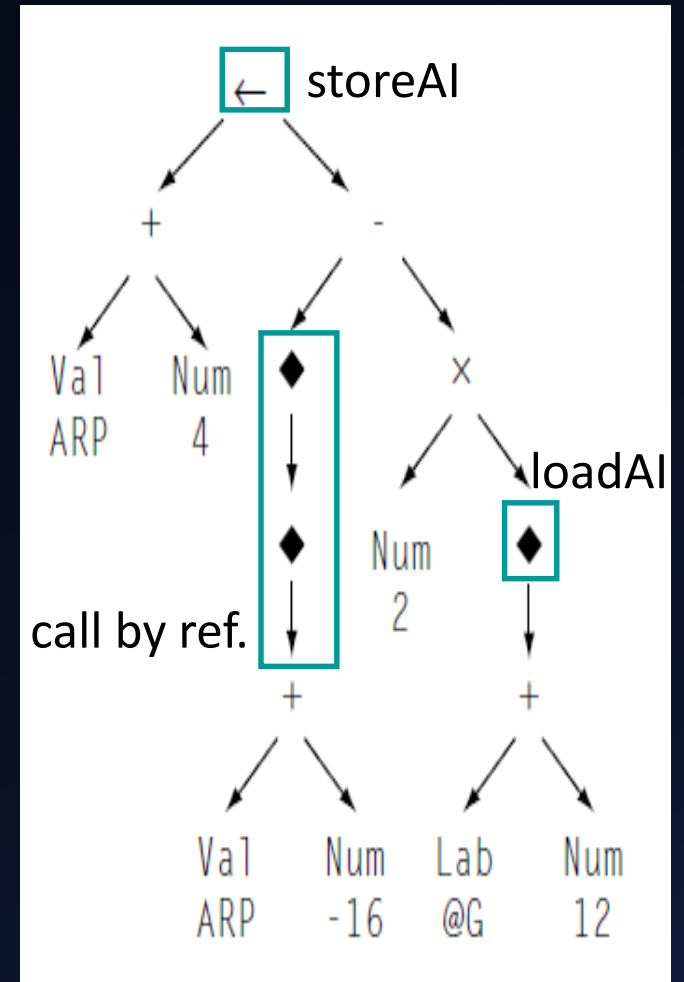
Ο κόμβος **Val** αναφέρεται σε μια μεταβλητή η οποία υπάρχει σε κάποιον καταχωρητή. Ο κόμβος **Lab** αναφέρεται σε μια ετικέτα που χρησιμοποιείτε είτε για κώδικα είτε για δεδομένα. Ο κόμβος \blacklozenge συμβολίζει έναν έμμεσο τύπο διευθυνσιοδότησης, δηλαδή το παιδί αυτού του κόμβου είναι μια address και παράγει την τιμή η οποία αποθηκεύεται στην address.



Τι είναι το Extending Simple Treewalk Scheme? (2/2)

Με την βοήθεια του κόμβου \diamond ο μεταγλωττιστής μπορεί να συγκεκριμενοποιήσει ποιο πολλούς matching rules και να κάνει καλύτερο optimization.

Το επίπεδο αφαίρεσης του δέντρου είναι χαμηλότερο από αυτό της ILOC έτσι εκτίθενται περισσότερες λεπτομέρειες όπως ότι το *a* είναι μια **τοπική μεταβλητή** με offset 4 από το ARP, ότι στην περίπτωση του *b* έχουμε **call-by-reference** επειδή υπάρχουν δυο \diamond κόμβοι. Επίσης η χρήση των λειτουργιών **loadAI** και **storeAI** εμφανίζεται ρητά στο δέντρο είτε σαν κόμβος \diamond είτε σαν το αριστερό παιδί ενός ' \leftarrow ' κόμβου.



Πόσο καλό είναι το Extending Treewalk Scheme?

Εκθέτοντας περισσότερη λεπτομέρεια στο AST σε συνδυασμό με έναν στοχευμένο επεξεργαστή με πολλές λειτουργίες (operations) οδηγούμαστε σε καλύτερο κώδικα **όμως** το CG για να είναι αποδοτικό πρέπει να λαμβάνει υπόψιν του όσους περισσότερους τρόπους είναι δυνατόν για την υλοποίησή ενός υποδένδρου.

Αυτό αντανακλά μια **θεμελιώδη πτυχή του βασικού προβλήματος** το οποίο είναι ότι έχουμε πολλούς τρόπους να υλοποιήσουμε μια IR. Έπειτα, το CG πρέπει να λάβει υπόψιν πολλά πιθανά matches(ταιριάσματα) για ένα υποδένδρο και να επιλέξει ένα από αυτά.

Για την επιλογή αυτή χρειάζεται ένα κόστος για κάθε pattern(πρότυπο), και να επιλέγει αυτό με το μικρότερο κόστος, άρα χρειάζεται άλλα εργαλεία - προσεγγίσεις.

Διαφορετικές προσεγγίσεις

Υπάρχουν ακόμα δυο διαφορετικές προσεγγίσεις για την διαχείριση της πολυπλοκότητας που προκύπτει, από το σύνολο εντολών ενός σύγχρονου μηχανήματος.

- Η πρώτη χρησιμοποιεί την τεχνική του **tree-pattern matching** όπου το σύστημα fold (μαζεύει-διπλώνει) την πολυπλοκότητα στη διαδικασία της κατασκευής του **matcher**.
- Η δεύτερη βασίζεται στην κλασική late-stage transformation (δηλαδή τον μετασχηματισμό τελικού σταδίου), γνωστό ως **peephole optimization** όπου η πολυπλοκότητα μετακινείται σε ένα ενιαίο σύστημα για χαμηλού-επιπέδου απλουστεύσεις που ακολουθείται από μια διαδικασία **pattern matching** για να βρεθούν οι κατάλληλες εντολές.

Και οι δυο τεχνικές για να κρατήσουν χαμηλό το κόστος του matching, έχουν ένα περιορισμένο score σε μικρά τμήματα κώδικα (δυο με τρεις λειτουργίες την φορά). Τέλος και οι δύο έχουν βρει ευρεία χρήση σε πραγματικούς μεταγλωττιστές.

INSTRUCTION SELECTION [IS]

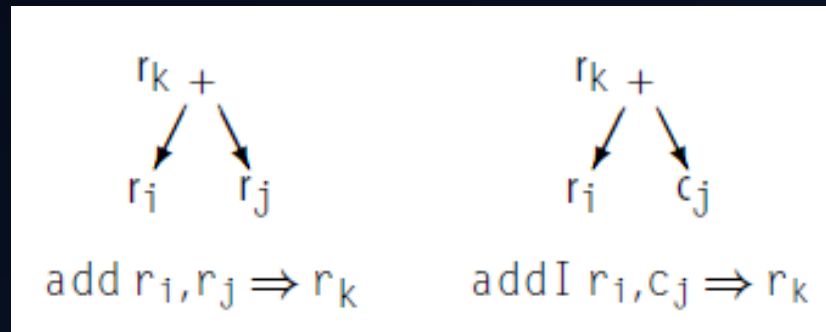
SIMPLE/EXTENDING TREEWALK SCHEME

TREE-PATTERN MATCHING

PEEPHOLE OPTIMIZATION

Τι είναι το Tree-Pattern Matching? (1/3)

Το Tree-Pattern Matching επιτίθεται στην πολυπλοκότητα του IS. Ο μεταγλωττιστής χρησιμοποιεί ένα low-level AST για την ενδιάμεση αναπαράσταση και άλλα low-level AST δένδρα για το Instruction Set του στοχευμένου επεξεργαστή. Έπειτα συστηματικά ο matcher ψάχνει για match ανάμεσα στα υποδένδρα της ενδιάμεσης αναπαράστασης και στα δένδρα του Instruction Set. Για παράδειγμα τα low-level AST δένδρα της πρόσθεσης στην ILOC θα είναι κάπως έτσι:



Τι είναι το Tree-Pattern Matching? (2/3)

Για να δουλέψουμε με τα tree patterns θα χρειαστούμε μια πιο βολική σημειογραφία όπου θα υπάρχει ένα prefix το οποίο θα συμβολίζει την λειτουργία. Πχ. για το **add** θα έχουμε $+(r_i, r_j)$ και για το **addl** θα έχουμε $+(c_i, r_j)$ όπου τα r και c είναι τύποι δεδομένων και αντιστοιχούν στα register και symbol (δηλαδή μια γνωστή σταθερά). Έτσι το Low-Level AST του $a \leftarrow b - 2 * c$ θα γράφετε έτσι:

```
←(+ (Val1, Num1), a  
  -(♦(♦(+ (Val2, Num2))), b  
    ×(Num3, ♦(+ (Lab1, Num4))))))  
      2          c
```

Τι είναι το Tree-Pattern Matching? (3/3)

Παρόλο που η ζωγραφιά ενός δένδρου ενστικτωδώς μας φαίνεται ότι έχει περισσότερη πληροφορία απο το **linear prefix** δεν ισχύει, είναι ισοδύναμα ως προ της πληροφορία.

Δίνοντας μας ένα AST και ένα σύνολο από δέντρα λειτουργιών, ο στόχος μας είναι να χαρτογραφήσουμε το AST σε λειτουργίες, μεσω της κατασκευής **tiling** του AST με τα δέντρα των λειτουργιών.

Ένα **tiling** είναι μια συλλογή από ζευγάρια **<ast_node, op_tree>** όπου **ast_node** είναι ένας κόμβος του AST και το **op_tree** είναι ένα δένδρο μιας λειτουργίας.

Διαφορετικές τεχνικές

Το κλειδί για να πετύχει αυτή η προσέγγιση είναι ο αλγόριθμος να βρίσκει γρήγορα τα καλύτερα tilings για ένα AST. Η χρήση του Low-Level AST είναι η πιο αποδοτική. Ακόμα πολλές φορές μπορεί να υπάρχουν πολλά tilings για έναν κομβό τότε πρέπει να επιλέγεται πάντα αυτό με το μικρότερο κόστος.

Υπάρχουν και άλλες τεχνικές που χρησιμοποιούνται για το matching όπως το tree matching, το text matching και τα bottom-up rewrite systems (BURS).

Τέλος μπορεί να είναι διαφορετικός ο τρόπος υπολογισμού του κόστους με δυο κατηγορίες την **static fixed cost** και αυτή που το κόστος της μπορεί να **αλλάζει** κατά την διαδικασία του matching.

Rewrite rules

Ο δημιουργός του μεταγλωττιστή κωδικοποιεί τις σχέσεις ανάμεσα στα δένδρα λειτουργιών και στα υποδένδρα του AST ξαναγράφοντας τους κανόνες (rewrite rules). Το σύνολο των κανόνων αυτών αποτελείται από μια **tree grammar**, ένα **code template** και ένα **σχετικό κόστος**.

Στην επόμενη διαφάνεια ακολουθεί παράδειγμα με ένα σύνολο από rewrite rules για να κάνουμε tiling ένα Low-Level AST με τις λειτουργίες της ILOC.

Παράδειγμα I, Rewrite rules (1/3)

Tree patterns

Nonterminals

Linearized tree pattern

	Production	Cost	Code Template
1	Goal → Assign	0	
2	Assign → $\leftarrow (\text{Reg}_1, \text{Reg}_2)$	1	store $r_2 \Rightarrow r_1$
3	Assign → $\leftarrow (+ (\text{Reg}_1, \text{Reg}_2), \text{Reg}_3)$	1	storeAO $r_3 \Rightarrow r_1, r_2$
4	Assign → $\leftarrow (+ (\text{Reg}_1, \text{Num}_2), \text{Reg}_3)$	1	storeAI $r_3 \Rightarrow r_1, n_2$
5	Assign → $\leftarrow (+ (\text{Num}_1, \text{Reg}_2), \text{Reg}_3)$	1	storeAI $r_3 \Rightarrow r_2, n_1$
6	Reg → Lab ₁	1	loadI $l_1 \Rightarrow r_{\text{new}}$
7	Reg → Val ₁	0	
8	Reg → Num ₁	1	loadI $n_1 \Rightarrow r_{\text{new}}$
9	Reg → $\blacklozenge (\text{Reg}_1)$	1	load $r_1 \Rightarrow r_{\text{new}}$
10	Reg → $\blacklozenge (+ (\text{Reg}_1, \text{Reg}_2))$	1	loadAO $r_1, r_2 \Rightarrow r_{\text{new}}$
11	Reg → $\blacklozenge (+ (\text{Reg}_1, \text{Num}_2))$	1	loadAI $r_1, n_2 \Rightarrow r_{\text{new}}$
12	Reg → $\blacklozenge (+ (\text{Num}_1, \text{Reg}_2))$	1	loadAI $r_2, n_1 \Rightarrow r_{\text{new}}$
13	Reg → $\blacklozenge (+ (\text{Reg}_1, \text{Lab}_2))$	1	loadAI $r_1, l_2 \Rightarrow r_{\text{new}}$
14	Reg → $\blacklozenge (+ (\text{Lab}_1, \text{Reg}_2))$	1	loadAI $r_2, l_1 \Rightarrow r_{\text{new}}$
15	Reg → $+ (\text{Reg}_1, \text{Reg}_2)$	1	add $r_1, r_2 \Rightarrow r_{\text{new}}$
16	Reg → $+ (\text{Reg}_1, \text{Num}_2)$	1	addI $r_1, n_2 \Rightarrow r_{\text{new}}$
17	Reg → $+ (\text{Num}_1, \text{Reg}_2)$	1	addI $r_2, n_1 \Rightarrow r_{\text{new}}$
18	Reg → $+ (\text{Reg}_1, \text{Lab}_2)$	1	addI $r_1, l_2 \Rightarrow r_{\text{new}}$
19	Reg → $+ (\text{Lab}_1, \text{Reg}_2)$	1	addI $r_2, l_1 \Rightarrow r_{\text{new}}$
20	Reg → $- (\text{Reg}_1, \text{Reg}_2)$	1	sub $r_1, r_2 \Rightarrow r_{\text{new}}$
21	Reg → $- (\text{Reg}_1, \text{Num}_2)$	1	subI $r_1, n_2 \Rightarrow r_{\text{new}}$
22	Reg → $- (\text{Num}_1, \text{Reg}_2)$	1	rsubI $r_2, n_1 \Rightarrow r_{\text{new}}$
23	Reg → $\times (\text{Reg}_1, \text{Reg}_2)$	1	mult $r_1, r_2 \Rightarrow r_{\text{new}}$
24	Reg → $\times (\text{Reg}_1, \text{Num}_2)$	1	multI $r_1, n_2 \Rightarrow r_{\text{new}}$
25	Reg → $\times (\text{Num}_1, \text{Reg}_2)$	1	multI $r_2, n_1 \Rightarrow r_{\text{new}}$

ILOC implementations

Παράδειγμα I, Rewrite rules (2/3)

Παρατηρούμε ότι υπάρχει η **αντιμεταθετική ιδιότητα** στους κανόνες 16 και 17 γιατί πρέπει να είναι σαφής οι κανόνες για να γίνει match.

Η χρήση του **Reg** φαίνεται σαν να είναι και Terminal και Nonterminal αλλά αυτό γίνεται για λόγους συντομογραφίας κανονικά θα έπρεπε να είναι $\text{Reg} \rightarrow r_0$, $\text{Reg} \rightarrow r_1, \dots \text{Reg} \rightarrow r_k$.

Τα **Nonterminals** επιτρέπουν την αφαιρετικότητα και εξυπηρετούν την σύνδεση των κανόνων της γραμματικής.

Το **Reg** αντιπροσωπεύει μια μεταβλητή από ένα υποδένδρο το οποίο αποθηκεύεται σε έναν καταχωρητή.

Το **Val** αντιπροσωπεύει μια μεταβλητή που είναι αποθηκευμένη σε έναν καταχωρητή πχ το ARP.

	Production	Cost	Code Template
1	Goal \rightarrow Assign	0	
2	Assign \rightarrow $\leftarrow (\text{Reg}_1, \text{Reg}_2)$	1	store $r_2 \Rightarrow r_1$
3	Assign \rightarrow $\leftarrow (+ (\text{Reg}_1, \text{Reg}_2), \text{Reg}_3)$	1	storeAO $r_3 \Rightarrow r_1, r_2$
4	Assign \rightarrow $\leftarrow (+ (\text{Reg}_1, \text{Num}_2), \text{Reg}_3)$	1	storeAI $r_3 \Rightarrow r_1, n_2$
5	Assign \rightarrow $\leftarrow (+ (\text{Num}_1, \text{Reg}_2), \text{Reg}_3)$	1	storeAI $r_3 \Rightarrow r_2, n_1$
6	Reg \rightarrow Lab ₁	1	loadI $l_1 \Rightarrow r_{\text{new}}$
7	Reg \rightarrow Val ₁	0	
8	Reg \rightarrow Num ₁	1	loadI $n_1 \Rightarrow r_{\text{new}}$
9	Reg \rightarrow $\blacklozenge (\text{Reg}_1)$	1	load $r_1 \Rightarrow r_{\text{new}}$
10	Reg \rightarrow $\blacklozenge (+ (\text{Reg}_1, \text{Reg}_2))$	1	loadAO $r_1, r_2 \Rightarrow r_{\text{new}}$
11	Reg \rightarrow $\blacklozenge (+ (\text{Reg}_1, \text{Num}_2))$	1	loadAI $r_1, n_2 \Rightarrow r_{\text{new}}$
12	Reg \rightarrow $\blacklozenge (+ (\text{Num}_1, \text{Reg}_2))$	1	loadAI $r_2, n_1 \Rightarrow r_{\text{new}}$
13	Reg \rightarrow $\blacklozenge (+ (\text{Reg}_1, \text{Lab}_2))$	1	loadAI $r_1, l_2 \Rightarrow r_{\text{new}}$
14	Reg \rightarrow $\blacklozenge (+ (\text{Lab}_1, \text{Reg}_2))$	1	loadAI $r_2, l_1 \Rightarrow r_{\text{new}}$
15	Reg \rightarrow $+$ $(\text{Reg}_1, \text{Reg}_2)$	1	add $r_1, r_2 \Rightarrow r_{\text{new}}$
16	Reg \rightarrow $+$ $(\text{Reg}_1, \text{Num}_2)$	1	addI $r_1, n_2 \Rightarrow r_{\text{new}}$
17	Reg \rightarrow $+$ $(\text{Num}_1, \text{Reg}_2)$	1	addI $r_2, n_1 \Rightarrow r_{\text{new}}$
18	Reg \rightarrow $+$ $(\text{Reg}_1, \text{Lab}_2)$	1	addI $r_1, l_2 \Rightarrow r_{\text{new}}$
19	Reg \rightarrow $+$ $(\text{Lab}_1, \text{Reg}_2)$	1	addI $r_2, l_1 \Rightarrow r_{\text{new}}$
20	Reg \rightarrow $-$ $(\text{Reg}_1, \text{Reg}_2)$	1	sub $r_1, r_2 \Rightarrow r_{\text{new}}$
21	Reg \rightarrow $-$ $(\text{Reg}_1, \text{Num}_2)$	1	subI $r_1, n_2 \Rightarrow r_{\text{new}}$
22	Reg \rightarrow $-$ $(\text{Num}_1, \text{Reg}_2)$	1	rsubI $r_2, n_1 \Rightarrow r_{\text{new}}$
23	Reg \rightarrow \times $(\text{Reg}_1, \text{Reg}_2)$	1	mult $r_1, r_2 \Rightarrow r_{\text{new}}$
24	Reg \rightarrow \times $(\text{Reg}_1, \text{Num}_2)$	1	multI $r_1, n_2 \Rightarrow r_{\text{new}}$
25	Reg \rightarrow \times $(\text{Num}_1, \text{Reg}_2)$	1	multI $r_2, n_1 \Rightarrow r_{\text{new}}$

Παράδειγμα I, Rewrite rules (3/3)

Παρατηρούμε ακόμα ότι οι κανόνες 10 ως 14 υποστηρίζουν τις λειτουργίες loadAO και loadAI πράγμα που στο Simple Treewalk Scheme δεν ήταν εφικτό.

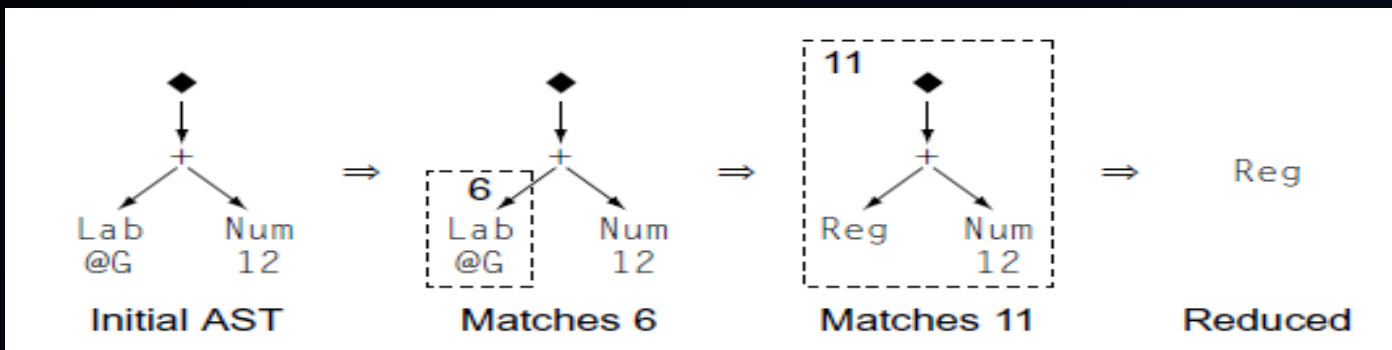
Κάθε υποδένδρο που μπορεί να κάνει match με έναν από τους πέντε παραπάνω κανόνες μπορεί να κάνει tile και με έναν συνδυασμό των άλλων κανόνων.

Πχ. ένα υποδένδρο που κάνει match με τον κανόνα 10, μπορεί να κάνει match και με τον συνδυασμό του κανόνα 15 που παράγει μια address και του κανόνα 9 που το φορτώνει. Αυτό γίνεται λόγω της ασάφειας(ambiguous) που έχουν οι rewrite rules.

Η ασάφεια αυτή μας δείχνει το γεγονός ότι στο στοχευμένο μηχάνημα υπάρχουν αρκετοί τρόποι για να υλοποιηθεί ένα υποδένδρο.

	Production	Cost	Code Template
1	Goal \rightarrow Assign	0	
2	Assign $\rightarrow \leftarrow (Reg_1, Reg_2)$	1	store $r_2 \Rightarrow r_1$
3	Assign $\rightarrow \leftarrow (+ (Reg_1, Reg_2), Reg_3)$	1	storeAO $r_3 \Rightarrow r_1, r_2$
4	Assign $\rightarrow \leftarrow (+ (Reg_1, Num_2), Reg_3)$	1	storeAI $r_3 \Rightarrow r_1, n_2$
5	Assign $\rightarrow \leftarrow (+ (Num_1, Reg_2), Reg_3)$	1	storeAI $r_3 \Rightarrow r_2, n_1$
6	Reg $\rightarrow Lab_1$	1	loadI $l_1 \Rightarrow r_{new}$
7	Reg $\rightarrow Val_1$	0	
8	Reg $\rightarrow Num_1$	1	loadI $n_1 \Rightarrow r_{new}$
9	Reg $\rightarrow \blacklozenge (Reg_1)$	1	load $r_1 \Rightarrow r_{new}$
10	Reg $\rightarrow \blacklozenge (+ (Reg_1, Reg_2))$	1	loadAO $r_1, r_2 \Rightarrow r_{new}$
11	Reg $\rightarrow \blacklozenge (+ (Reg_1, Num_2))$	1	loadAI $r_1, n_2 \Rightarrow r_{new}$
12	Reg $\rightarrow \blacklozenge (+ (Num_1, Reg_2))$	1	loadAI $r_2, n_1 \Rightarrow r_{new}$
13	Reg $\rightarrow \blacklozenge (+ (Reg_1, Lab_2))$	1	loadAI $r_1, l_2 \Rightarrow r_{new}$
14	Reg $\rightarrow \blacklozenge (+ (Lab_1, Reg_2))$	1	loadAI $r_2, l_1 \Rightarrow r_{new}$
15	Reg $\rightarrow + (Reg_1, Reg_2)$	1	add $r_1, r_2 \Rightarrow r_{new}$
16	Reg $\rightarrow + (Reg_1, Num_2)$	1	addI $r_1, n_2 \Rightarrow r_{new}$
17	Reg $\rightarrow + (Num_1, Reg_2)$	1	addI $r_2, n_1 \Rightarrow r_{new}$
18	Reg $\rightarrow + (Reg_1, Lab_2)$	1	addI $r_1, l_2 \Rightarrow r_{new}$
19	Reg $\rightarrow + (Lab_1, Reg_2)$	1	addI $r_2, l_1 \Rightarrow r_{new}$
20	Reg $\rightarrow - (Reg_1, Reg_2)$	1	sub $r_1, r_2 \Rightarrow r_{new}$
21	Reg $\rightarrow - (Reg_1, Num_2)$	1	subI $r_1, n_2 \Rightarrow r_{new}$
22	Reg $\rightarrow - (Num_1, Reg_2)$	1	rsubI $r_2, n_1 \Rightarrow r_{new}$
23	Reg $\rightarrow \times (Reg_1, Reg_2)$	1	mult $r_1, r_2 \Rightarrow r_{new}$
24	Reg $\rightarrow \times (Reg_1, Num_2)$	1	multI $r_1, n_2 \Rightarrow r_{new}$
25	Reg $\rightarrow \times (Num_1, Reg_2)$	1	multI $r_2, n_1 \Rightarrow r_{new}$

Παράδειγμα II, πως γίνονται τα matches? (1/3)

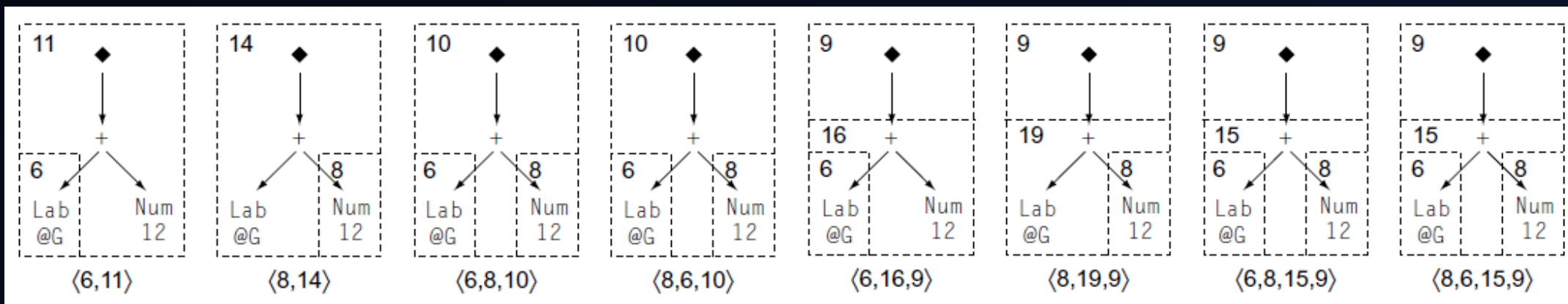
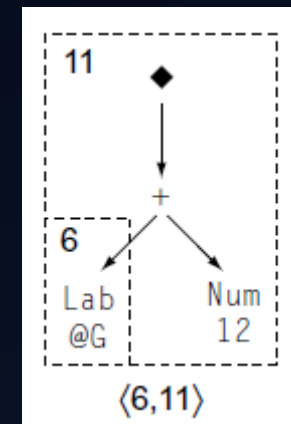


Αρχικά, βλέπουμε το υποδένδρο της μεταβλητής c από ένα παλιότερο παράδειγμα μας. Στην συνέχεια φαίνεται το **match** που υπάρχει με τον **κανόνα 6** της δεξιάς εικόνας το οποίο μας επιτρέπει να το ξαναγράψουμε σαν **Reg**. Τέλος, φαίνονται στο ξαναγραμμένο δένδρο το **match** που έχουμε με τον **κανόνα 11** της δεξιάς εικόνας το οποίο μας επιτρέπει να ξαναγράψουμε ολόκληρο το υποδένδρο σαν **Reg**.

	Production	Cost	Code Template
1	Goal \rightarrow Assign	0	
2	Assign \rightarrow $\leftarrow (Reg_1, Reg_2)$	1	store $r_2 \Rightarrow r_1$
3	Assign \rightarrow $\leftarrow (+ (Reg_1, Reg_2), Reg_3)$	1	storeAO $r_3 \Rightarrow r_1, r_2$
4	Assign \rightarrow $\leftarrow (+ (Reg_1, Num_2), Reg_3)$	1	storeAI $r_3 \Rightarrow r_1, n_2$
5	Assign \rightarrow $\leftarrow (+ (Num_1, Reg_2), Reg_3)$	1	storeAI $r_3 \Rightarrow r_2, n_1$
6	Reg \rightarrow Lab ₁	1	loadI $l_1 \Rightarrow r_{new}$
7	Reg \rightarrow Val ₁	0	
8	Reg \rightarrow Num ₁	1	loadI $n_1 \Rightarrow r_{new}$
9	Reg \rightarrow $\blacklozenge (Reg_1)$	1	load $r_1 \Rightarrow r_{new}$
10	Reg \rightarrow $\blacklozenge (+ (Reg_1, Reg_2))$	1	loadAO $r_1, r_2 \Rightarrow r_{new}$
11	Reg \rightarrow $\blacklozenge (+ (Reg_1, Num_2))$	1	loadAI $r_1, n_2 \Rightarrow r_{new}$
12	Reg \rightarrow $\blacklozenge (+ (Num_1, Reg_2))$	1	loadAI $r_2, n_1 \Rightarrow r_{new}$
13	Reg \rightarrow $\blacklozenge (+ (Reg_1, Lab_2))$	1	loadAI $r_1, l_2 \Rightarrow r_{new}$
14	Reg \rightarrow $\blacklozenge (+ (Lab_1, Reg_2))$	1	loadAI $r_2, l_1 \Rightarrow r_{new}$
15	Reg \rightarrow $+$ (Reg_1, Reg_2)	1	add $r_1, r_2 \Rightarrow r_{new}$
16	Reg \rightarrow $+$ (Reg_1, Num_2)	1	addI $r_1, n_2 \Rightarrow r_{new}$
17	Reg \rightarrow $+$ (Num_1, Reg_2)	1	addI $r_2, n_1 \Rightarrow r_{new}$
18	Reg \rightarrow $+$ (Reg_1, Lab_2)	1	addI $r_1, l_2 \Rightarrow r_{new}$
19	Reg \rightarrow $+$ (Lab_1, Reg_2)	1	addI $r_2, l_1 \Rightarrow r_{new}$
20	Reg \rightarrow $- (Reg_1, Reg_2)$	1	sub $r_1, r_2 \Rightarrow r_{new}$
21	Reg \rightarrow $- (Reg_1, Num_2)$	1	subI $r_1, n_2 \Rightarrow r_{new}$
22	Reg \rightarrow $- (Num_1, Reg_2)$	1	rsubI $r_2, n_1 \Rightarrow r_{new}$
23	Reg \rightarrow $\times (Reg_1, Reg_2)$	1	mult $r_1, r_2 \Rightarrow r_{new}$
24	Reg \rightarrow $\times (Reg_1, Num_2)$	1	multI $r_1, n_2 \Rightarrow r_{new}$
25	Reg \rightarrow $\times (Num_1, Reg_2)$	1	multI $r_2, n_1 \Rightarrow r_{new}$

Παράδειγμα II, πως γίνονται τα matches? (2/3)

Η προηγούμενη ακολουθία των matches με τους κανόνες 6 και 11 μπορεί να συμβολιστεί με το $\langle 6,11 \rangle$, το οποίο μείωσε ολόκληρο το υποδένδρο σε έναν Reg. Η λίστα κάτω από την εικόνα που είναι δεξιά, μας δείχνουν την σειρά με την οποία εφαρμόστηκαν οι κανόνες. Όμως υπάρχουν και άλλα πιθανά matches όπως αυτά:



Κόστος:

2

2

3

3

3

3

4

4

Παράδειγμα II, πως γίνονται τα matches? (3/3)

Ο IS πρέπει να επιλέξει το tiling που παράγει το μικρότερο κόστος. Στην περίπτωση μας έχουμε τα $\langle 6,11 \rangle$ και $\langle 8,14 \rangle$ με κόστος δυο και μπορούμε ελεύθερα να επιλέξουμε ένα από τα δυο.

Εδώ χρειάζεται **προσοχή** γιατί εντολές όπως η **loadAI** δέχονται σαν ορίσματα μεταβλητές με περιορισμένο μέγεθος, οπότε η ακολουθία $\langle 8,14 \rangle$ μπορεί να μην δουλέψει για μια πολύ μεγάλη address του @G. Για να μην έχουμε τέτοια προβλήματα δίνεται η δυνατότητα να προσθέσουμε ένα νέο τερματικό σύμβολο στην γραμματική μας το οποίο να δέχεται ορισμένο εύρος πχ. το πολύ ως 12-bit.

<div>loadI @G $\Rightarrow r_i$ loadAI $r_i, 12 \Rightarrow r_j$</div> <div>(6,11)</div>	<div>loadI 12 $\Rightarrow r_i$ loadAI $r_i, @G \Rightarrow r_j$</div> <div>(8,14)</div>	<div>loadI @G $\Rightarrow r_i$ loadI 12 $\Rightarrow r_j$ loadAO $r_i, r_j \Rightarrow r_k$</div> <div>(6,8,10)</div>	<div>loadI 12 $\Rightarrow r_i$ loadI @G $\Rightarrow r_j$ loadAO $r_i, r_j \Rightarrow r_k$</div> <div>(8,6,10)</div>	<div>loadI @G $\Rightarrow r_i$ addI $r_i, 12 \Rightarrow r_j$ load $r_j \Rightarrow r_k$</div> <div>(6,16,9)</div>	<div>loadI 12 $\Rightarrow r_i$ addI $r_i, @G \Rightarrow r_j$ load $r_j \Rightarrow r_k$</div> <div>(8,19,9)</div>	<div>loadI @G $\Rightarrow r_i$ loadI 12 $\Rightarrow r_j$ add $r_i, r_j \Rightarrow r_k$ load $r_k \Rightarrow r_l$</div> <div>(6,8,15,9)</div>	<div>loadI 12 $\Rightarrow r_i$ loadI @G $\Rightarrow r_j$ add $r_i, r_j \Rightarrow r_k$ load $r_k \Rightarrow r_l$</div> <div>(8,6,15,9)</div>
--	--	---	---	--	--	--	--

Κόστος:

2

2

3

3

3

3

4

4

Αλγόριθμοι tree pattern matching

Ο αλγόριθμος του tree pattern matching κάνει μια post order διάσχιση του AST δένδρου και εξετάζει όλους τους κόμβους του ξεχωριστά με όλες τις λειτουργίες του στοχευμένου επεξεργαστή.

Για να επιταχυνθεί αυτή η διαδικασία το CG προϋπολογίζει όλα τα πιθανά matches και τα αποθηκεύει σε έναν πίνακα. Για να γίνει ακόμα πιο αποδοτικός ο αλγόριθμος αντικαθιστά όλα τα for loops με έναν lookup πίνακα, έτσι ο αλγόριθμος γίνεται γραμμικός ως προς το κόστος διάσχισης του δένδρου.

Παρόλο που ο lookup πίνακας περιμένουμε να είναι μεγάλος, ξέρουμε από την δομή της γραμματικής μας ότι αποκλείονται πολλοί συνδυασμοί και αυτό έχει σαν αποτέλεσμα ο πίνακας να είναι αραιός και αυτό μας επιτρέπει την αποδοτική του κωδικοποίηση.

Εργαλεία

1. Φτιάξιμο του matcher **με το χέρι**.
2. Μπορούμε να το κωδικοποιήσουμε σαν ένα πρόβλημα **Finite Automaton** (tree matching automaton) και να αποκτήσει την συμπεριφορά ενός **DFA**. Πολλά συστήματα έχουν κατασκευάσει και χρησιμοποιούν αυτήν την προσέγγιση, και την αποκαλούν bottom-up rewrite systems (BURS).
3. Μπορούμε να χρησιμοποιήσουμε **αλγόριθμους parsing** αρκεί να τους επεκτείνουμε ώστε να χειρίζονται ασαφής γραμματικές και να επιλέγουμε το parse με το μικρότερο κόστος.
4. Με γραμμική μορφοποίηση (linearizing) του δένδρου σε prefix strings το πρόβλημα μπορεί να μεταφραστεί σε **string-matching πρόβλημα**. Έτσι, ο μεταγλωττιστής χρησιμοποιεί string-pattern matching για να βρει τα πιθανά matches.

Όλα τα εργαλεία παράγουν γρήγορο και αποδοτικό κώδικα. Ειδικά τα τρία τελευταία εργαλεία είναι διαθέσιμα, αρκεί να τους δώσεις μια περιγραφή του instruction set του μηχανήματος και ο CG παράγει εκτελέσιμο κώδικα για αυτήν την περιγραφή.

INSTRUCTION SELECTION [IS]

SIMPLE/EXTENDING TREEWALK SCHEME

TREE-PATTERN MATCHING

PEEPHOLE OPTIMIZATION

Τι είναι το Peephole Optimization ?

Η βασική αρχή του Peephole Optimization είναι απλή: ο μεταγλωττιστής μπορεί αποτελεσματικά να βρει **τοπικές βελτιώσεις** εξετάζοντας μικρές ακολουθίες γειτονικών λειτουργιών. Ο Optimizer τρέχει κατά την διάρκεια της μεταγλώττισής, ταυτόχρονα **καταναλώνει και παράγει** assembly κώδικα. Έχει ένα **μικρό κυλιόμενο παράθυρο** ή ματάκι (peephole) μέσα στο οποίο εξετάζει τις λειτουργίες και ψάχνει για συγκεκριμένα patterns. Όταν αναγνωρίσει ένα pattern τότε απλά θα το **ξαναγράψει με μια καλύτερη** ακολουθία λειτουργιών. Ο συνδυασμός των περιορισμένων patterns και ο μικρός αριθμός λειτουργιών που εστιάζει κάθε φορά οδηγεί σε γρήγορη επεξεργασία.

Παραδείγματα, Peephole Optimization

- Έχουμε ένα store το οποίο ακολουθείτε από ένα load από την ίδια θέση. Οπότε ο Peephole Optimizer αντικαθιστά το load με ένα copy.

```
storeAI r1    ⇒ rarp,8    ⇒ storeAI r1 ⇒ rarp,8  
loadAI  rarp,8 ⇒ r15      ⇒ i2i      r1 ⇒ r15
```

- Εδώ έχουμε μια πρόσθεση και έναν πολλαπλασιασμό. Οπότε ο Peephole Optimizer βλέπει ότι στον καταχωρητή r_7 αποθηκεύεται η πρόσθεση του r_2 με το 0, άρα το $r_7 = r_2$ έτσι αντικαθιστά το r_7 με το r_2 της mult και δεν τρέχει καν το addl. Ο Peephole Optimizer **περιλαμβάνει και απλές αλγεβρικές ταυτότητες**.

```
addI r2,0 ⇒ r7    ⇒ mult r4,r2 ⇒ r10  
mult r4,r7 ⇒ r10
```

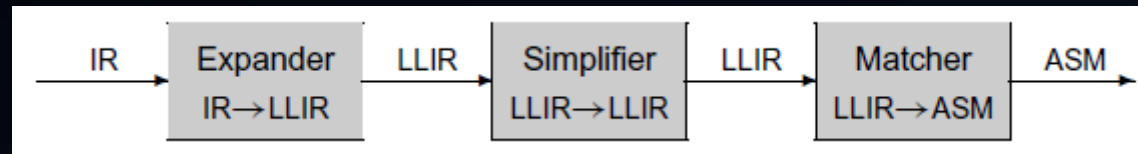
Πως δουλεύει το σύγχρονο Peephole Optimization?

(1/2)

Παλαιότερα τα patterns γράφονταν στο χέρι και έκαναν εξαντλητική αναζήτηση για να κάνουν match και αυτό το σύστημα έτρεχε γρήγορα επειδή είχε λίγα patterns και μικρό παράθυρο που χώραγε δυο με τρεις λειτουργίες.

Πλέον έχει προχωρήσει πέρα από το matching με ένα μικρό αριθμό από patterns. Οι σύγχρονες ISAs οδηγούν σε πιο συστηματικές προσεγγίσεις. Ο σύγχρονος Peephole Optimizer χωρίζει την διαδικασία σε τρεις φάσεις οι οποίες είναι: η επέκταση (expansion), η απλοποίηση (simplifier) και το ταίριασμα (matching). Η διαδικασία αυτή μοιάζει με έναν μεταγλωττιστή.

Πως δουλεύει το σύγχρονο Peephole Optimization? (2/2)



Ο **expander** ξαναγράφει την ενδιάμεση αναπαράσταση (IR) λειτουργία - λειτουργία σε μια ακολουθία από lower-level IR (LLIR) λειτουργίες. Έχει μια απλή δομή και το expand γίνεται ανεξαρτήτως περιεχομένου και για όλες τις λειτουργίες. Πχ. το **add** $r_i, r_j \rightarrow r_k$ γίνεται $r_i + r_j \rightarrow r_k$.

*Ο **simplifier** κάνει ένα πέρασμα στο LLIR και εφαρμόζει ένα μικρό παράθυρο και προσπαθεί συστηματικά να τα βελτιώσει. Οι βασικοί του μηχανισμοί είναι οι: forward substitution (αντικατάσταση), αλγεβρικές απλοποιήσεις (πχ. $x+0=x$), αξιολόγηση σταθερών-τιμών σε εκφράσεις (πχ. $2+17=19$) και ο αποκλεισμός αχρήστων effects όπως κώδικας αχρησιμοποίητων συνθηκών.

Ο **matcher** συγκρίνει την απλοποιημένη LLIR με την pattern βιβλιοθήκη του και κοιτάει για το pattern το οποίο αποτυπώνει την καλύτερη επίδραση στην LLIR και παράγει την assembly.

Παράδειγμα, σύγχρονου Peephole Optimization (1/4)

Έχουμε μια τετραπλέτα η οποία αναπαριστά την ανάθεση $a \leftarrow b - 2 * c$ την οποία έχουμε χρησιμοποιήσει και σε παλαιότερα παραδείγματα και το αποτέλεσμα του expander είναι:

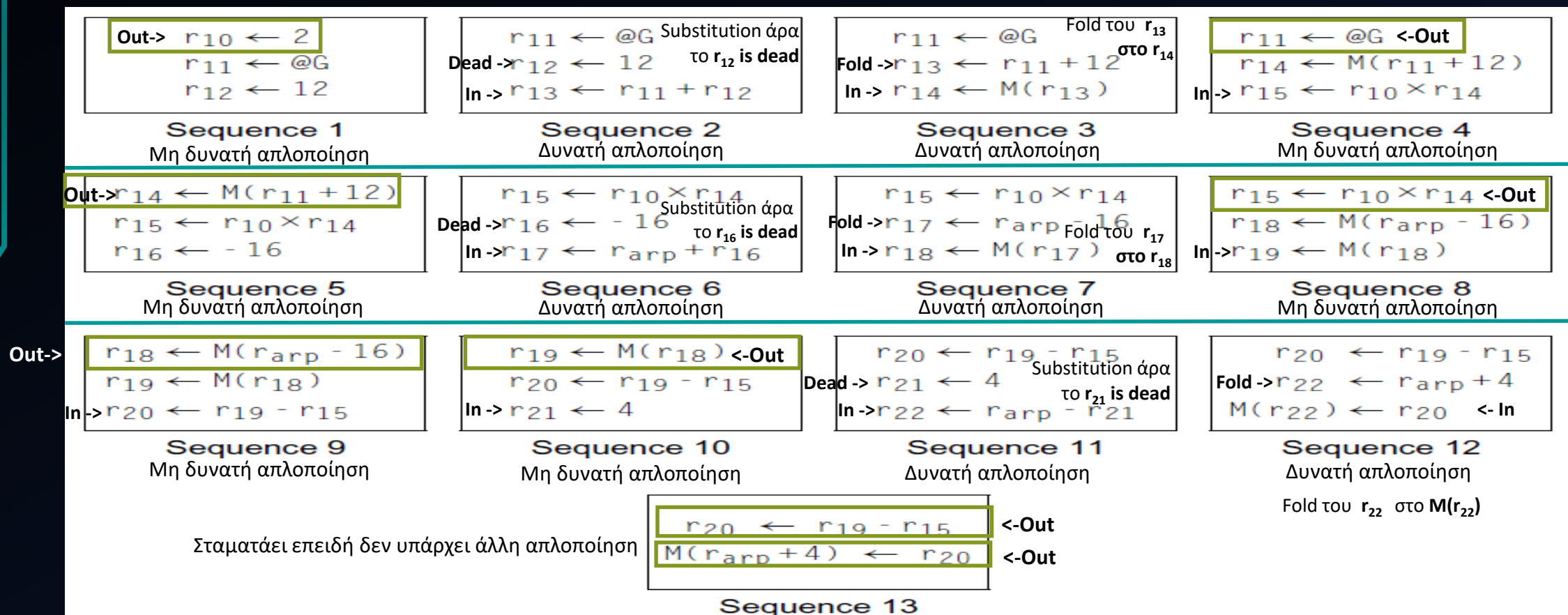
Op	Arg ₁	Arg ₂	Result
×	2	c	t ₁
-	b	t ₁	a

Expand
⇒

r10	← 2	2
r11	← @G	label @G
r12	← 12	offset 12
r13	← r11 + r12	
r14	← M(r13)	c
r15	← r10 × r14	2 * c
r16	← -16	offset -16
r17	← rarp + r16	
r18	← M(r17)	
r19	← M(r18)	b called by reference
r20	← r19 - r15	b - (2 * c)
r21	← 4	offset 4
r22	← rarp + r21	
M(r22)	← r20	a ← b - (2 * c)

Παράδειγμα, σύγχρονου Peephole Optimization (2/4)

Εδώ έχουμε τις ακολουθίες μαζί με τα παράθυρα του Simplifier:



Παράδειγμα, σύγχρονου Peephole Optimization (3/4)

Ότι είδαμε στην προηγούμενη διαφάνεια σαν **out** αποτελεί το Simplified LLIR, τώρα είναι η σειρά του Matcher να παράξει την assembly:

Op	Arg ₁	Arg ₂	Result
×	2	c	t ₁
-	b	t ₁	a

Expand

⇒

```
r10 ← 2
r11 ← @G
r12 ← 12
r13 ← r11 + r12
r14 ← M(r13)
r15 ← r10 × r14
r16 ← -16
r17 ← rarp + r16
r18 ← M(r17)
r19 ← M(r18)
r20 ← r19 - r15
r21 ← 4
r22 ← rarp + r21
M(r22) ← r20
```

Simplify

⇒

Matcher

```
r10 ← 2          loadI  2          ⇒ r10
r11 ← @G         loadI  @G         ⇒ r11
r14 ← M(r11+12)  Match loadAI r11,12 ⇒ r14
r15 ← r10 × r14  mult   r10,r14    ⇒ r15
r18 ← M(rarp - 16) ⇒ loadAI rarp,-16 ⇒ r18
r19 ← M(r18)     load   r18         ⇒ r19
r20 ← r19 - r15  sub    r19,r15     ⇒ r20
M(rarp+4) ← r20  storeAI r20        ⇒ rarp,4
```

Παράδειγμα, σύγχρονου Peephole Optimization (4/4)

Ο απολογισμός είναι τελικά:

- Χρησιμοποιήσαμε 8 αντί για 14 λειτουργίες (operations) άρα 57% λιγότερους operations.
- Χρησιμοποιήσαμε 7 (χωρίς τον ARP) αντί για 13 καταχωρητές άρα 53% λιγότερους registers.

Op	Arg1	Arg2	Result
×	2	c	t ₁
-	b	t ₁	a

Expand
⇒

```
r10 ← 2
r11 ← @G
r12 ← 12
r13 ← r11 + r12
r14 ← M(r13)
r15 ← r10 × r14
r16 ← -16
r17 ← rarp + r16
r18 ← M(r17)
r19 ← M(r18)
r20 ← r19 - r15
r21 ← 4
r22 ← rarp + r21
M(r22) ← r20
```

Simplify
⇒

```
r10 ← 2
r11 ← @G
r14 ← M(r11+12)
r15 ← r10 × r14
r18 ← M(rarp - 16)
r19 ← M(r18)
r20 ← r19 - r15
M(rarp+4) ← r20
```

Match
⇒

```
loadI 2 ⇒ r10
loadI @G ⇒ r11
loadAI r11,12 ⇒ r14
mult r10,r14 ⇒ r15
loadAI rarp,-16 ⇒ r18
load r18 ⇒ r19
sub r19,r15 ⇒ r20
storeAI r20 ⇒ rarp,4
```

Σχεδιαστικά θέματα του Peephole Optimization

Για να έχουμε έναν καλύτερο κώδικα, κρίσιμο ρόλο παίζει η ικανότητα που έχει ο optimizer για να ανιχνεύσει τις νεκρές (dead) μεταβλητές. Ακόμα το μέγεθος του παραθύρου περιορίζει την ικανότητα του optimizer να ενώσει σχετικές λειτουργίες πχ. ένα μεγάλο παράθυρο θα έκανε fold την σταθερά 2 στο πολλαπλασιασμό του προηγούμενου παραδείγματος. Τέλος, και ο χειρισμός των control-flow λειτουργιών (πχ jumps) καθορίζει τι θα συμβεί στα όρια των blocks.

Ερωτήσεις, απορίες, παρατηρήσεις ...

