Instruction Selection

Compilers course

Masters in Informatics and Computing Engineering (MIEIC), 3rd Year

João M. P. Cardoso
Email: jmpc@fe.up.pt

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

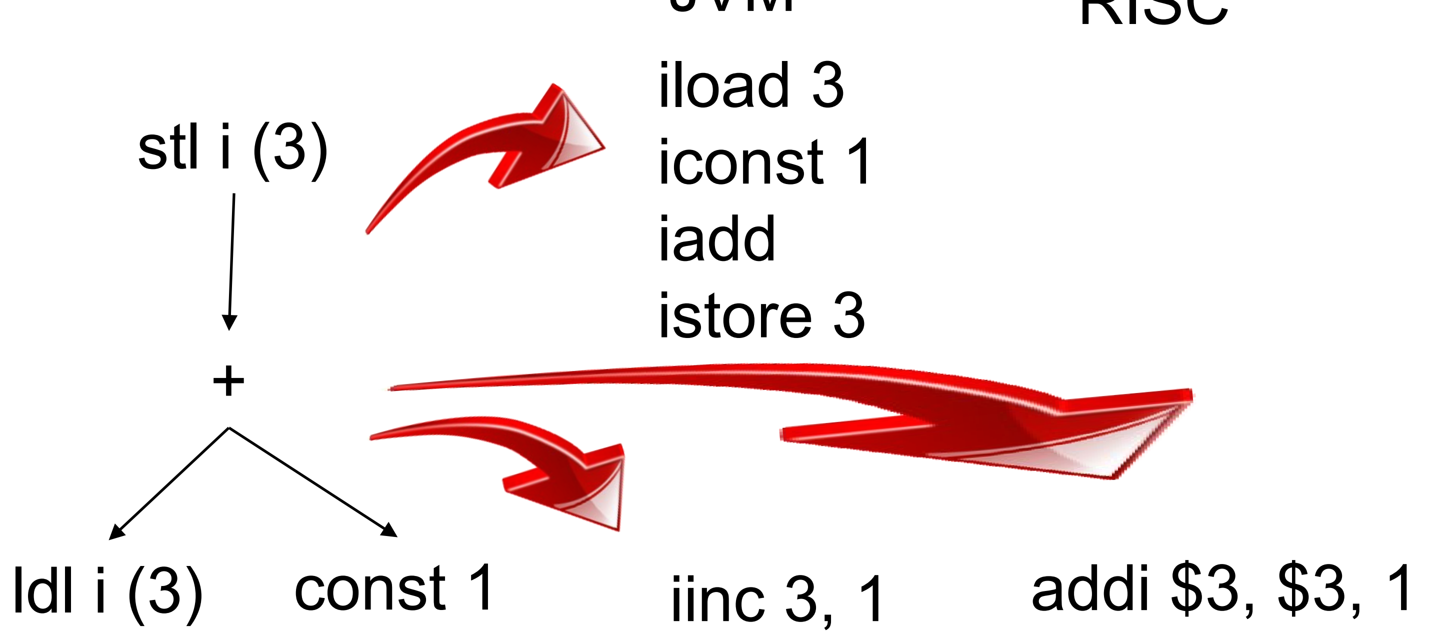
Outline

- Instruction Selection Overview
- Maximal Munch
  - Example
- Dynamic Programming
  - Example
- Other Approaches

Instruction Selection Overview

Problem:

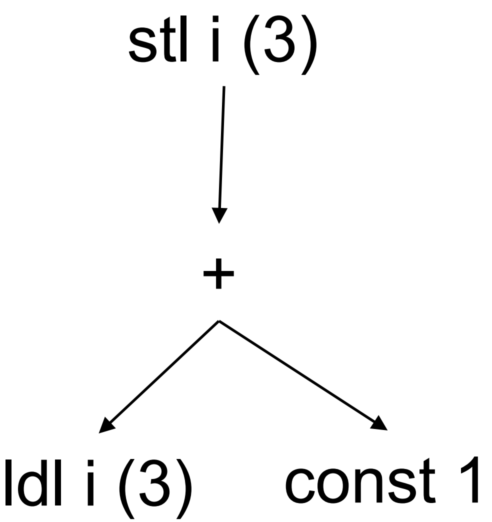
- Find for the operations in the given intermediate representation the appropriate machine instructions

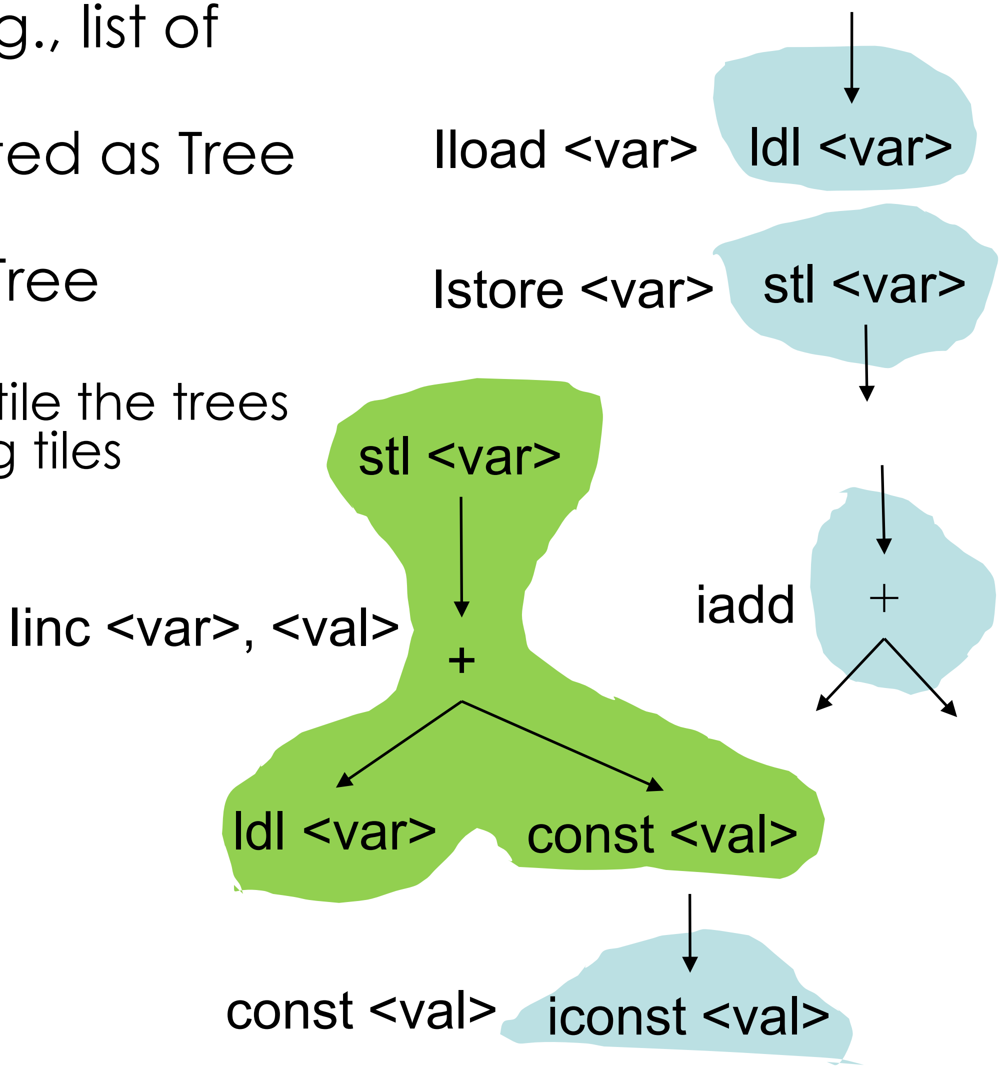


Instruction Selection Overview

From Tree-based IRs (e.g., list of trees):

- Instructions represented as Tree Patterns
- Problem resumes to Tree covering/tiling
  - Completelly Cover /tile the trees with non-overlapping tiles

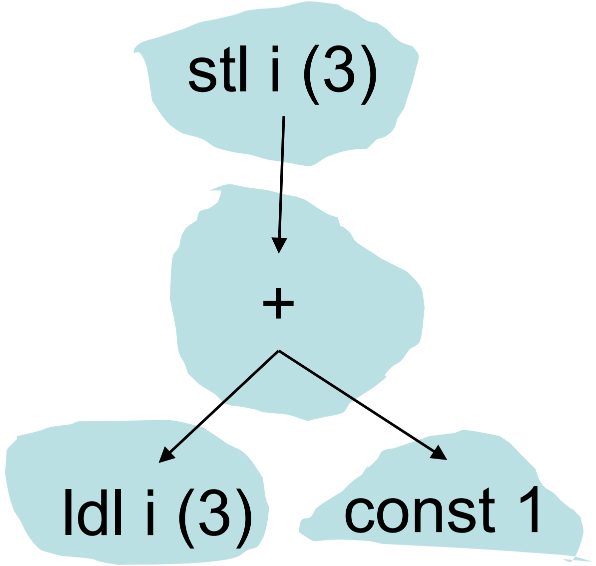


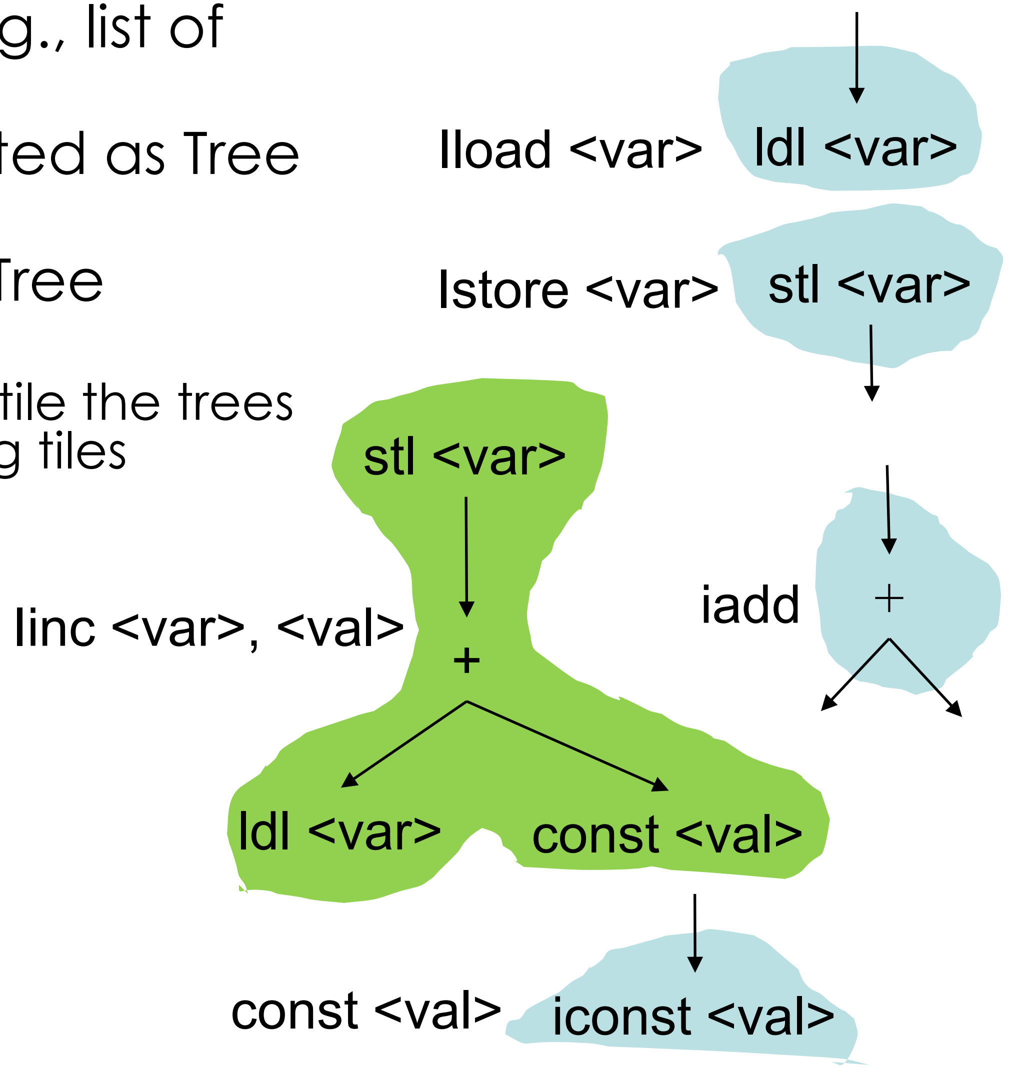


Instruction Selection Overview

From Tree-based IRs (e.g., list of trees):

- Instructions represented as Tree Patterns
- Problem resumes to Tree covering/tiling
  - Completelly Cover /tile the trees with non-overlapping tiles

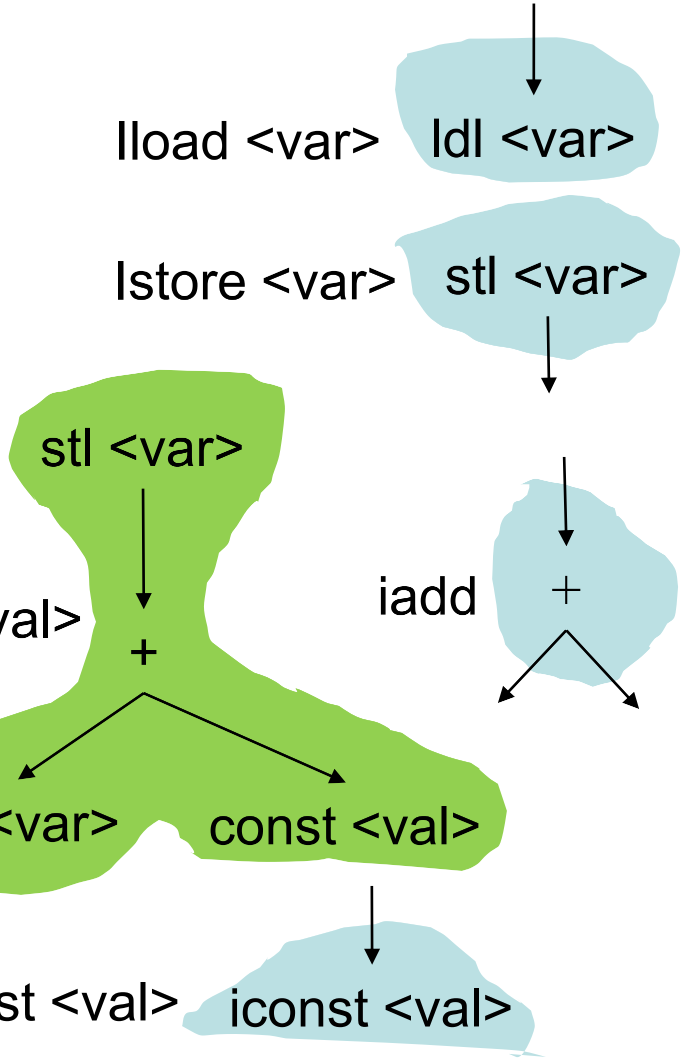


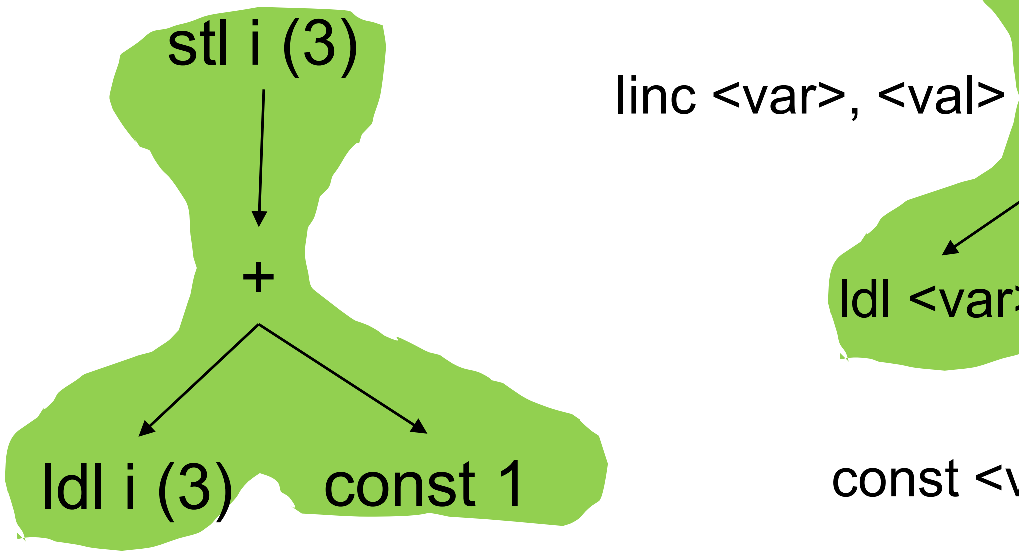


Instruction Selection Overview

From Tree-based IRs (e.g., list of trees):

- Instructions represented as Tree Patterns
- Problem resumes to Tree covering/tiling
  - Completelly Cover /tile the trees with non-overlapping tiles





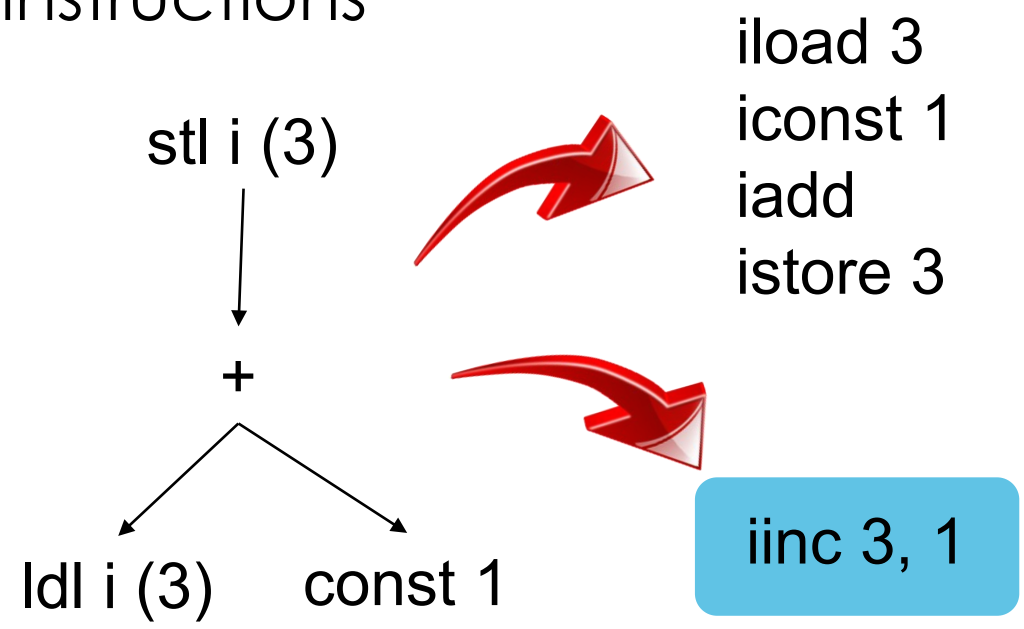
Instruction Selection Overview

Find the best cover/tile

- The one that gives the instruction sequence of least cost

Not always!

- Least cost == the shortest sequence of instructions



Instruction Selection Overview

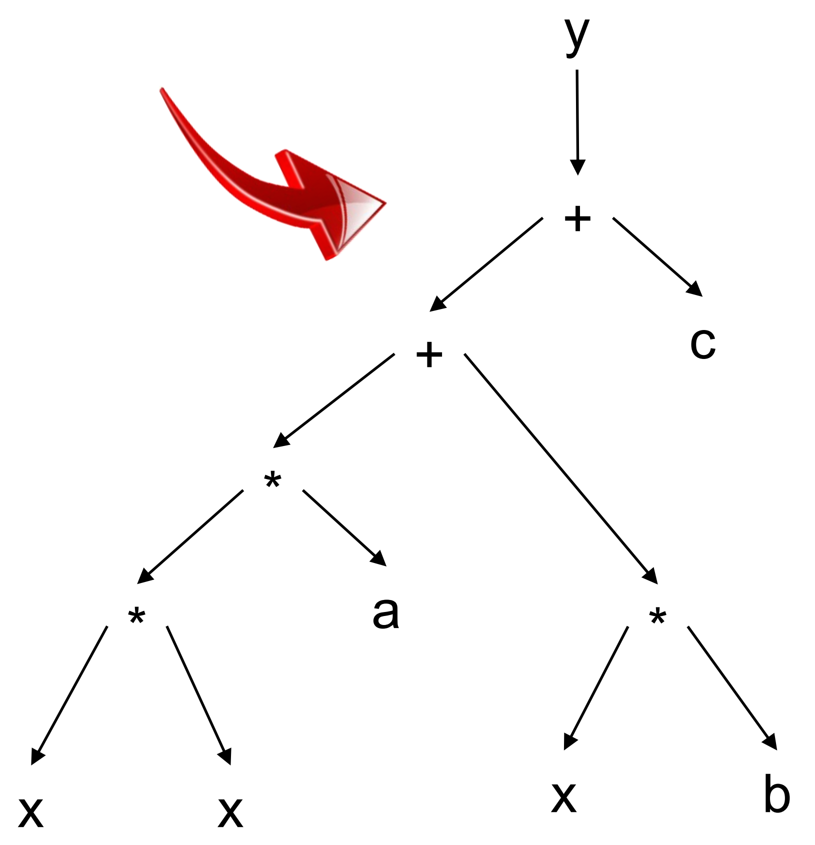
- Each tree pattern can be assigned with a cost
  - Problem is to cover/tile the trees of the program achieving the minimum cost
- However, this simple cost model does not take into account the possible interactions between instructions
- Target machines with reduced instruction set (RISC) have simple tree patterns
  - simple instruction selection algorithms are sufficient

Instruction Selection: Maximal Munch

- A simple algorithm that finds an optimal tiling: Maximal Munch (greedy, top-down pattern match)
  - Starting at the root of the tree
  - Find the largest tile that fits (the tile with most nodes)
  - Cover the root node and the possible nodes with this tile
  - Repeat the algorithm for each subtree of the tile until all the tree is tiled
  - For each tile generates the instructions of that tile
    - code generation is performed in reverse order, least instruction firts

Instruction Selection: Maximal Munch

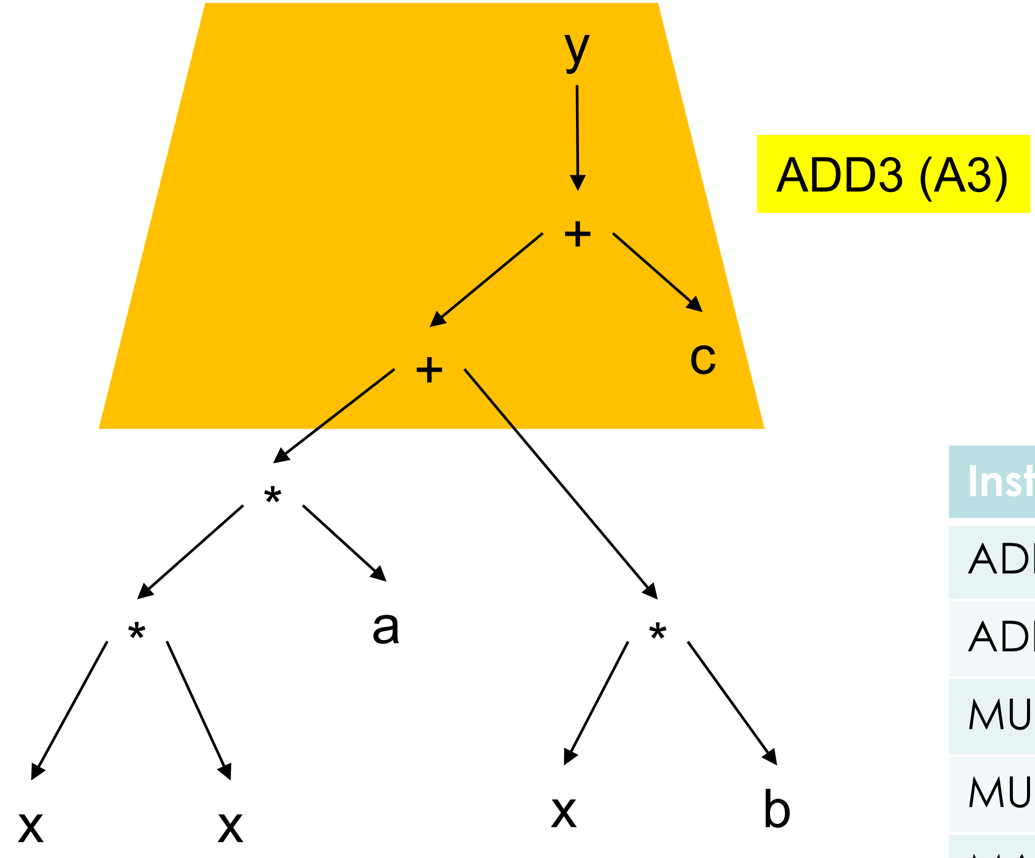
Example: y=a*x*x+b*x+c;



Instruction	function	cost
ADD2 (A2)	a ←b+c	1
ADD3 (A3)	a ←b+c+d	2
MUL2 (M2)	a ←b*c	4
MUL3 (M3)	a ←b*c*d	7
MADD (MA)	a ←b*c+d	4

Instruction Selection: Maximal Munch

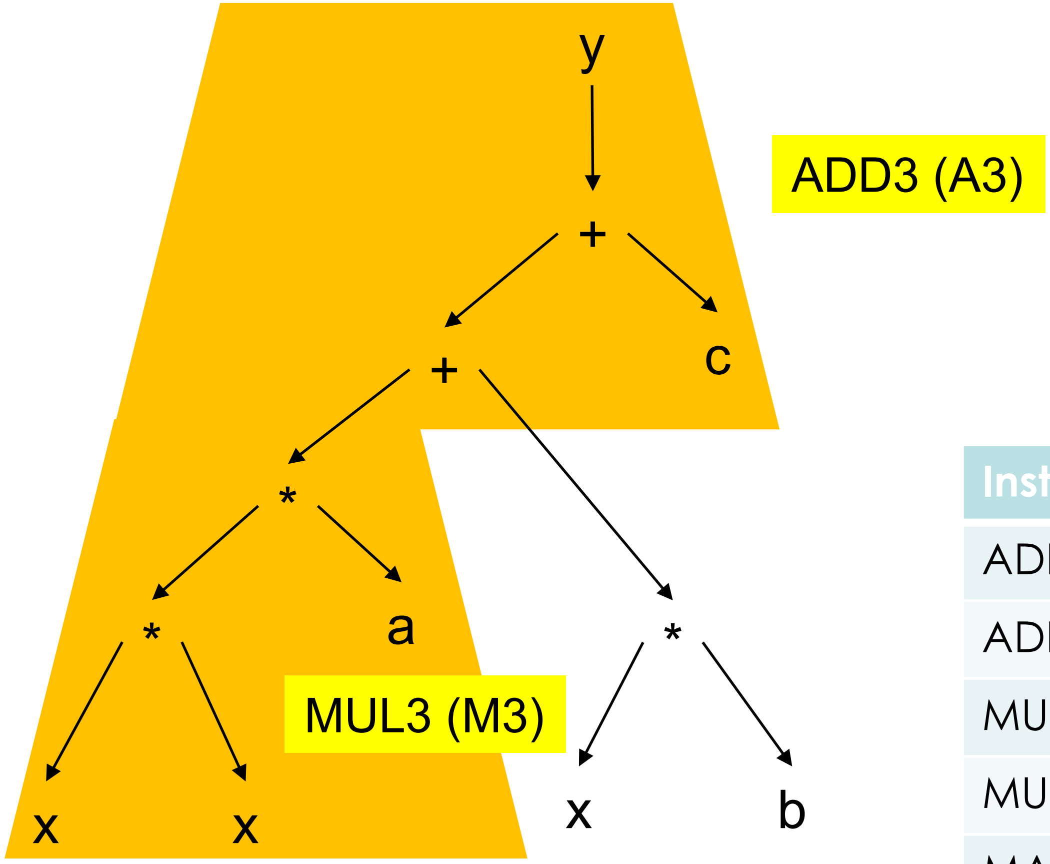
- y=a*x*x+b*x+c;



Instruction	function	cost
ADD2 (A2)	a ←b+c	1
ADD3 (A3)	a ←b+c+d	2
MUL2 (M2)	a ←b*c	4
MUL3 (M3)	a ←b*c*d	7
MADD (MA)	a ←b*c+d	4

Instruction Selection: Maximal Munch

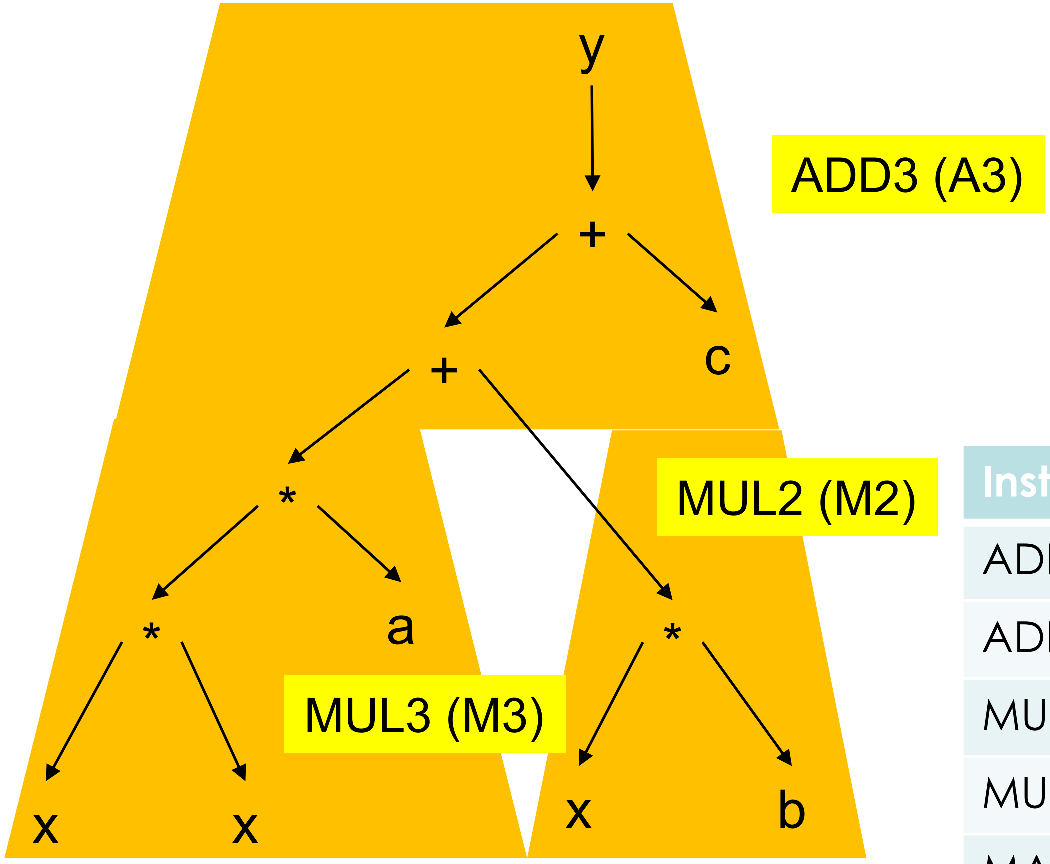
- y=a*x*x+b*x+c;



Instruction	function	cost
ADD2 (A2)	a ←b+c	1
ADD3 (A3)	a ←b+c+d	2
MUL2 (M2)	a ←b*c	4
MUL3 (M3)	a ←b*c*d	7
MADD (MA)	a ←b*c+d	4

Instruction Selection: Maximal Munch

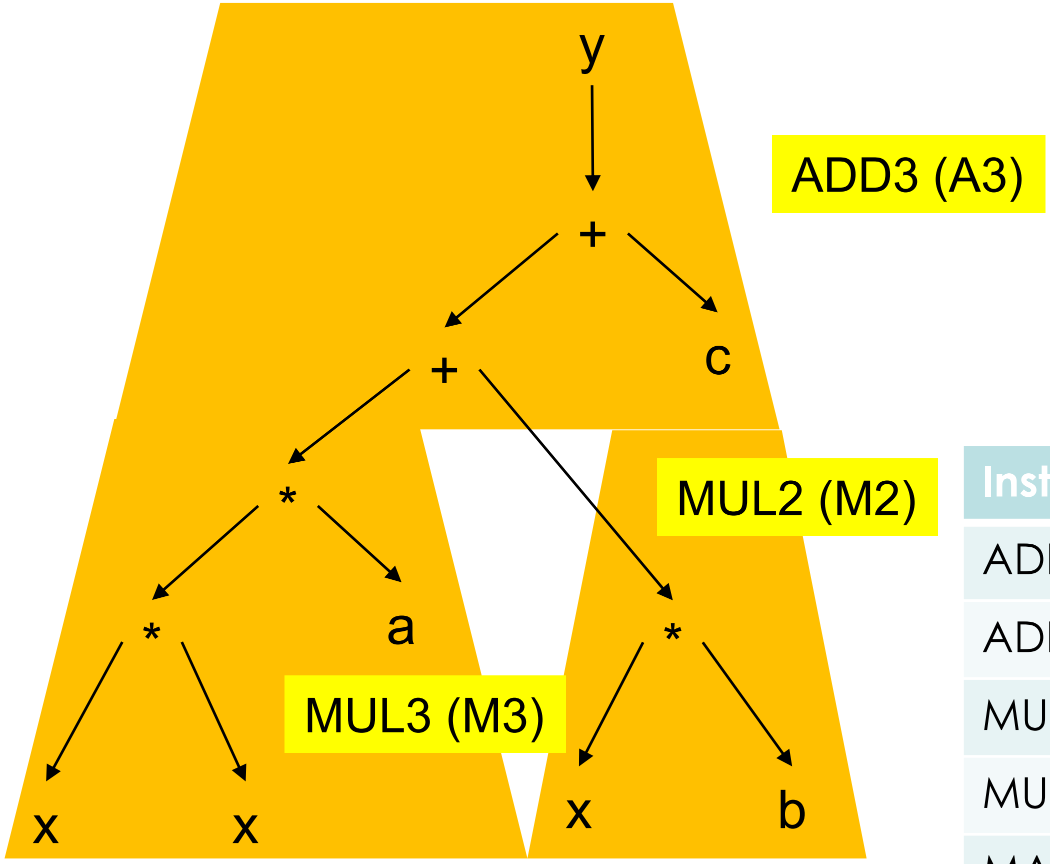
- y=a*x*x+b*x+c;



Instruction	function	cost
ADD2 (A2)	a ←b+c	1
ADD3 (A3)	a ←b+c+d	2
MUL2 (M2)	a ←b*c	4
MUL3 (M3)	a ←b*c*d	7
MADD (MA)	a ←b*c+d	4

Instruction Selection: Maximal Munch

- y=a*x*x+b*x+c;



MUL2 (M2)

MUL3 (M3)

ADD3 (A3)

Cost = 4+7+2
= 13

Instruction	function	cost
ADD2 (A2)	a ←b+c	1
ADD3 (A3)	a ←b+c+d	2
MUL2 (M2)	a ←b*c	4
MUL3 (M3)	a ←b*c*d	7
MADD (MA)	a ←b*c+d	4

Instruction Selection: Maximal Munch

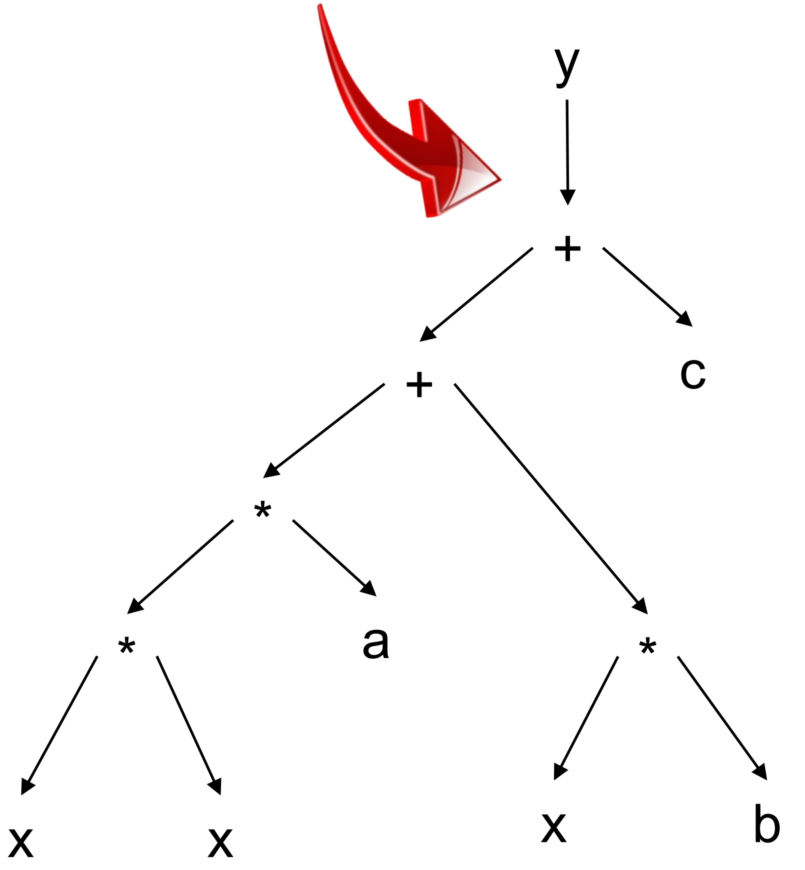
- Maximal Munch does not give the tiling with the minimum cost:
  - It decides locally about the largest pattern to fit, this might prevent the tiling of large patterns in the subtrees
- Gives optimal tiling, i.e., no adjacent tiles can form a tile with lower cost
- One possible solution to achieve minimum cost (i.e., tiling with minimum global cost)
  - Dynamic Programming

Instruction Selection: Dynamic Programming

- Bottom-Up Exhaustive Cataloging of Optimum Solutions
- Optimum Solution of Node Based on Optimum Solution of Subnodes
- Delivers the Global Optimum
- Very Efficient
  - Used in, e.g., Twig, and BURG

Dynamic Programming Example

- y=a*x*x+b*x+c;

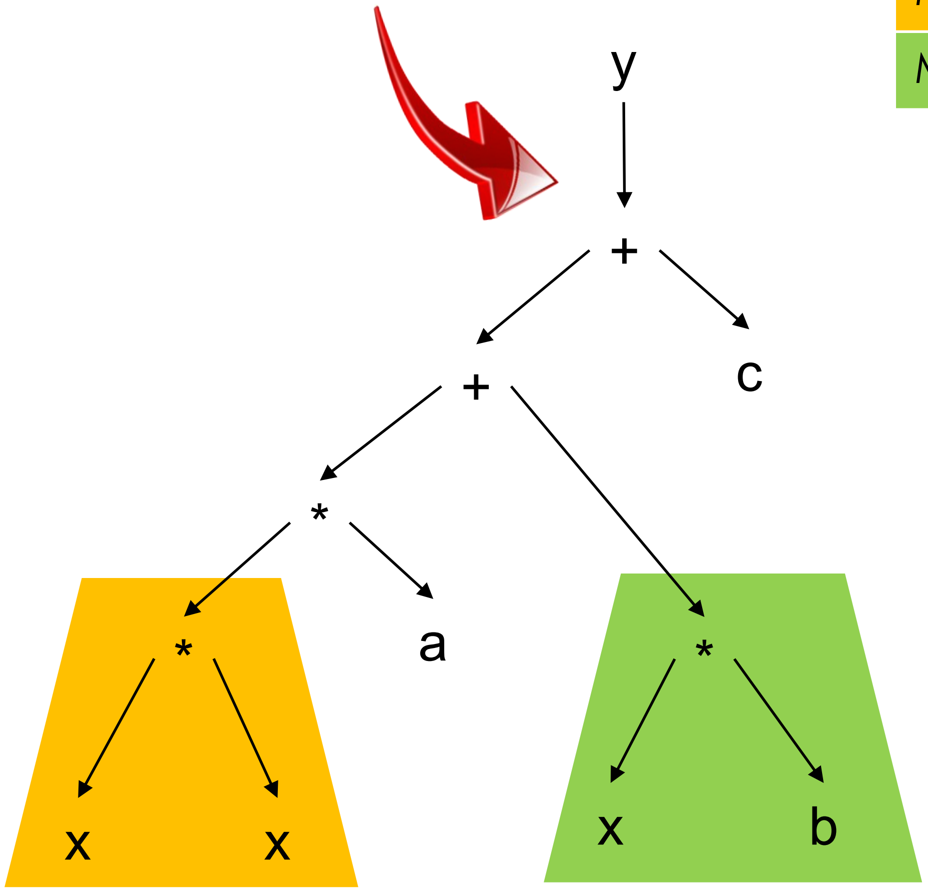


Start at the bottom and tile the first nodes
Select the tiles with minimum costs

Instruction	function	cost
ADD2 (A2)	a ←b+c	1
ADD3 (A3)	a ←b+c+d	2
MUL2 (M2)	a ←b*c	4
MUL3 (M3)	a ←b*c*d	7
MADD (MA)	a ←b*c+d	4

Dynamic Programming Example

- y=a*x*x+b*x+c;



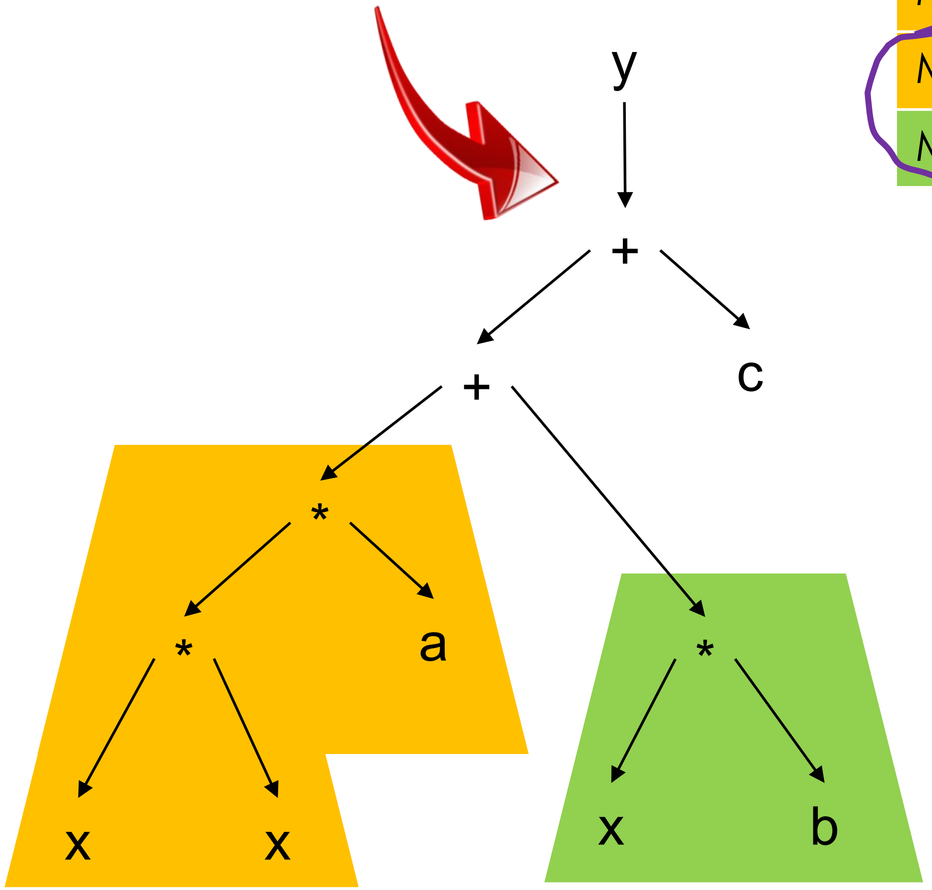
Instructions	Instruction	cost	Leaves cost	total
M2	M2	4	0	4
M2	M2	4	0	4

Go to the next node and tile the subtree with that node as the root

Instruction	function	cost
ADD2 (A2)	a ←b+c	1
ADD3 (A3)	a ←b+c+d	2
MUL2 (M2)	a ←b*c	4
MUL3 (M3)	a ←b*c*d	7
MADD (MA)	a ←b*c+d	4

Dynamic Programming Example

- y=a*x*x+b*x+c;



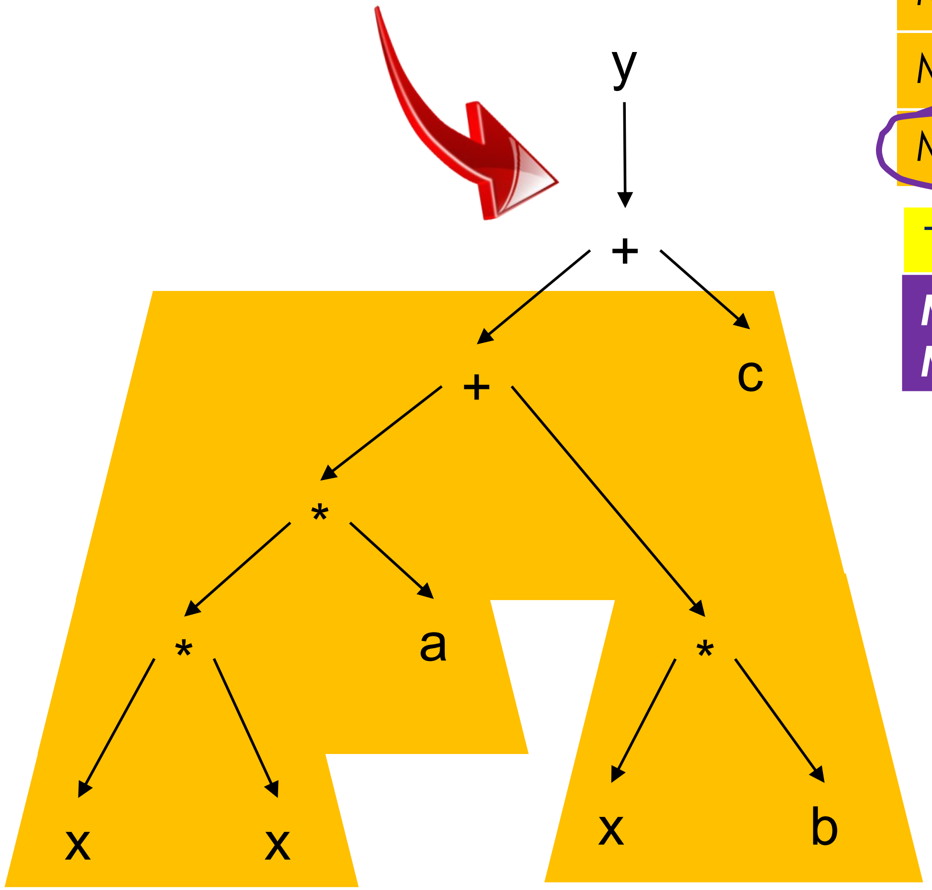
Instructions	Instruction	cost	Leaves cost	total
M2-M2	M2	4	4	8
M3	M3	7	0	7
M2	M2	4	0	4

minimum costs for the subtrees represented as orange and green regions

Instruction	function	cost
ADD2 (A2)	a ←b+c	1
ADD3 (A3)	a ←b+c+d	2
MUL2 (M2)	a ←b*c	4
MUL3 (M3)	a ←b*c*d	7
MADD (MA)	a ←b*c+d	4

Dynamic Programming Example

- y=a*x*x+b*x+c;



Instructions	Instruction	cost	Leaves cost	total
M3-A2, M2	A2	1	11	12
M2-MA, M2	MA	4	8	12
M3-MA	MA	4	7	11

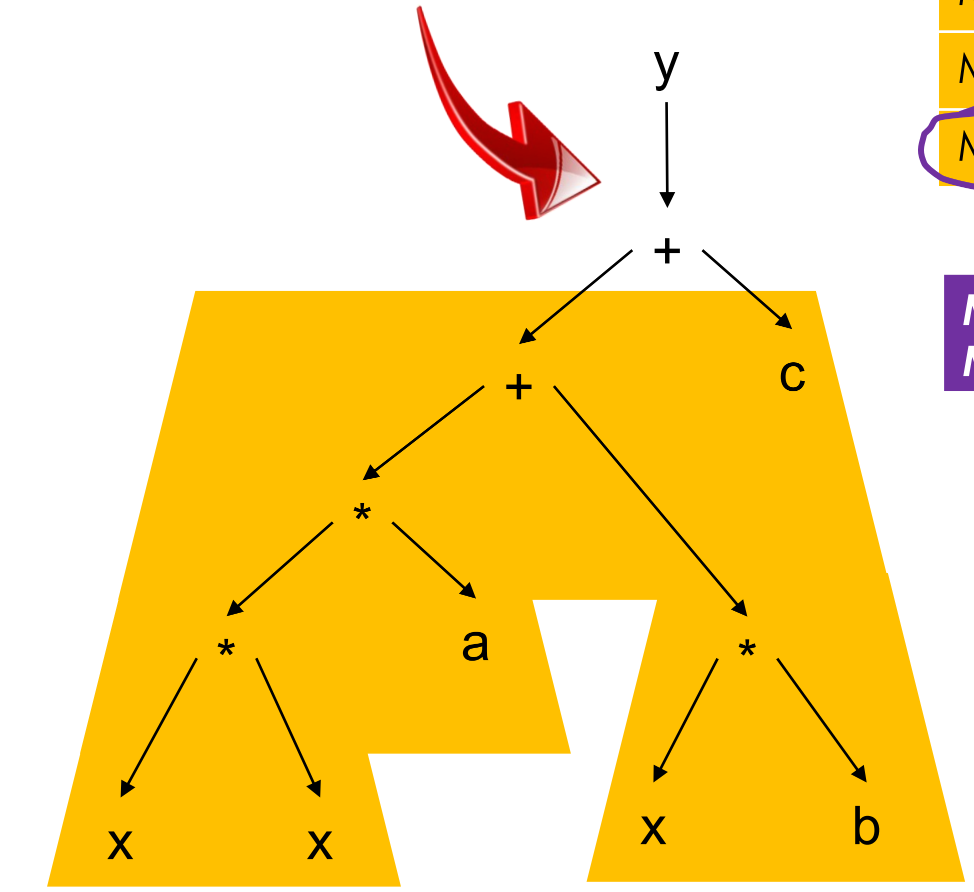
Tile not considered as it uses non-optimal subtree tiles:

M2-M2-A2, M2	**A2**	**1**	**12**	**13**

Instruction	function	cost
ADD2 (A2)	$a \leftarrow b+c$	1
ADD3 (A3)	$a \leftarrow b+c+d$	2
MUL2 (M2)	$a \leftarrow b*c$	4
MUL3 (M3)	$a \leftarrow b*c*d$	7
MADD (MA)	$a \leftarrow b*c+d$	4

Dynamic Programming Example

- y=a*x*x+b*x+c;



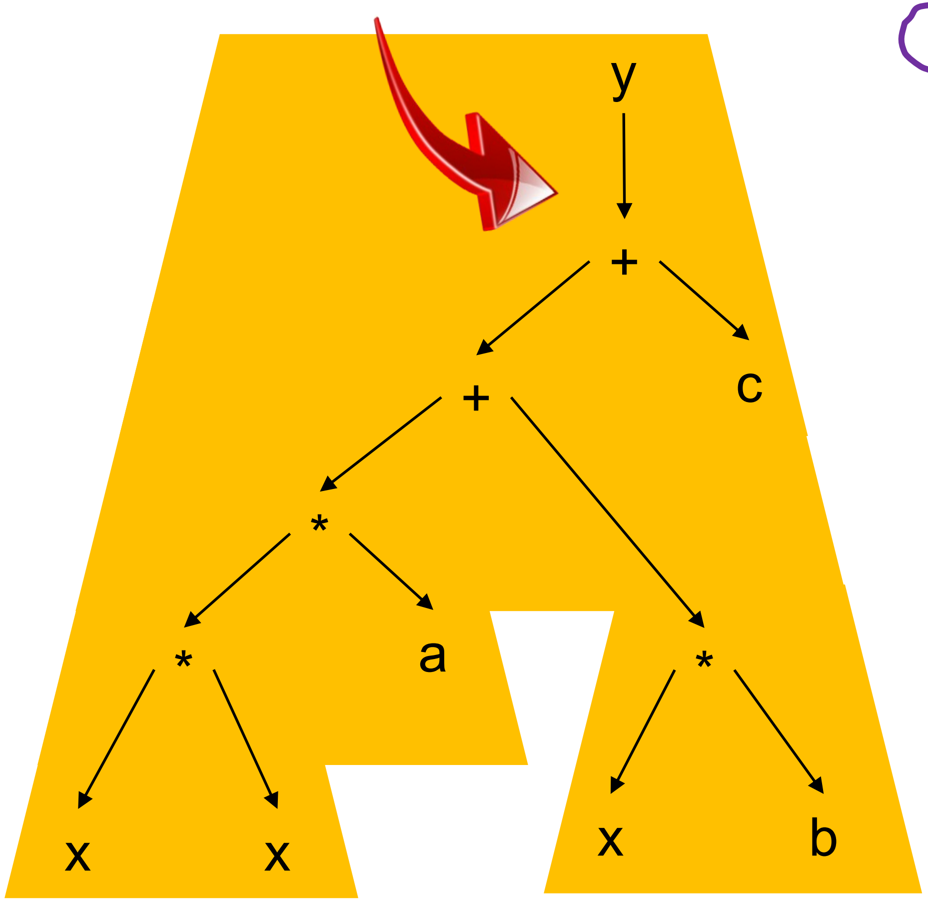
Instructions	Instruction	cost	Leaves cost	total
M3-A2, M2	A2	1	11	12
M2-MA, M2	MA	4	8	12
M3-MA	MA	4	7	11
M2-M2-A2, M2	A2			13

considering the commutativity of the addition in MA

Instruction	function	cost
ADD2 (A2)	a ←b+c	1
ADD3 (A3)	a ←b+c+d	2
MUL2 (M2)	a ←b*c	4
MUL3 (M3)	a ←b*c*d	7
MADD (MA)	a ←b*c+d	4

Dynamic Programming Example

- y=a*x*x+b*x+c;

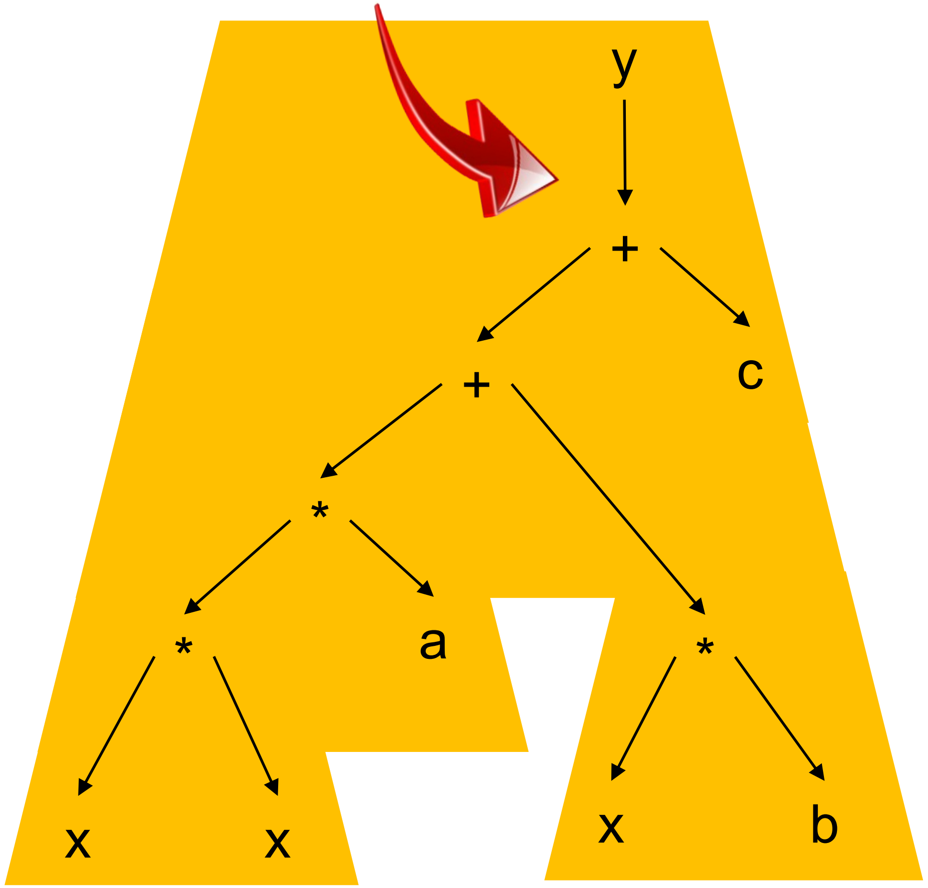


Instructions	Instruction	cost	Leaves cost	total
M3-A3, M2	A3	2	11	13
M3-MA-A2	A2	1	11	12

Instruction	function	cost
ADD2 (A2)	a ←b+c	1
ADD3 (A3)	a ←b+c+d	2
MUL2 (M2)	a ←b*c	4
MUL3 (M3)	a ←b*c*d	7
MADD (MA)	a ←b*c+d	4

Dynamic Programming Example

- y=a*x*x+b*x+c;



Tiles not considered as they use non-optimal subtree tiles:

M2-M2-A2-A2, M2	A2	1	13	14
M2-M2-A3, M2	A3	2	12	14
M3-A2-A2, M2	A2	1	12	13
M2-MA-A2, M2	A2	1	12	13
M2-M2-A2-A2, M2	A2	1	13	14

Instruction	function	cost
ADD2 (A2)	$a \leftarrow b+c$	1
ADD3 (A3)	$a \leftarrow b+c+d$	2
MUL2 (M2)	$a \leftarrow b*c$	4
MUL3 (M3)	$a \leftarrow b*c*d$	7
MADD (MA)	$a \leftarrow b*c+d$	4

Other Approaches

- Graham-Glanville Parser-Based Approach
- Naive/Canonical Generation
  - Transform each node in the equivalent sequence of machine instructions
  - Can be followed by Peephole optimization

A. Aho, M. Lam, R. Sethi, J. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd Edition, Addison Wesley, 2007

EXERCISE

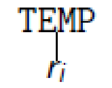
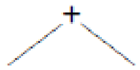
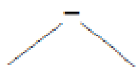
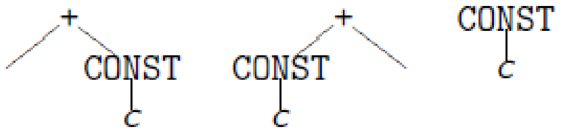
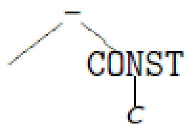
Exercise

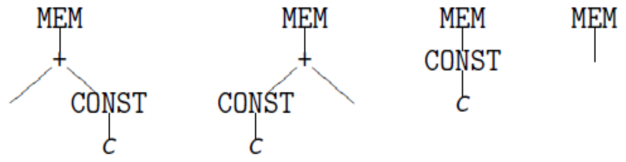
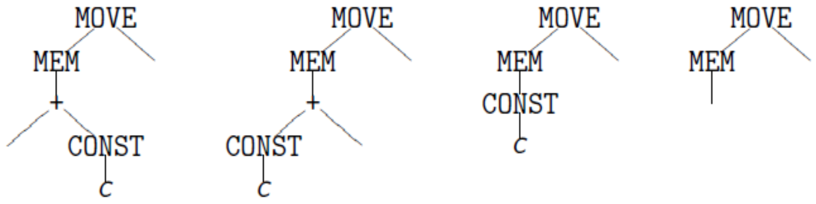
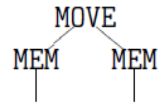
- Consider a microprocessor with the following instructions:
  - ADD rd = rs1 + rs2
  - ADDI rd = rs + c
  - SUB rd = rs1 − rs2
  - SUBI rd = rs − c
  - MUL rd = rs1* rs2
  - DIV rd = rs1/rs2
  - LOAD rd = M[rs + c]
  - STORE M[rs1 + c] = rs2
  - MOVEM M[rs1] = M[rs2]
- Where rd, rs identify registers of the architecture (from r0 to r31 and r0 stores the non-modified value 0) and c identifies literals

This exercise is based on the Andrew W. Appel “Tiger” book.

Exercise

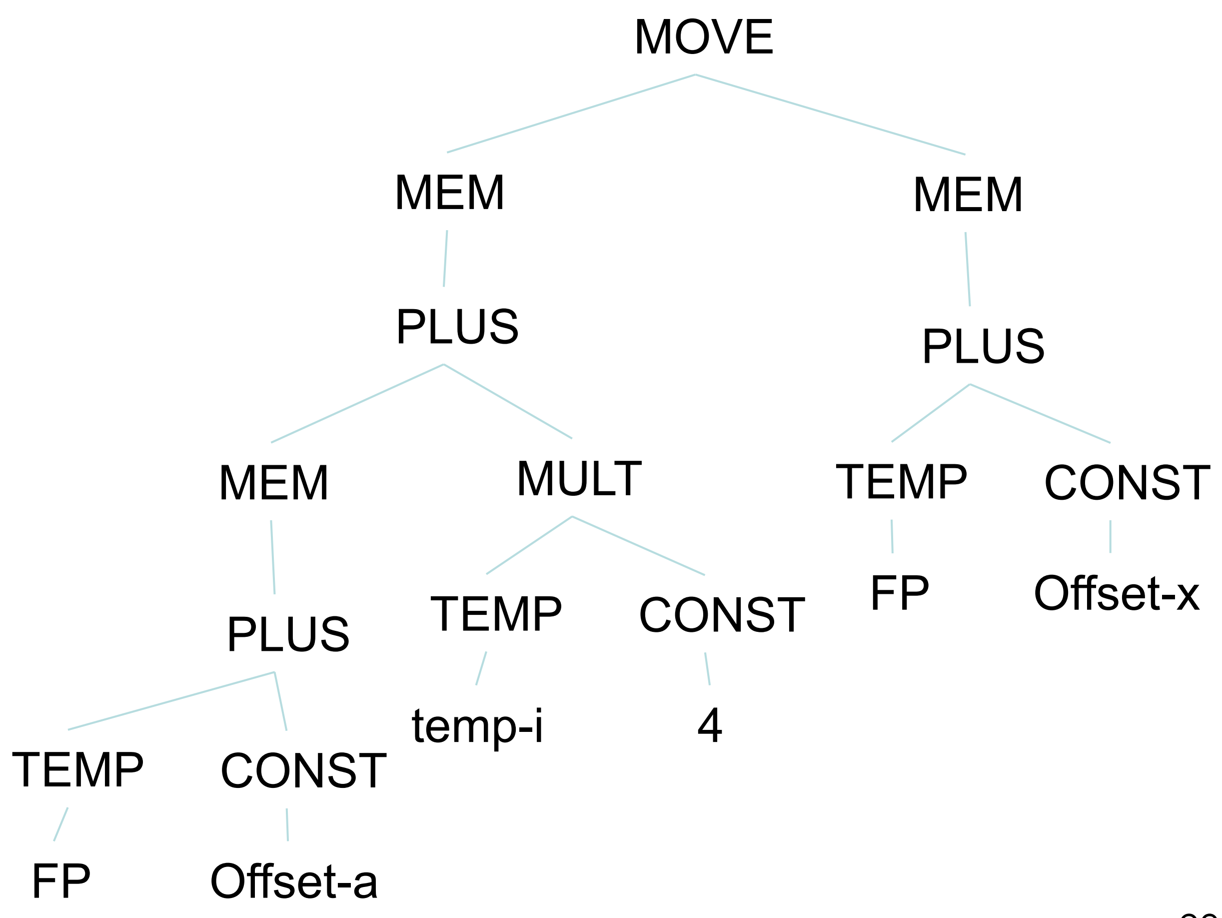
➢ The corresponding Instruction Tree Patterns are the following:

Instruction	Effect	IR Tree Pattern
—	r_i	
add	$r_i \leftarrow r_j + r_k$	
mul	$r_i \leftarrow r_j * r_k$	
sub	$r_i \leftarrow r_j - r_k$	
div	$r_i \leftarrow r_j / r_k$	
addi	$r_i \leftarrow r_j + c$	
subi	$r_i \leftarrow r_j - c$	

Instruction	Effect	IR Tree Pattern
load	$r_i \leftarrow M[r_j + c]$	
store	$M[r_j + c] \leftarrow r_i$	
movem	$M[r_j] \leftarrow M[r_i]$	

Exercise

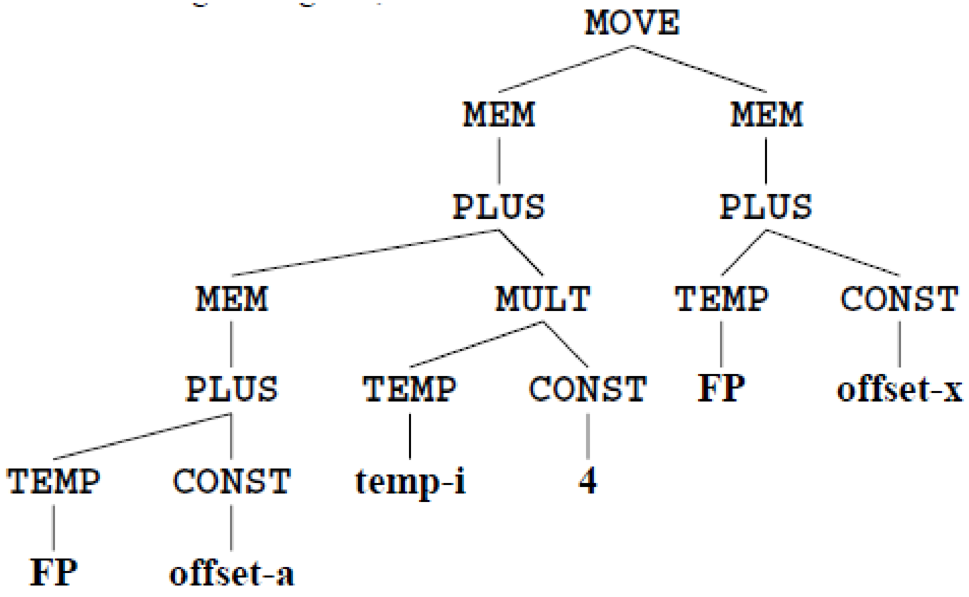
- Consider the input intermediate representation illustrated below for the statement: a[i] = x; (assuming *i* stored in a register identified by *r_i*, and *a* and x are frame residents), where *FP* represents the register with the frame pointer, *offset-a* and *offset*-x represent two constants, and *temp-i* identifies the variable *i*.



Exercise

a) Use individual node selection to generate the assembly instructions.
b) Use the Maximal-Munch algorithm for instruction selection and write the instructions generated.
c) Use dynamic programming to obtain an optimum solution for instruction selection (considering as goal the minimum number of instructions) and write the instructions generated.

- Consider the input intermediate representation illustrated below for the statement: a[i] = x; (assuming *i* stored in a register identified by *r_i*, and *a* and x are frame residents), where *FP* represents the register with the frame pointer, *offset-a* and *offset-x* represent two constants, and *temp-i* identifies the variable *i*.



Instruction	Effect	IR Tree Pattern
load	$r_i \leftarrow M[r_j + c]$	MEM + CONST c; MEM + CONST c; MEM CONST c; MEM
store	$M[r_j + c] \leftarrow r_i$	MOVE MEM + CONST c; MOVE MEM + CONST c; MOVE MEM CONST c; MOVE MEM
movem	$M[r_j] \leftarrow M[r_i]$	MOVE MEM MEM

Instruction	Effect	IR Tree Pattern
—	r_i	TEMP r_i
add	$r_i \leftarrow r_j + r_k$	+
mul	$r_i \leftarrow r_j * r_k$	*
sub	$r_i \leftarrow r_j - r_k$	−
div	$r_i \leftarrow r_j / r_k$	/
addi	$r_i \leftarrow r_j + c$	+ CONST c; CONST + c; CONST c
subi	$r_i \leftarrow r_j - c$	− CONST c