编译原理

9. 指令选择

rainoftime.github.io

浙江大学

计算机科学与技术学院

Content

Overview of IR → Machine Code

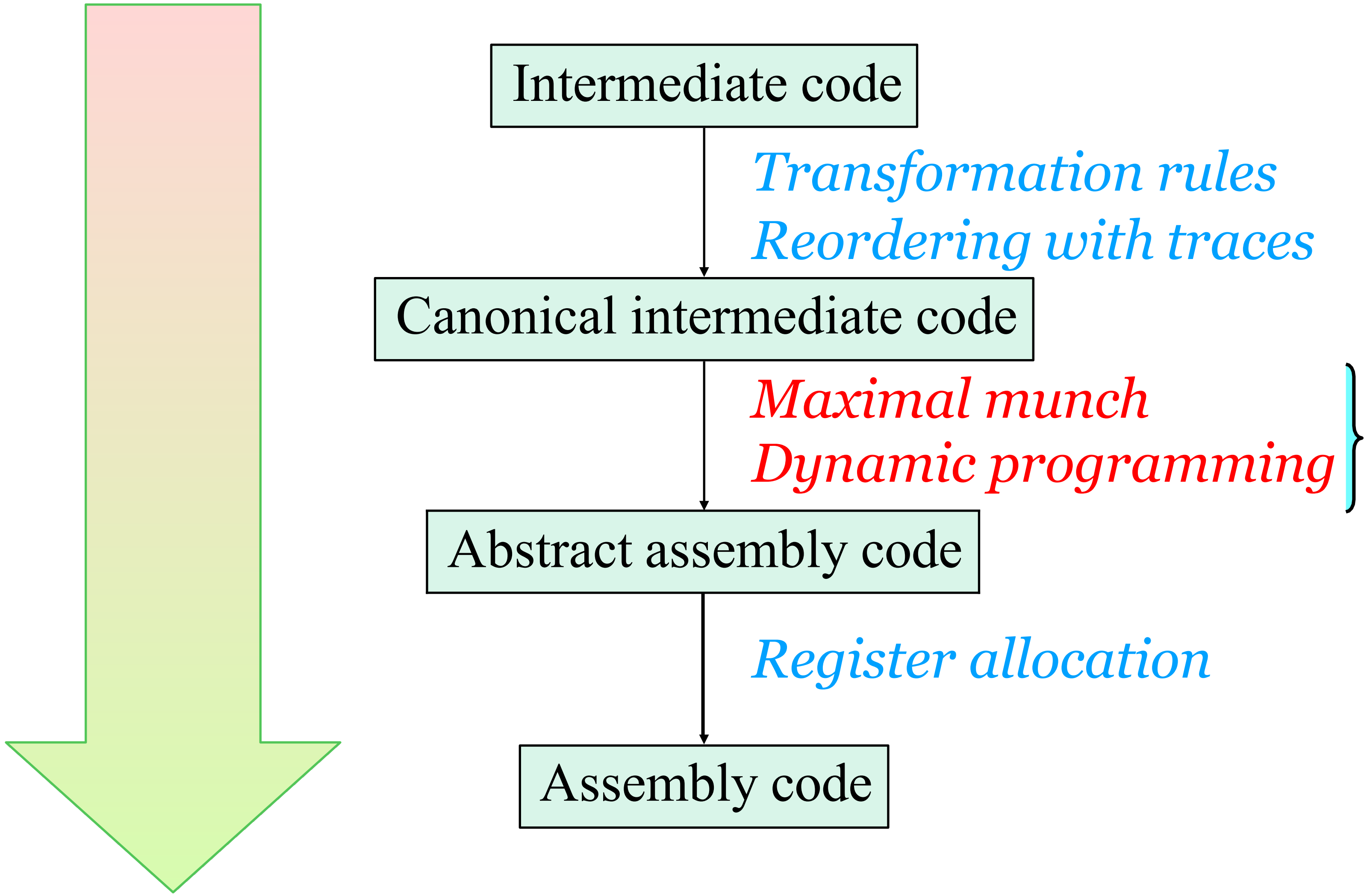
- **Step #1** : Transform the IR trees into **a list of canonical trees**
  - a. eliminate SEQ and ESEQ nodes
  - b. the arguments of a CALL node should never be other CALL nodes

- **Step #2** : Rearrange the **canonical trees** (into **traces**) so that every CJUMP(cond,lt,lf) is immediately followed by LABEL(lf)

- **Step #3** : **Instruction Selection** --- generate the pseudo-assembly code from the canonical trees in the step #2

- **Step #4** : Perform **register allocations** on pseudo-assembly code

Where We Are


Intermediate code
Transformation rules
Reordering with traces
Canonical intermediate code
Maximal munch
Dynamic programming
Abstract assembly code
Register allocation
Assembly code

本讲内容

1 指令选择概述

2 指令选择算法

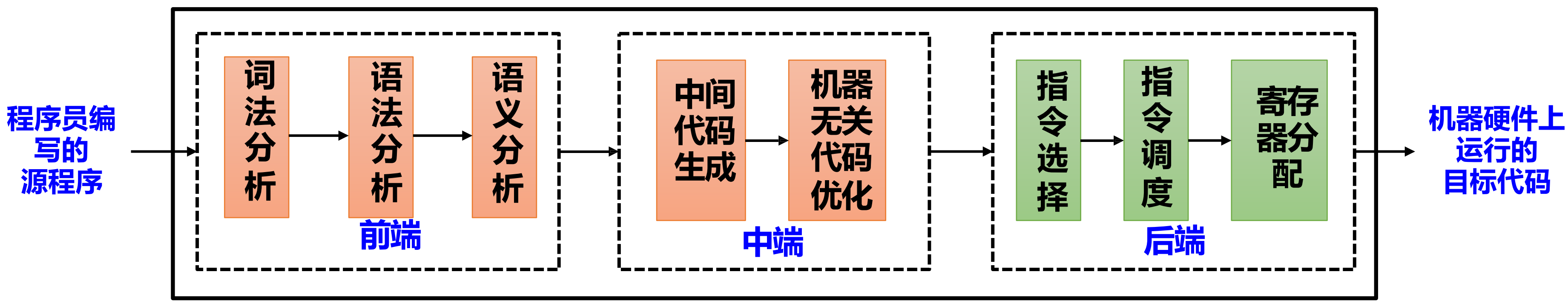
3 CISC vs RISC

1. 指令选择概述

- 指令选择问题
- 基于树覆盖的指令选择
- Optimal and Optimum Tilings

Compiler Organization

- **现代编译器的典型架构**



infrastructure – symbol tables, trees, graphs, etc

编译后端: 目标代码生成

- **指令选择 Instruction Selection**
  Mapping *IR* into abstract assembly code

- **寄存器分配 Register Allocation**
  Deciding which values will reside in registers

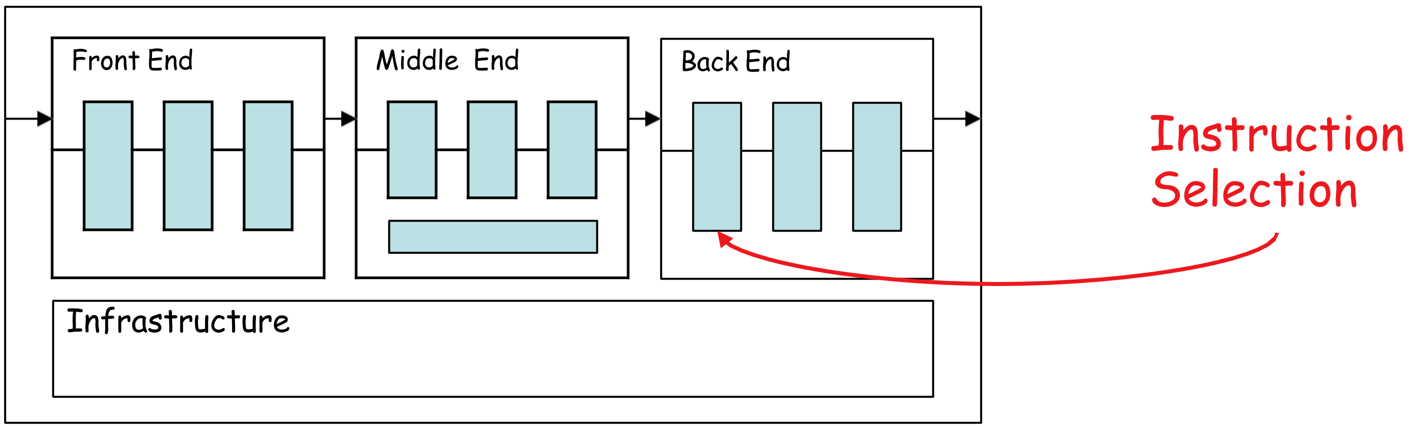
- **指令调度Instruction Scheduling (本课程不讲)**
  Reordering operations to hide latencies and exploit intraprocessor parallelism.

Big Picture

- Compiler consists of lots of fast stuff followed by hard problems
  - Scanner: O(n); Parser: O(n); Analysis: O(n), $O(n^3)$, …
  - **Instruction selection**:
    - Fast or NP-Complete
  - **Instruction scheduling**:
    - Simple basic block: quick heuristics
    - General problem: NP-Complete
  - **Register allocation**:
    - Linear or NP-Complete

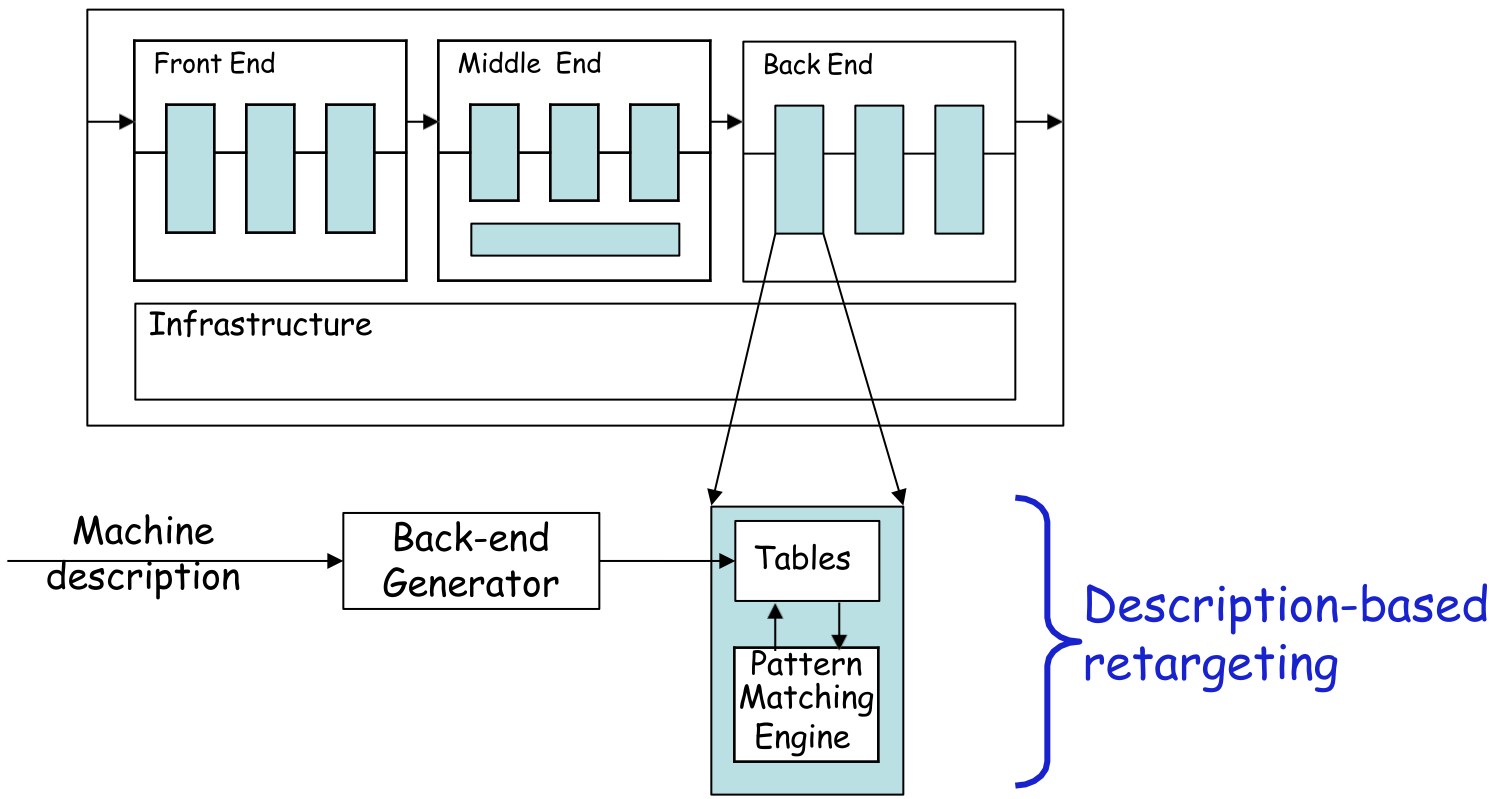
Instruction Selection: The Problem

- **Mapping *IR* into abstract assembly code**
- Abstract assembly = assembly with infinite registers
  - Invent new temporaries for intermediate results
  - Map to actual registers later



Instruction Selection: The Problem

Want to automate generation of instruction selectors

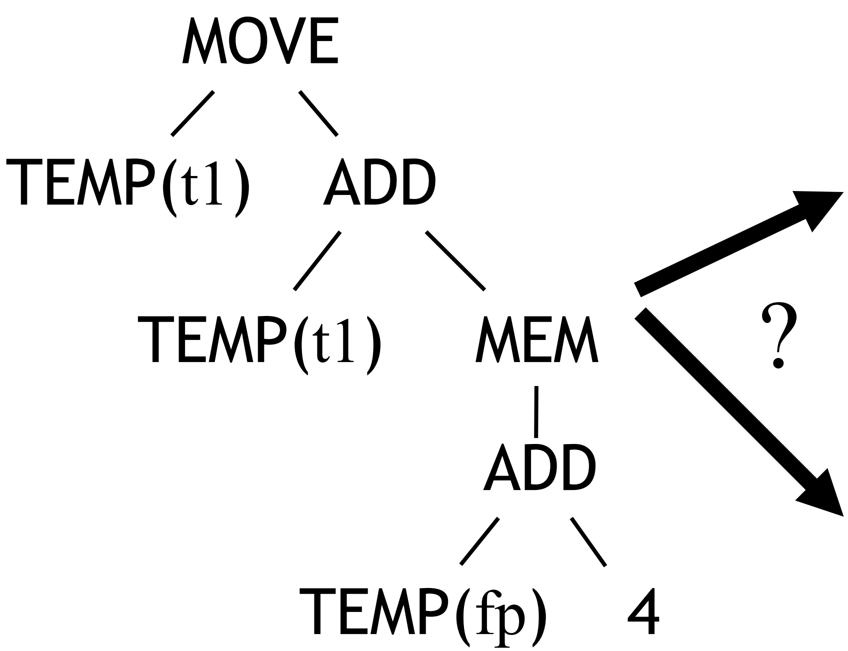


E.g., LLVM支持用tblgen描述后端, 定义寄存器、指令集、调用约定等

Instruction Selection Criteria

- More than one way to translate a given statement.
- How to choose?

MOVE(TEMP(t1), TEMP(t1) + MEM(TEMP(FP)+4))




```
mov t2, rbp
add t2, 4
mov r3, [t2]
add t1, t3
```

```
add t1, [rbp + 4]
```

Instruction Selection Criteria

- Instruction selection techniques
  - Must produce good code
  - Must run quickly

(*some metric for good*)

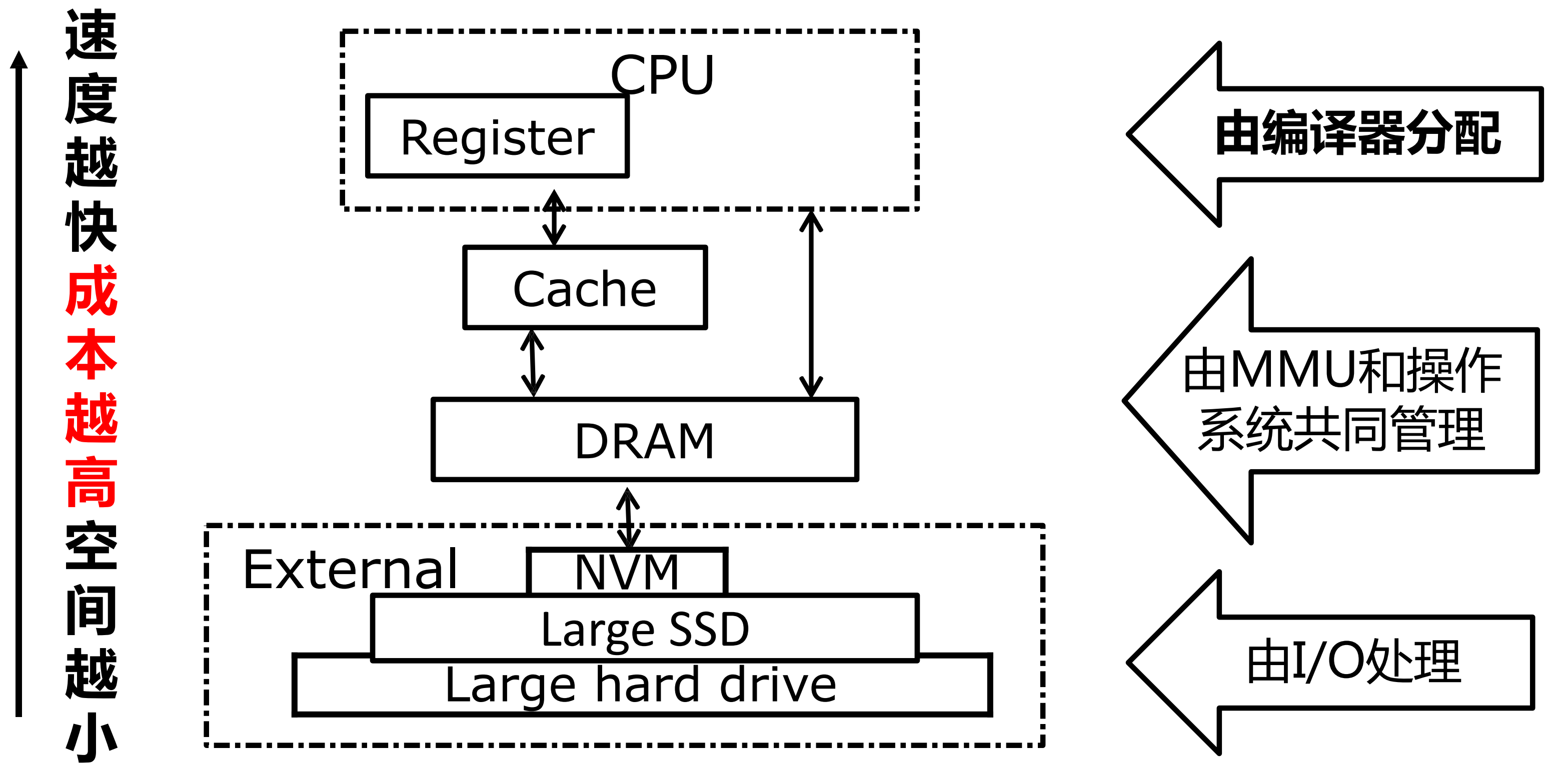
- Several metrics for "good"
  - Fastest
  - Smallest
  - Minimize power consumption

- Sometimes not obvious

Instruction Selection Criteria

- Guide inst. selection via metrics for “good”

需考虑指令代价、运算对象和结果如何存储等多重因素



Instruction Selection Implementation

Use **pattern matching techniques** to pick machine instructions that match fragments of the program IR

- **Tree-oriented IR** suggests pattern matching on trees
  - Tree-patterns as input, matcher as output
  - E.g., Dynamic programming-based matching

- **Linear IR** suggests using some sort of string matching
  - Strings as input, matcher as output
  - E.g., Text matching, peephole matching, etc.

In practice, both work well; matchers are usually quite different

1. 指令选择概述

- □ 指令选择问题
- □ 基于树覆盖的指令选择
- □ Optimal and Optimum Tilings

The Jouette Instruction Set

- To illustrate instruction selection, we use a simple instruction set: the Jouette architecture

Name	*Effect*	
—	r_i	
ADD	r_i	$r_j + r_k$
MUL	r_i	$r_j \times r_k$
SUB	r_i	$r_j \quad r_k$
DIV	r_i	r_j / r_k
ADDI	r_i	$r_j + c$
SUBI	r_i	$r_j \quad c$
LOAD	r_i	$M[r_j + c]$

Name	*Effect*	
—	M[]	
STORE	$M[r_j + c]$	r_i
MOVEM	$M[r_j]$	$M[r_i]$

The Jouette Instruction Set

- RISC-style, load/store architecture
- Relative large, general purpose register file
  - Data or address can reside in registers
  - Each instruction can access any register
- r_0 always contains zero
- Each instruction has latency of one cycle
  - Except MOVEM
- Execution of only one instruction per cycle

Example: The Jouette Architecture

- Arithmetic

ADD	$r_d = r_{s1} + r_{s2}$
ADDI	$r_d = r_s + c$
SUB	$r_d = r_{s1} - r_{s2}$
SUBI	$r_d = r_s - c$
MUL	$r_d = r_{s1} * r_{s2}$
DIV	$r_d = r_{s1} / r_{s2}$

- Memory

LOAD	$r_d = M[r_s + c]$
STORE	$M[r_{s1} + c] = r_{s2}$
MOVEM	$M[r_{s1}] = M[r_{s2}]$

Instruction Selection as Macro Expansion

- **宏展开/模版匹配**
  - 对于每条IR，有一条或多条机器指令与其相对应
  - 使用预制好的指令模板替换对应的每一条输入IR

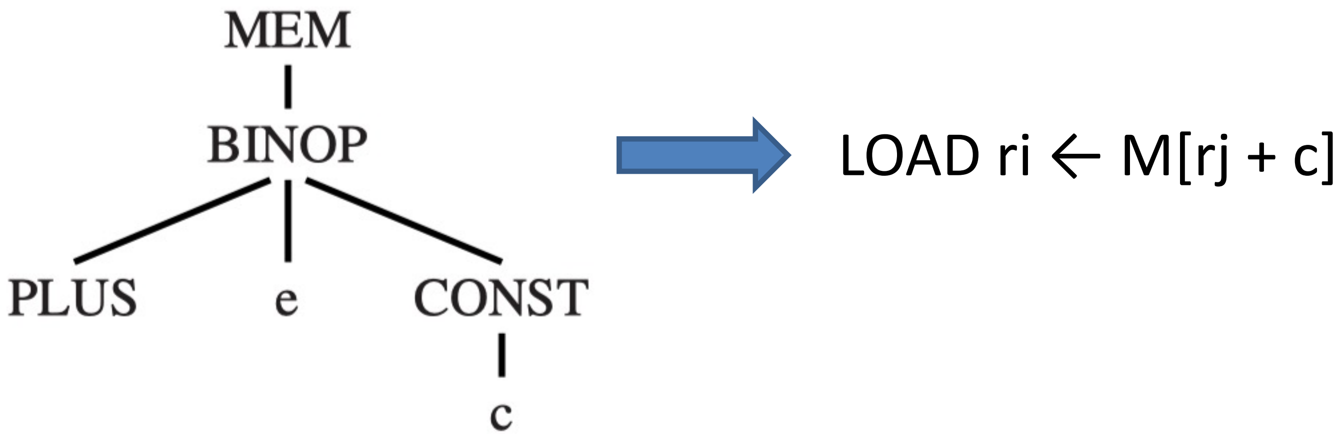
- **优点**：实现简单，易于理解
- **缺点**：通常只支持1:1或1:N的情况，难以处理N:1或N:M的场景(多条IR对应一条或者多条机器指令)
  - 往往导致生成的指令比较低效

Instruction Selection via Tree Patterns

- Finding set of machine instructions
  - that implement operations specified in IR tree.
- **Tree pattern**
  - Each machine instruction can be specified as an IR tree fragment
  - A tree pattern is also called a tile
- **Goal of instruction selection**
  - **Tiling**: **cover the IR tree** (of a program fragment) with a set of non-overlapping tree patterns

Example: Tree Patterns

- Each machine instruction can be expressed a fragment of an IR tree, called a tree pattern

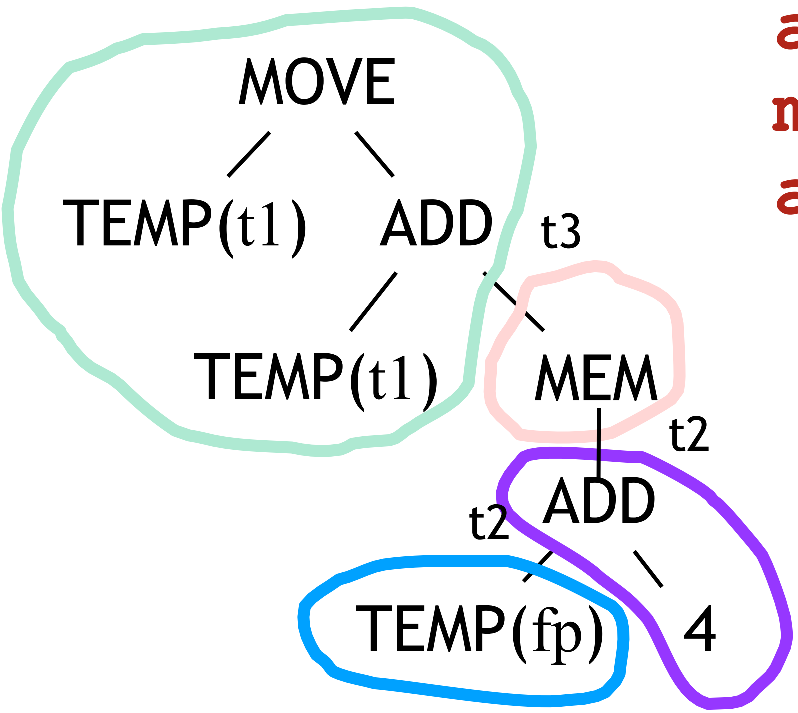


- The Tree language expresses only one operation in each tree node: Fetch, store, addition,...
- A real machine instruction can often perform **several of primitive operations.**

将IR与后端的机器指令**都转换为树结构**。这样就把指令选择问题转换为机器指令树覆盖全IR Tree的问题。

Example: Tiling

- **Idea**: each machine instruction performs computation for a piece of the IR tree: a tile

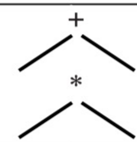
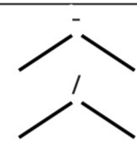
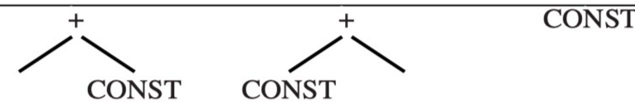
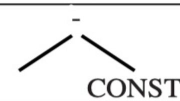
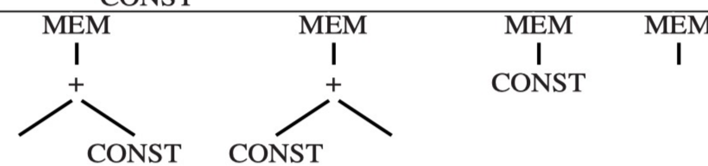
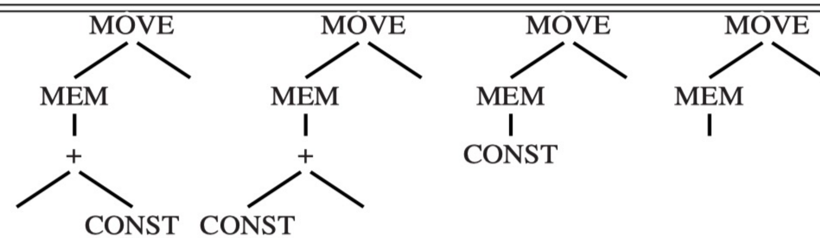
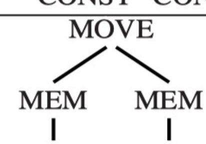



```
mov t2, rbp
add t2, 4
mov t3, [t2]
add t1, t3
```

"X86" instructions

- **Tiles connected by new temporary registers (t2, t3) that hold result of tile**

Tree Patterns of Jouette

Name	*Effect*	*Trees*
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	
MUL	$r_i \leftarrow r_j \times r_k$	
SUB	$r_i \leftarrow r_j - r_k$	
DIV	$r_i \leftarrow r_j / r_k$	
ADDI	$r_i \leftarrow r_j + c$	
SUBI	$r_i \leftarrow r_j - c$	
LOAD	$r_i \leftarrow M[r_j + c]$	
STORE	$M[r_j + c] \leftarrow r_i$	
MOVEM	$M[r_j] \leftarrow M[r_i]$	

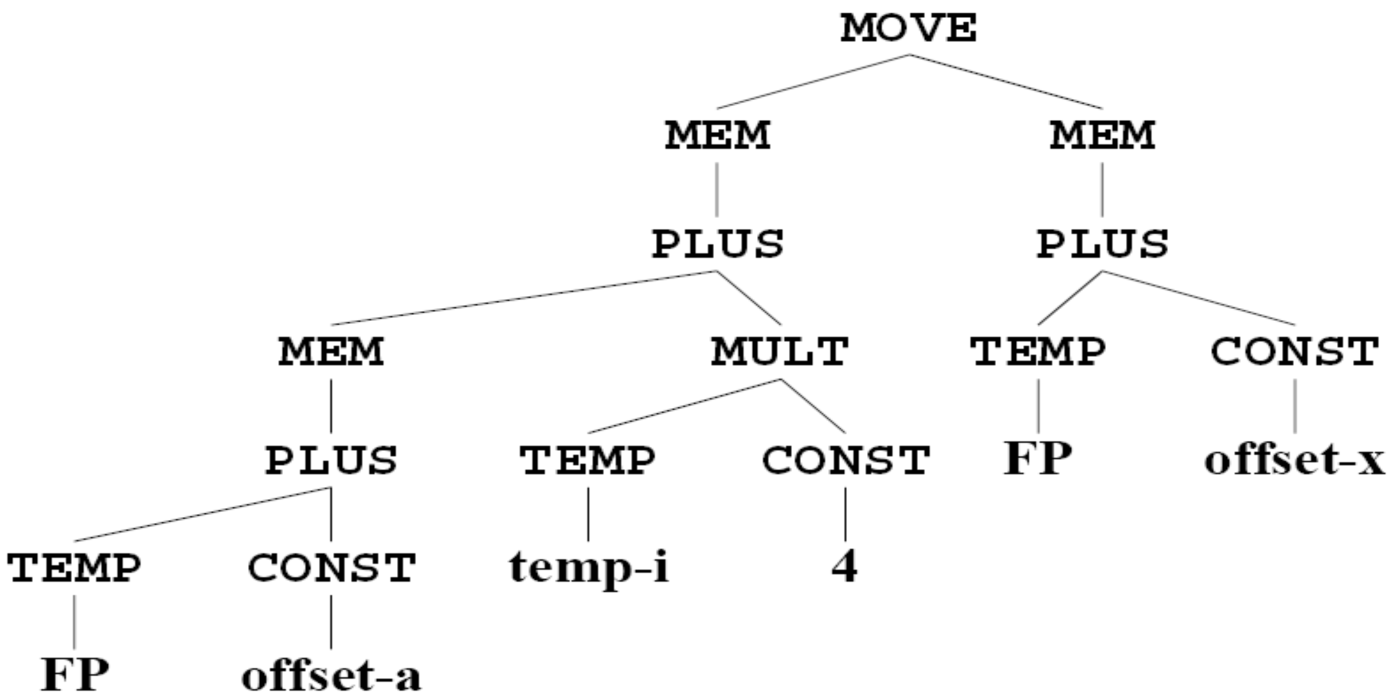
The actual values of CONST and TEMP nodes will not always be shown.

Some instructions correspond to more than one tree pattern

Example: Tiling via Jouette Tree Patterns

- **a[i] := x**, assuming i in register, a and x in stack frame

*M[a+i*4]:=x:* each element takes up 4 storage locations

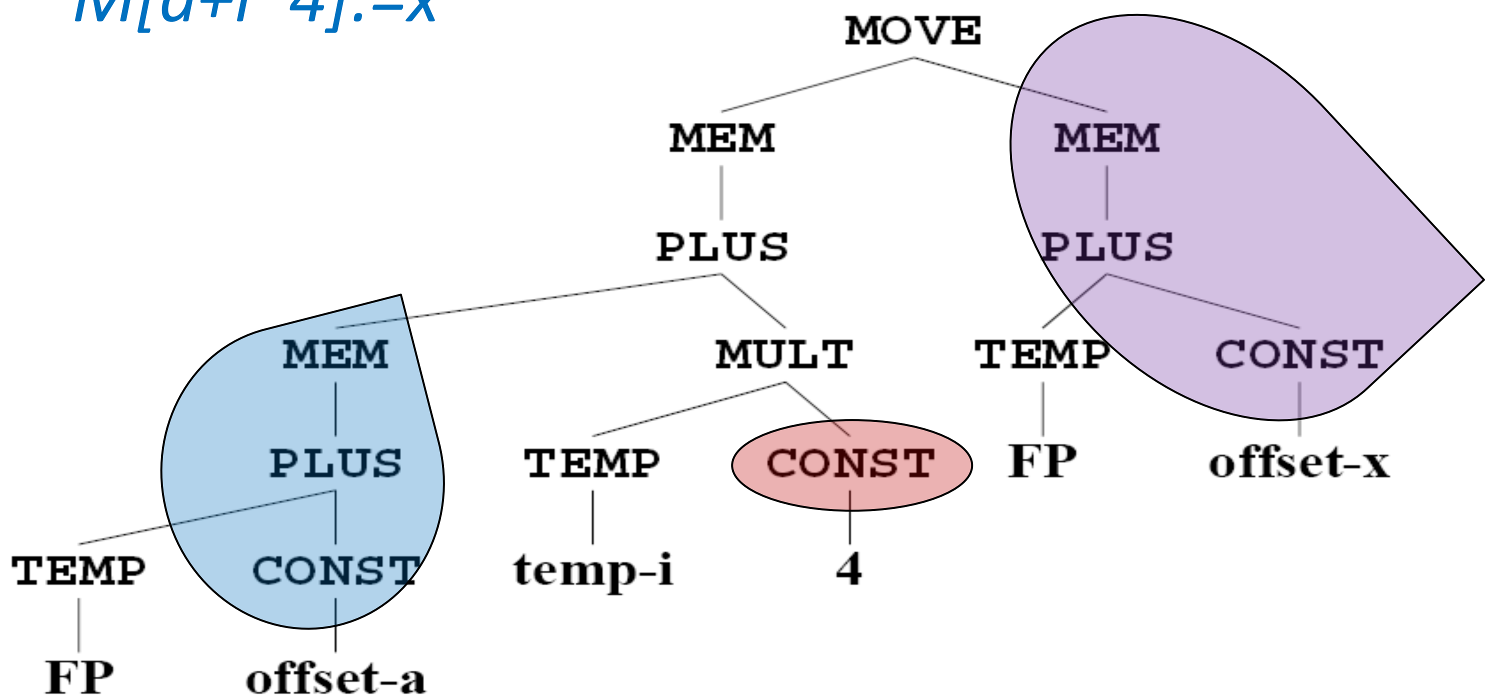


IR Tree for a[i] := x (*M[a+i*4]:=x*)

Example: Tiling via Jouette Tree Patterns

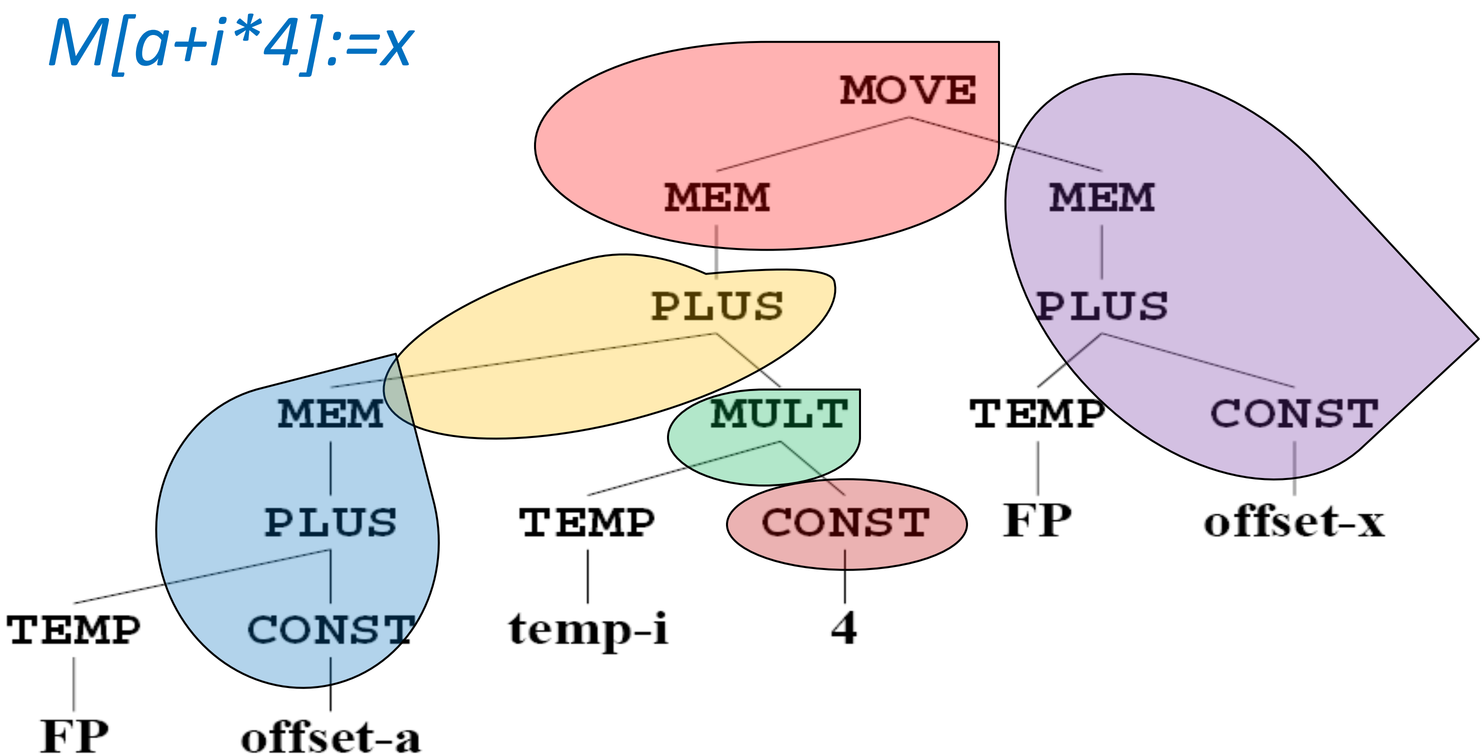
- **a[i] := x**, assuming i in register, a and x in stack frame

*M[a+i*4]:=x*



Example: Tiling via Jouette Tree Patterns

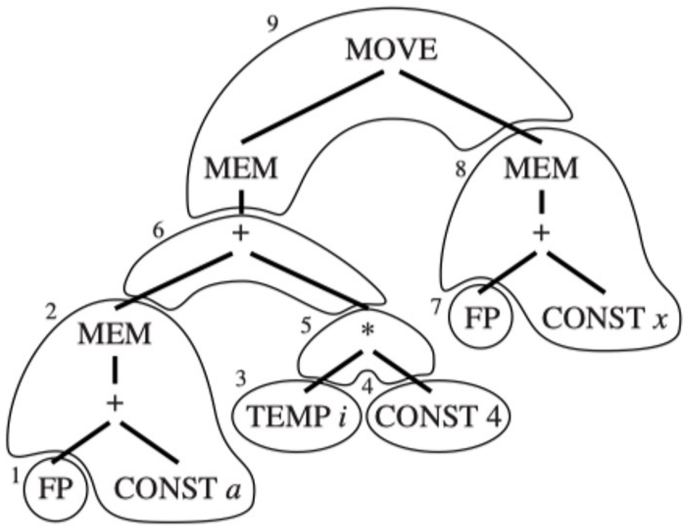
- **a[i] := x**, assuming i in register, a and x in stack frame



可以想象为铺地板的过程，每一个tree pattern就是各种大小不一的瓷砖（tile），指令选择就类似用瓷砖铺满屋子

Example: Generate Instructions from Tilings

- **a[i] := x**, assuming i in register, a and x in stack frame

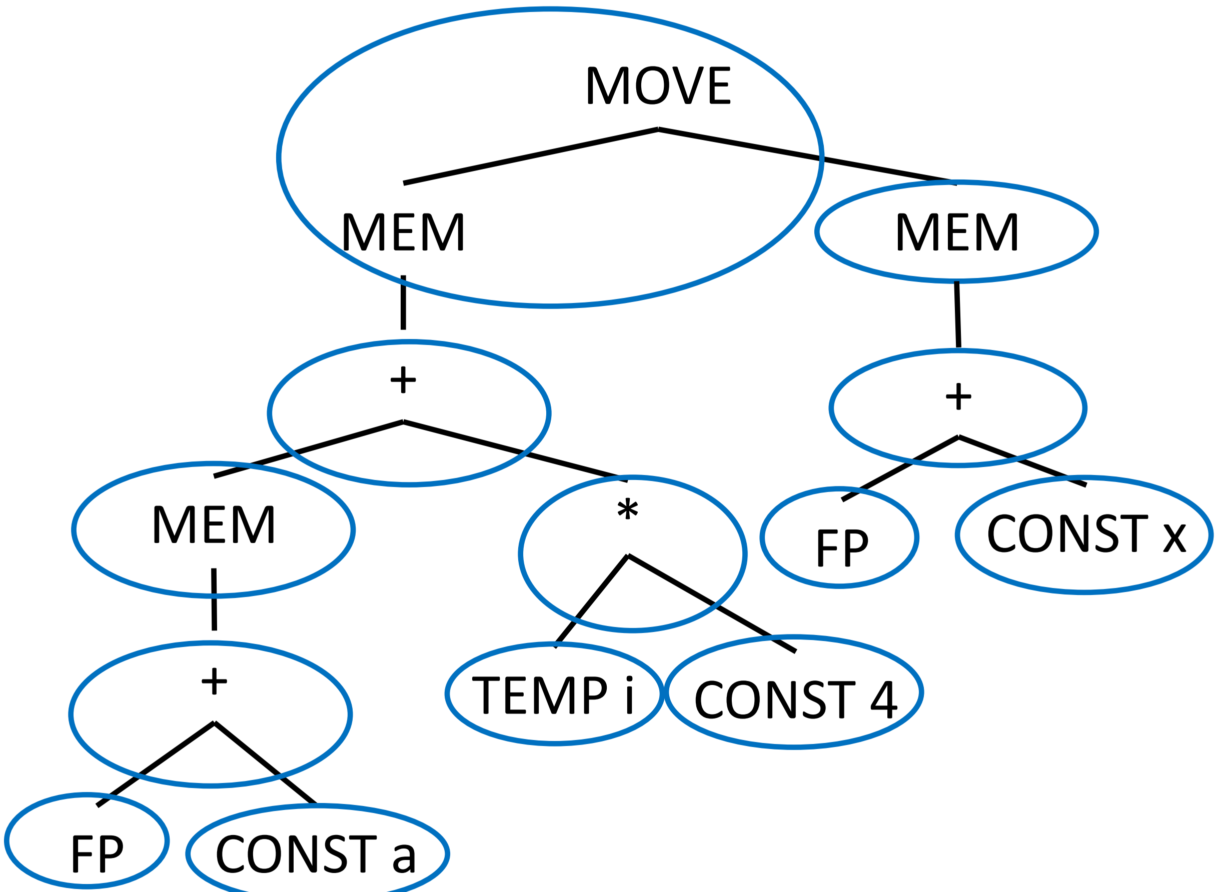


2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	LOAD	$r_2 \leftarrow M[\mathbf{fp} + x]$
9	STORE	$M[r_1 + 0] \leftarrow r_2$

- Tiles 1, 3, and 7 do not correspond to any machine instructions, because they are just (virtual) registers (TEMPs).

Example: Generate Instructions from Tilings

- It is always possible to tile the tree with tiny tiles, each covering only one node.
- For **a[i] := x**, such a tiling looks like this:



ADDI	**r1 ← r0 + a**
ADD	**r1 ← fp + r1**
LOAD	**r1 ← M[r1 + 0]**
ADDI	**r2 ← r0 + 4**
MUL	**r2 ← ri × r2**
ADD	**r1 ← r1 + r2**
ADDI	**r2 ← r0 + x**
ADD	**r2 ← fp + r2**
LOAD	**r2 ← M[r2 + 0]**
STORE	**M[r1 + 0] ← r2**

1. 指令选择概述

- □ 指令选择问题
- □ 基于树覆盖的指令选择
- □ Optimal and Optimum Tilings

Problem

- How to pick tiles that cover IR statement tree with minimum execution time?
- **Need a good selection of tiles**
  - Small tiles to make sure we can tile every tree
  - Large tiles for efficiency
- Usually want to pick large tiles
    - Fewer instructions
- Instructions ≠ cycles: RISC core instructions take 1 cycle, other instructions may take more

Node Selection

- There exist many possible tiles
- We want to have an instruction sequence of least cost
  - Sequence of instructions that take the least time to execute
  - For single issue fixed-latency machine, meaning fewest number of instruction

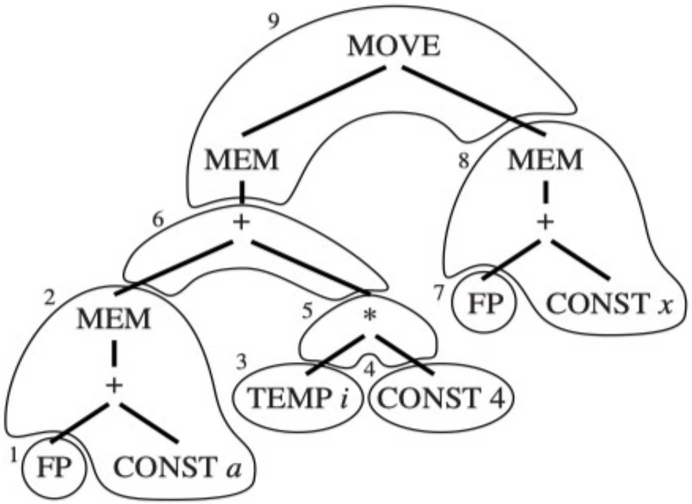
Node Selection Cont.

- Optimum tiling
  - Tiles sum to the lowest possible value.
  - Globally "the best"
- Optimal tiling
  - No wo adjacent tiles can be combined into a single tile of lower cost
  - Locally "the best"

Every optimum tiling is also optimal, but not vice versa

Example: Optimal and Optimum Tilings

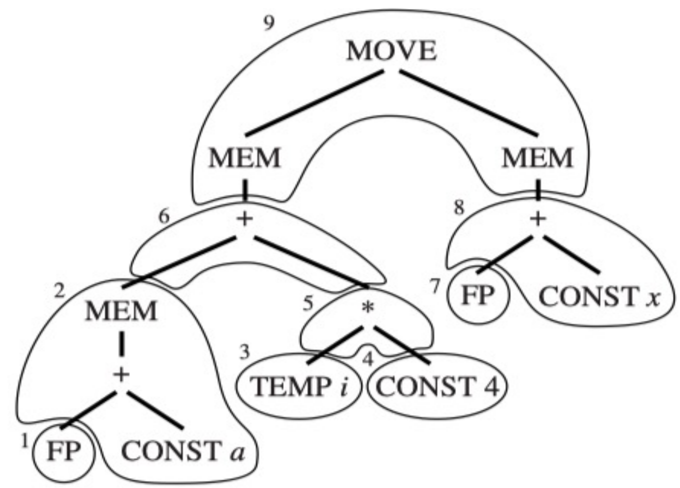
- Suppose every instruction costs one unit, except for MOVEM which costs m units.



2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	LOAD	$r_2 \leftarrow M[\mathbf{fp} + x]$
9	STORE	$M[r_1 + 0] \leftarrow r_2$

(a)

6 units



2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	ADDI	$r_2 \leftarrow \mathbf{fp} + x$
9	MOVEM	$M[r_1] \leftarrow M[r_2]$

(b)

5+m units

2. 指令选择算法

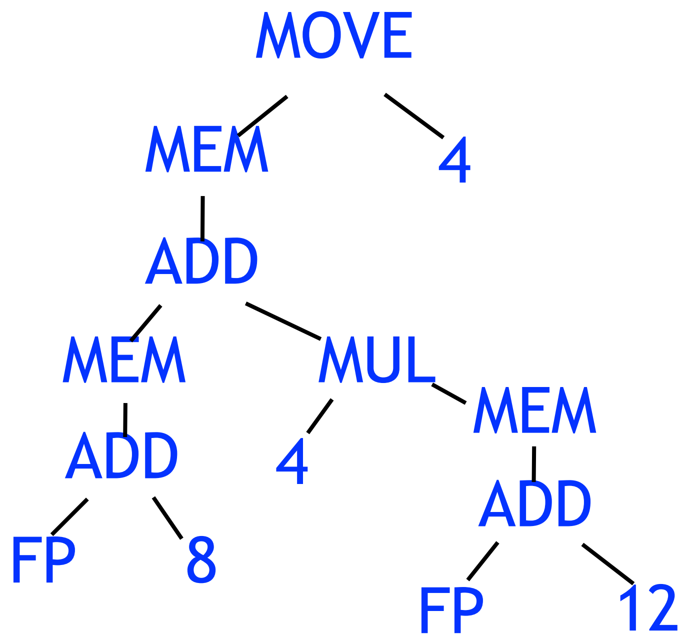
- Maximal Munch
- Dynamic Programming
- Tree Grammar

Algorithms for Instruction Selection

- **Maximal Munch**: Find an optimal tiling
  - Top-down strategy
  - Cover the current node with the largest tile
  - Repeat on subtrees
  - Generate instructions in **reverse-order** after tile placement
- **Dynamic Programming**: Find an optimum tiling
  - Bottom-up strategy
  - Assign cost to each node.
  - Cost = cost of selected tile + cost of subtrees
  - Select a tile with minimal cost and recurse upward

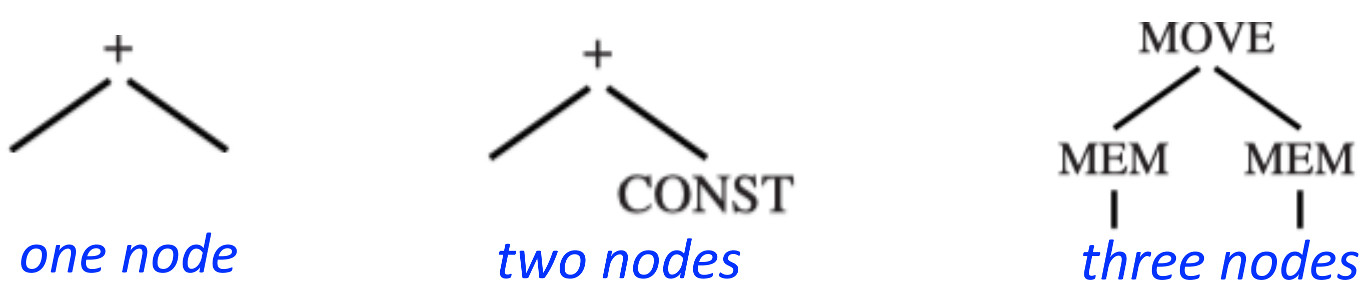
Maximal Munch: Greedy Tiling

- Assume larger tiles = better
- **Main idea**
  - Greedy algorithm: start from top of tree and use largest tile that matches tree
- Tile remaining subtrees recursively



Maximal Munch

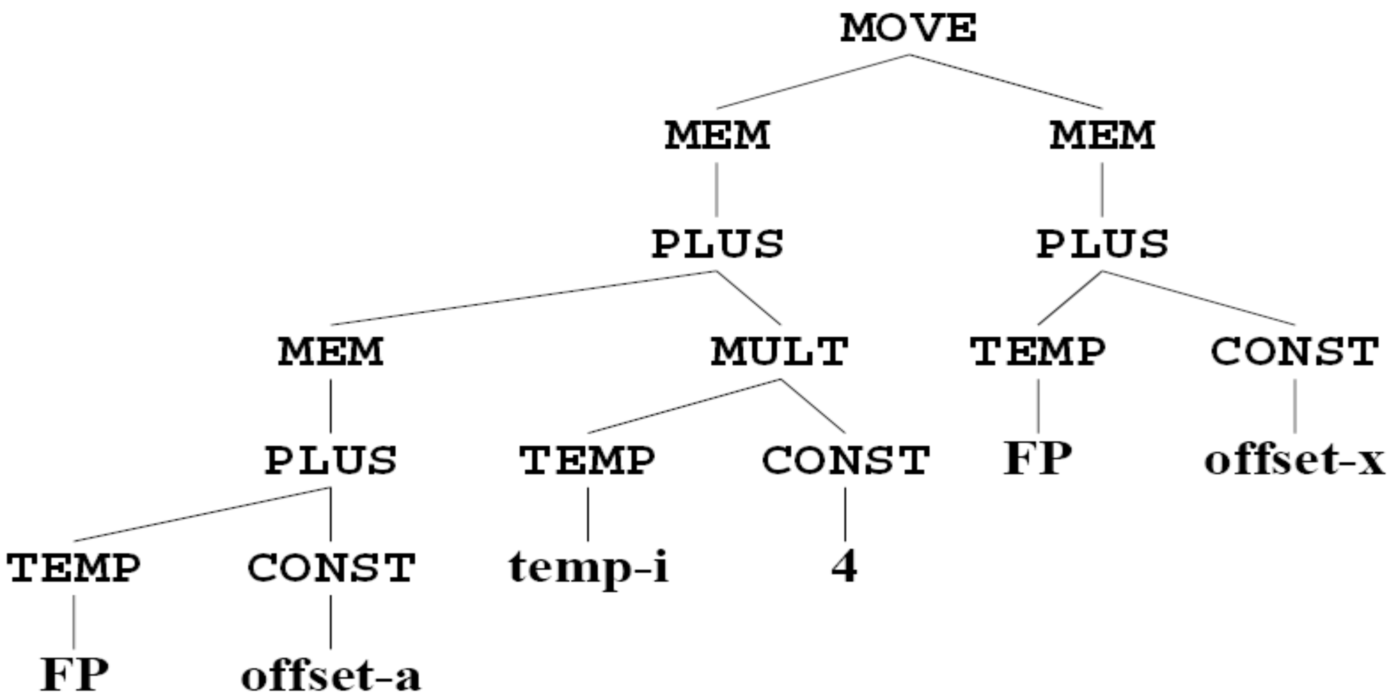
- Overall procedure:
  - Starting at the tree's root, find the largest tile that fits
  - Cover the root node – and perhaps several other nodes near the root – with this tile, leaving several subtrees
  - Repeat the same algorithm for each subtree
- **Largest tile**: the one with the most nodes
  - Tiles of equivalent size => arbitrarily choose one



Example: Maximal Munch

- Top-down strategy
- Cover the current node with the largest tile
- Repeat on subtrees

Name	*Effect*	*Trees*
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	+(_, _)
MUL	$r_i \leftarrow r_j \times r_k$	*(_, _)
SUB	$r_i \leftarrow r_j - r_k$	-(_, _)
DIV	$r_i \leftarrow r_j / r_k$	/(_, _)
ADDI	$r_i \leftarrow r_j + c$	+(_, CONST); +(CONST, _); CONST
SUBI	$r_i \leftarrow r_j - c$	-(_, CONST)
LOAD	$r_i \leftarrow M[r_j + c]$	MEM(+(_, CONST)); MEM(+(CONST, _)); MEM(CONST); MEM(_)
STORE	$M[r_j + c] \leftarrow r_i$	MOVE(MEM(+(_, CONST)), _); MOVE(MEM(+(CONST, _)), _); MOVE(MEM(CONST), _); MOVE(MEM(_), _)
MOVEM	$M[r_j] \leftarrow M[r_i]$	MOVE(MEM(_), MEM(_))

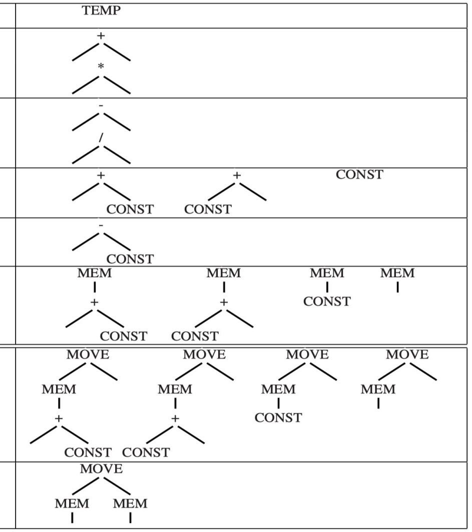


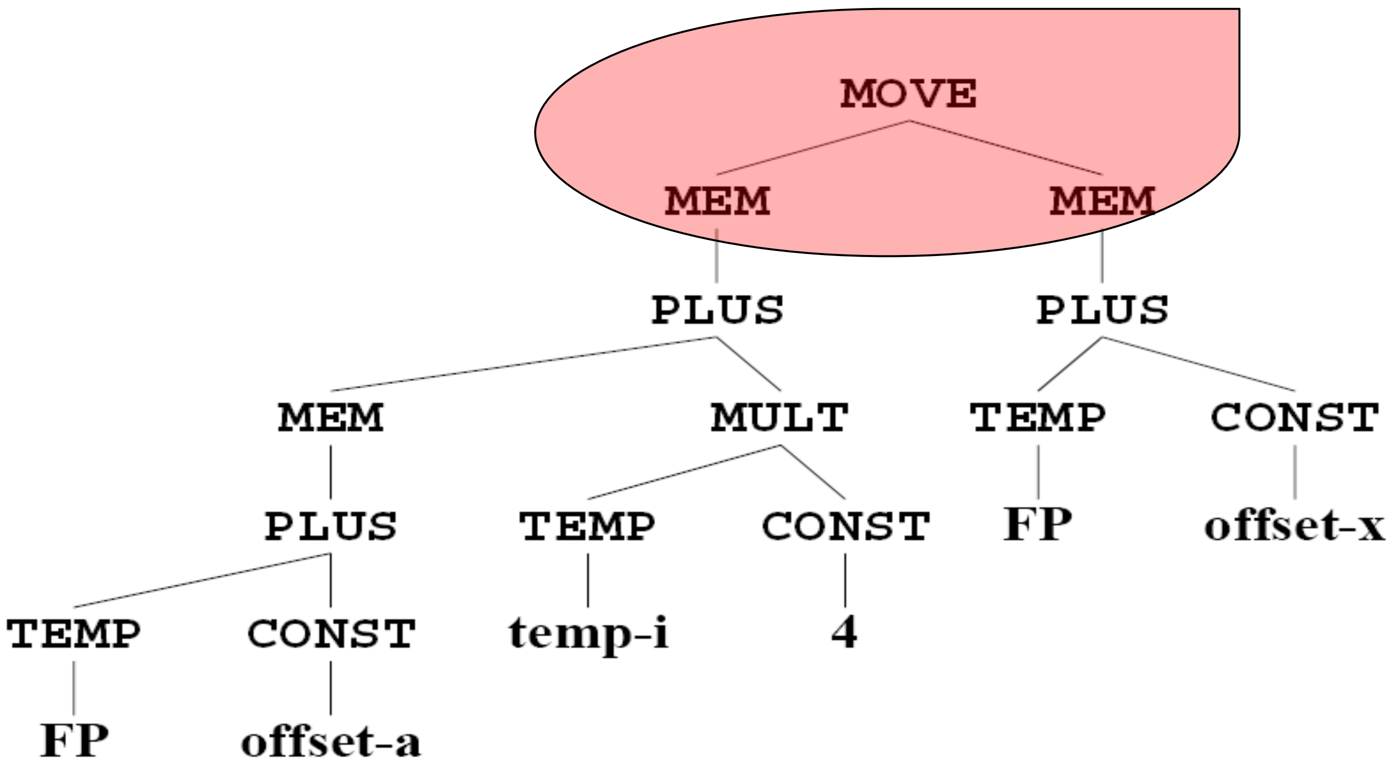
Example: Maximal Munch

- Top-down strategy
- Cover the current node with the largest tile
- Repeat on subtrees

Name	*Effect*
—	r_i
ADD	$r_i \leftarrow r_j + r_k$
MUL	$r_i \leftarrow r_j \times r_k$
SUB	$r_i \leftarrow r_j - r_k$
DIV	$r_i \leftarrow r_j / r_k$
ADDI	$r_i \leftarrow r_j + c$
SUBI	$r_i \leftarrow r_j - c$
LOAD	$r_i \leftarrow M[r_j + c]$
STORE	$M[r_j + c] \leftarrow r_i$
MOVEM	$M[r_j] \leftarrow M[r_i]$

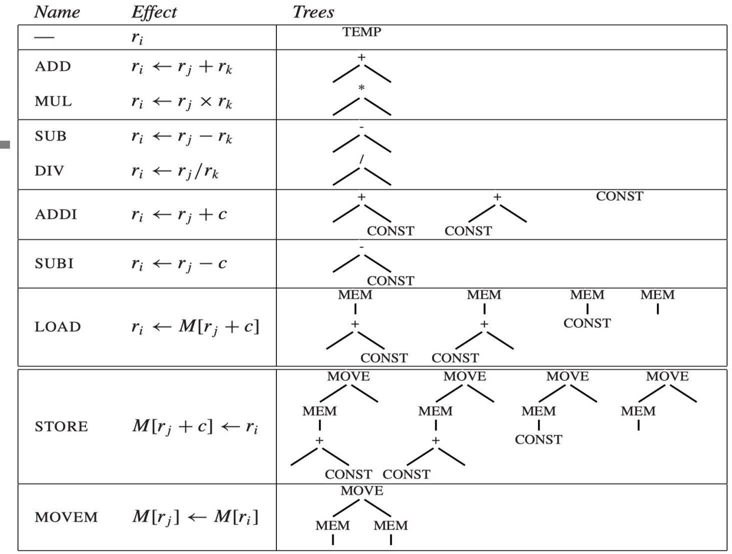
Trees



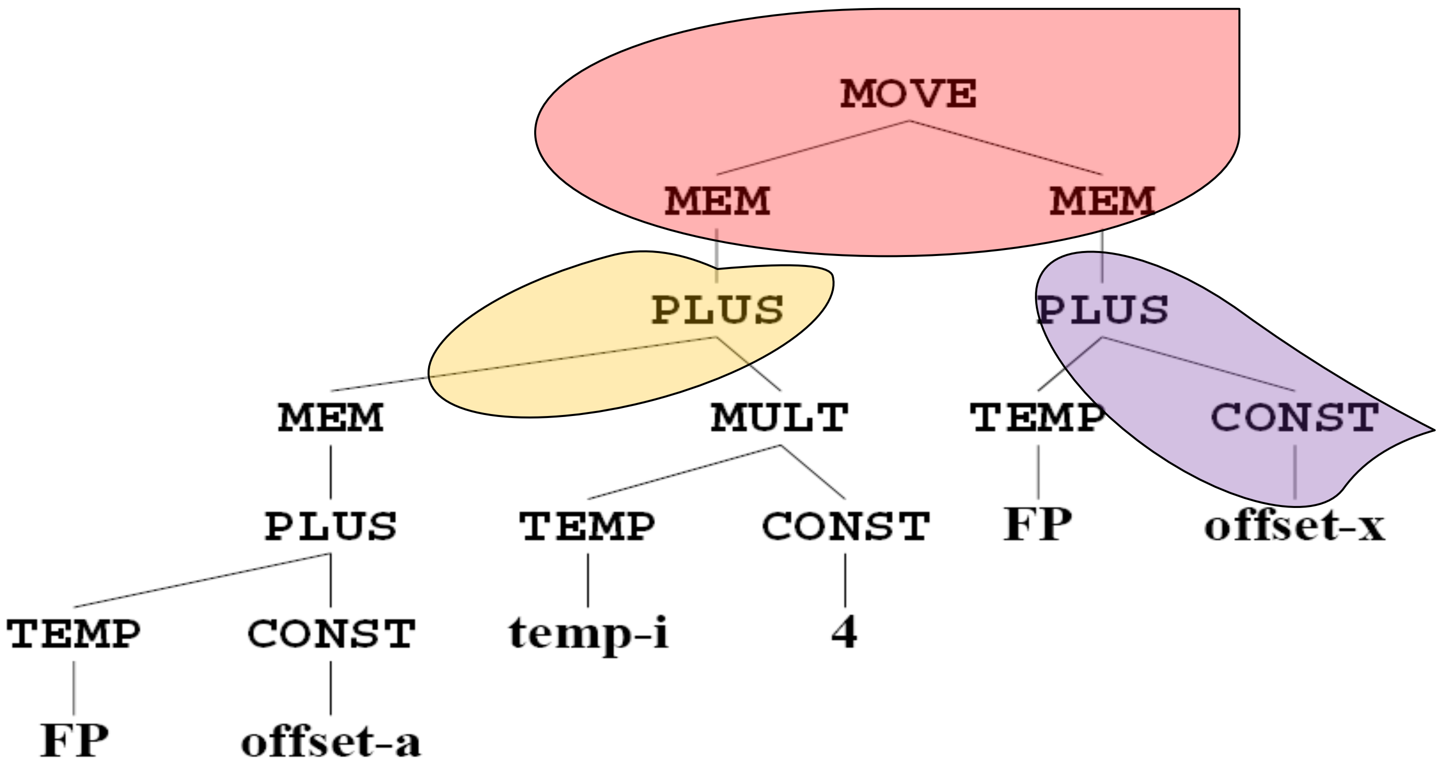


Example: Maximal Munch

- Top-down strategy
- Cover the current node with the largest tile
- Repeat on subtrees

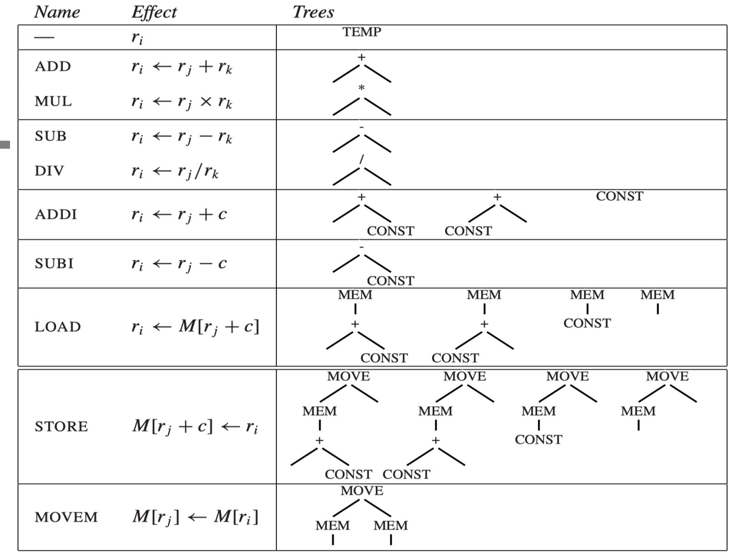


Name	Effect	Trees
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	+
MUL	$r_i \leftarrow r_j \times r_k$	*
SUB	$r_i \leftarrow r_j - r_k$	-
DIV	$r_i \leftarrow r_j / r_k$	/
ADDI	$r_i \leftarrow r_j + c$	+ CONST; + CONST; CONST
SUBI	$r_i \leftarrow r_j - c$	- CONST
LOAD	$r_i \leftarrow M[r_j + c]$	MEM + CONST; MEM + CONST; MEM CONST; MEM
STORE	$M[r_j + c] \leftarrow r_i$	MOVE MEM + CONST; MOVE MEM + CONST; MOVE MEM CONST; MOVE MEM
MOVEM	$M[r_j] \leftarrow M[r_i]$	MOVE MEM MEM

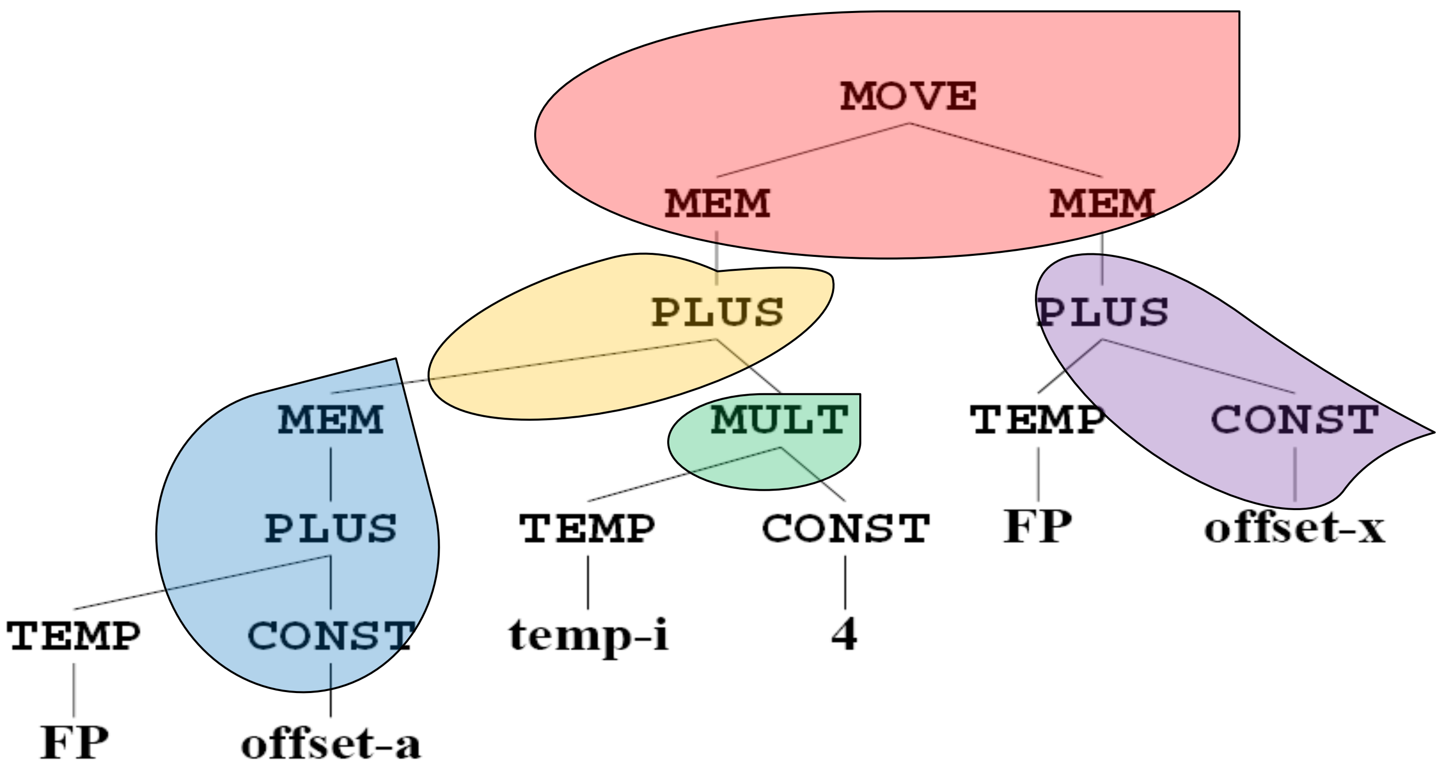


Example: Maximal Munch

- Top-down strategy
- Cover the current node with the largest tile
- Repeat on subtrees

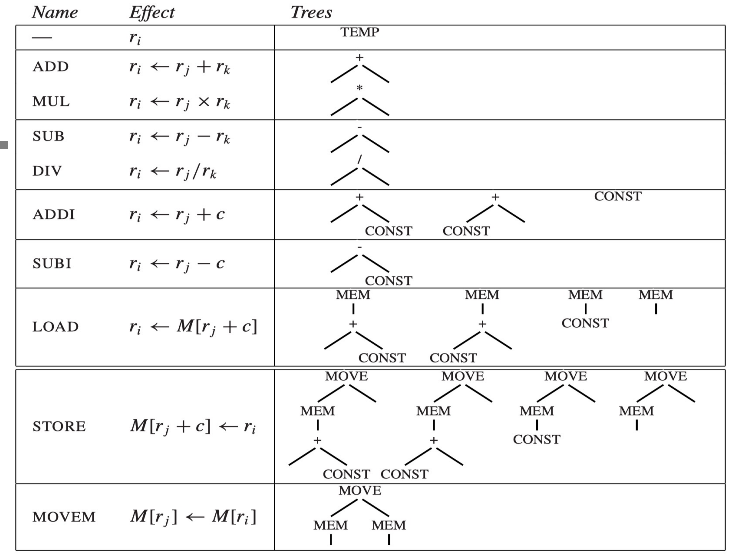


Name	Effect	Trees
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	+
MUL	$r_i \leftarrow r_j \times r_k$	*
SUB	$r_i \leftarrow r_j - r_k$	-
DIV	$r_i \leftarrow r_j / r_k$	/
ADDI	$r_i \leftarrow r_j + c$	+ CONST; + CONST; CONST
SUBI	$r_i \leftarrow r_j - c$	- CONST
LOAD	$r_i \leftarrow M[r_j + c]$	MEM + CONST; MEM + CONST; MEM CONST; MEM
STORE	$M[r_j + c] \leftarrow r_i$	MOVE MEM + CONST; MOVE MEM + CONST; MOVE MEM CONST; MOVE MEM
MOVEM	$M[r_j] \leftarrow M[r_i]$	MOVE MEM MEM

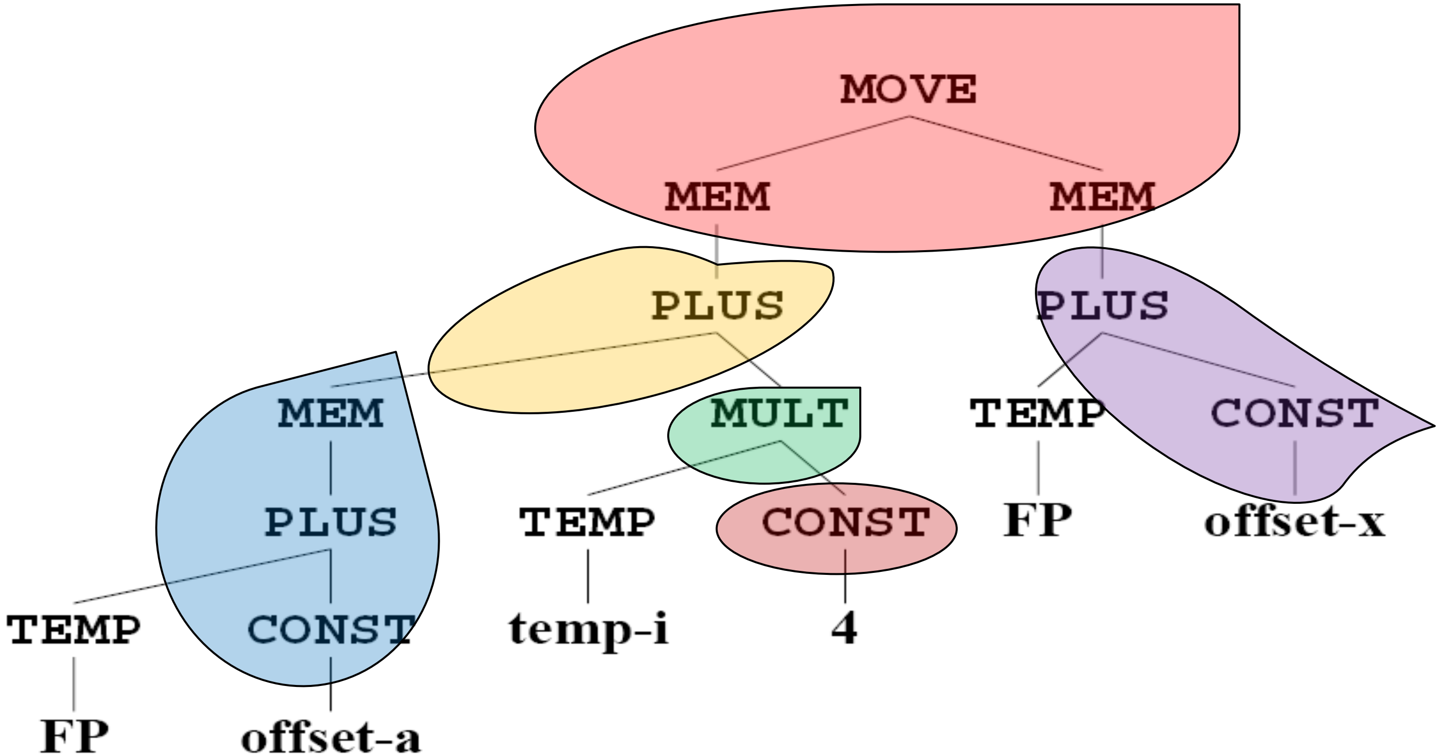


Example: Maximal Munch

- Top-down strategy
- Cover the current node with the largest tile
- Repeat on subtrees

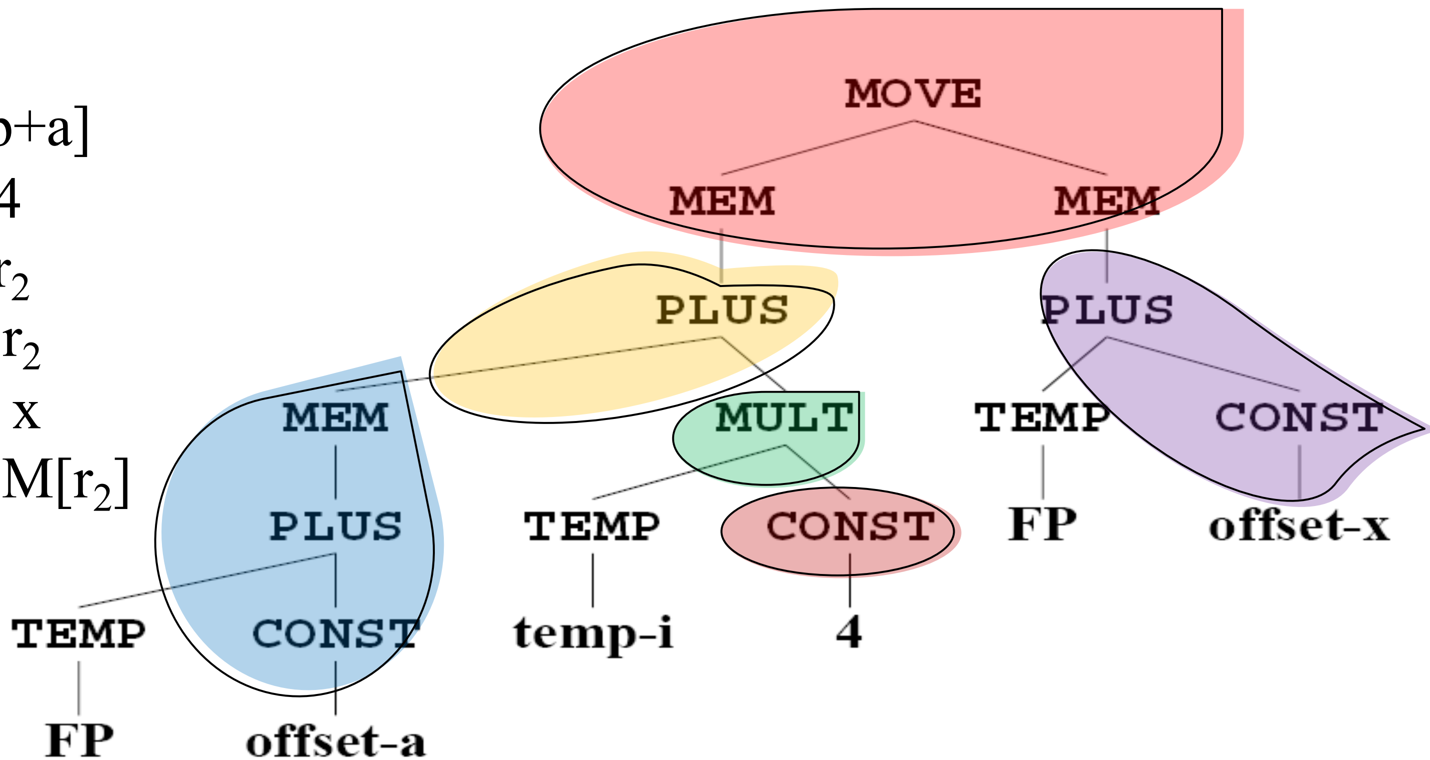


Name	Effect
—	r_i
ADD	$r_i \leftarrow r_j + r_k$
MUL	$r_i \leftarrow r_j \times r_k$
SUB	$r_i \leftarrow r_j - r_k$
DIV	$r_i \leftarrow r_j / r_k$
ADDI	$r_i \leftarrow r_j + c$
SUBI	$r_i \leftarrow r_j - c$
LOAD	$r_i \leftarrow M[r_j + c]$
STORE	$M[r_j + c] \leftarrow r_i$
MOVEM	$M[r_j] \leftarrow M[r_i]$



Example: Maximal Munch

LOAD	$r_1 = M[fp+a]$
ADDI	$r_2 = r_0 + 4$
MUL	$r_2 = r_i * r_2$
ADD	$r_1 = r_1 + r_2$
ADD	$r_2 = fp + x$
MOVEM	$M[r_1] = M[r_2]$



2. 指令选择算法

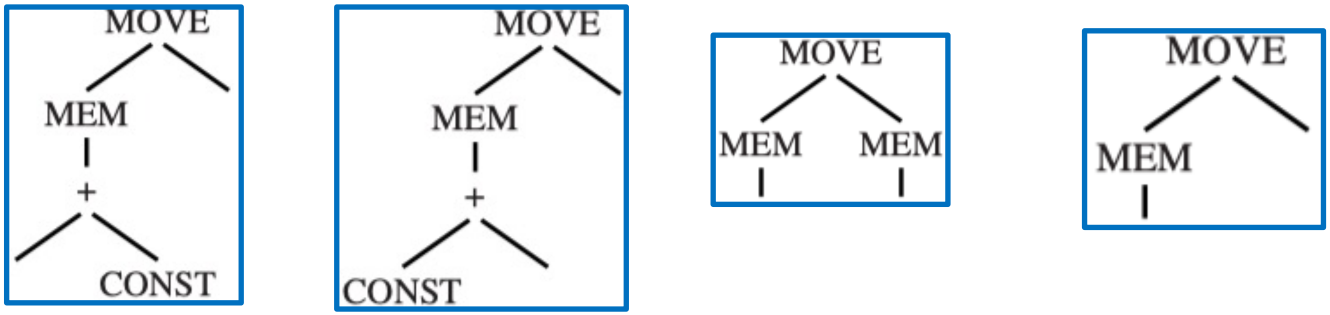
- Maximal Munch
- Dynamic Programming
- Tree Grammar

Optimum Instruction Selection

- **Goal**: find minimum total cost tiling of tree
- **Idea**: dynamic programming
  - For every node, find minimum total cost tiling of that node and sub-tree.
- **Lemma**:
  - Once minimum cost tiling of all children of a node is known, can find minimum cost tiling of the node by trying out all possible tiles matching the node

Dynamic Programming

- **Maximal munch**: always finds an optimal tiling, but not necessarily an optimum. It works top-down



- **Dynamic programming**: can find the optimum based on the optimum solution of each subproblem
  - It works bottom-up
  - Assign a cost to every node in the tree.
  - The cost of a node: the sum of the instruction-costs of the best instruction sequence that can tile the subtree rooted at that node.

Dynamic Programing

- Maintain a table: node x→ the optimum tiling covering node x and its cost
- For a node *x*, let *f(x)* be the cost of the optimal tiling for the whole expression tree rooted at *x*. Then

$$f(x) = \min_{\forall \textbf{tile } T \textbf{ covering } x} \left(\textbf{cost}(T) + \sum_{\forall \textbf{child } y \textbf{ of tile } T} f(y) \right)$$

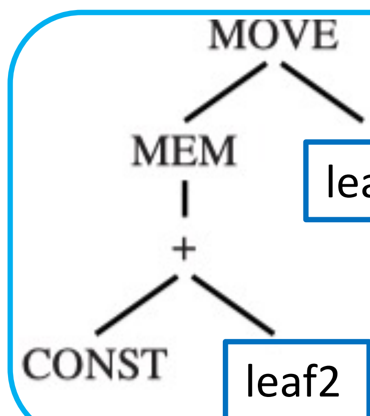
Dynamic Programming -- Details

Given the IR Tree with root node n

- First, the costs of all the children (and grandchildren, etc.) of node n are found recursively.
- Then, each tree-pattern (tile kind) is matched against node n
- Each tile has zero or more leaves, which are places where subtrees can be attached.
- For each tile t of cost c_t that matches at node n, the cost of matching tile t is: (c_i has already been computed)

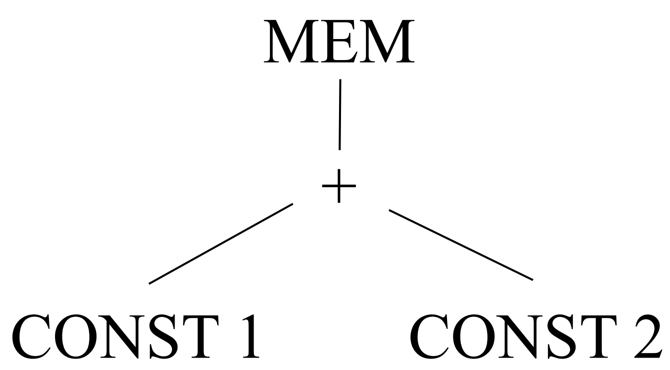
$$c_t + \sum_{all\ leaves\ i\ of\ t} c_i$$

- **The tree pattern leading to minimum cost is chosen.**



Example: Dynamic Programming

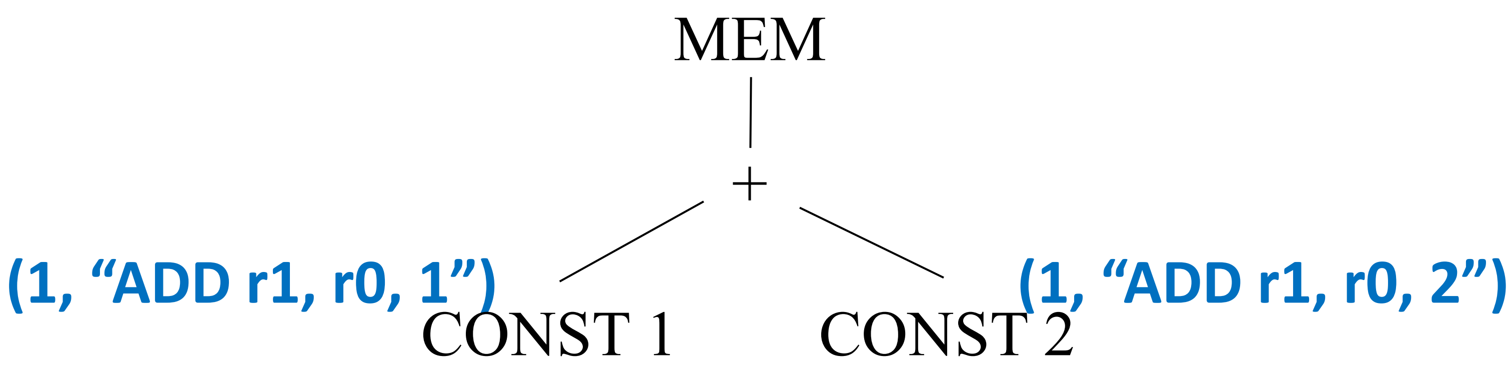
- MEM(BINOP(PLUS, CONST(1), CONST(2)))
- MEM(PLUS(CONST(1), CONST(2)))



Example: Dynamic Programming

(a, b): we mark a as the minimum cost of a node,

b as the optimal instruction

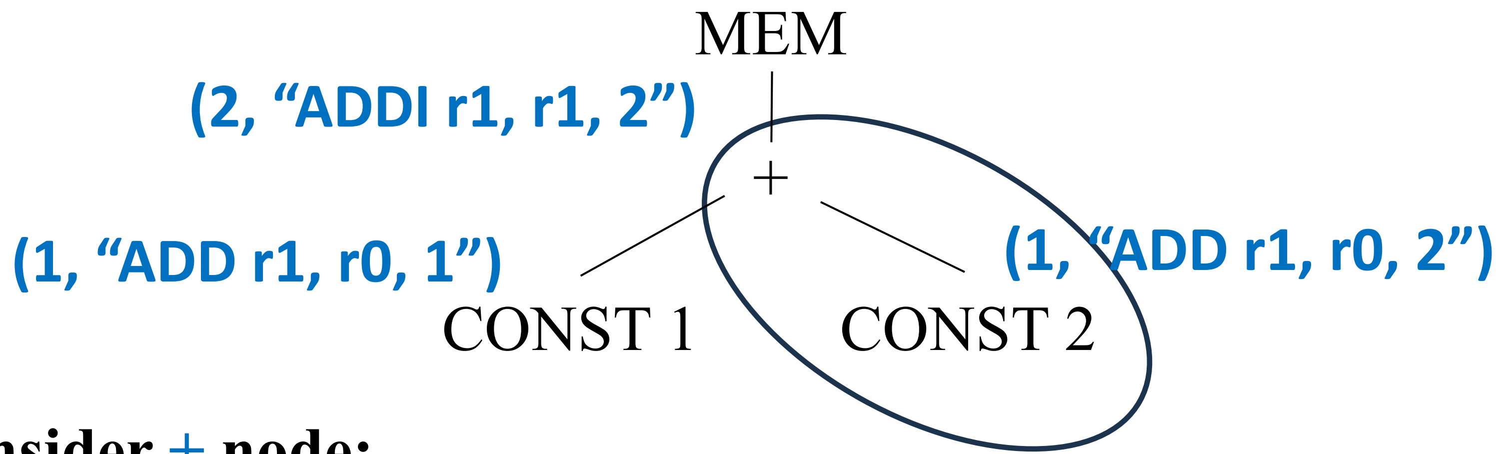


Consider CONST node:

- The only tile that matches CONST 1 is an ADDI instruction with cost 1
- Similarly, CONST 2

Pattern (Tile)	Instruction Cost	Leaves Cost	Total
CONST	1(ADDI)	0	1

Example: Dynamic Programming

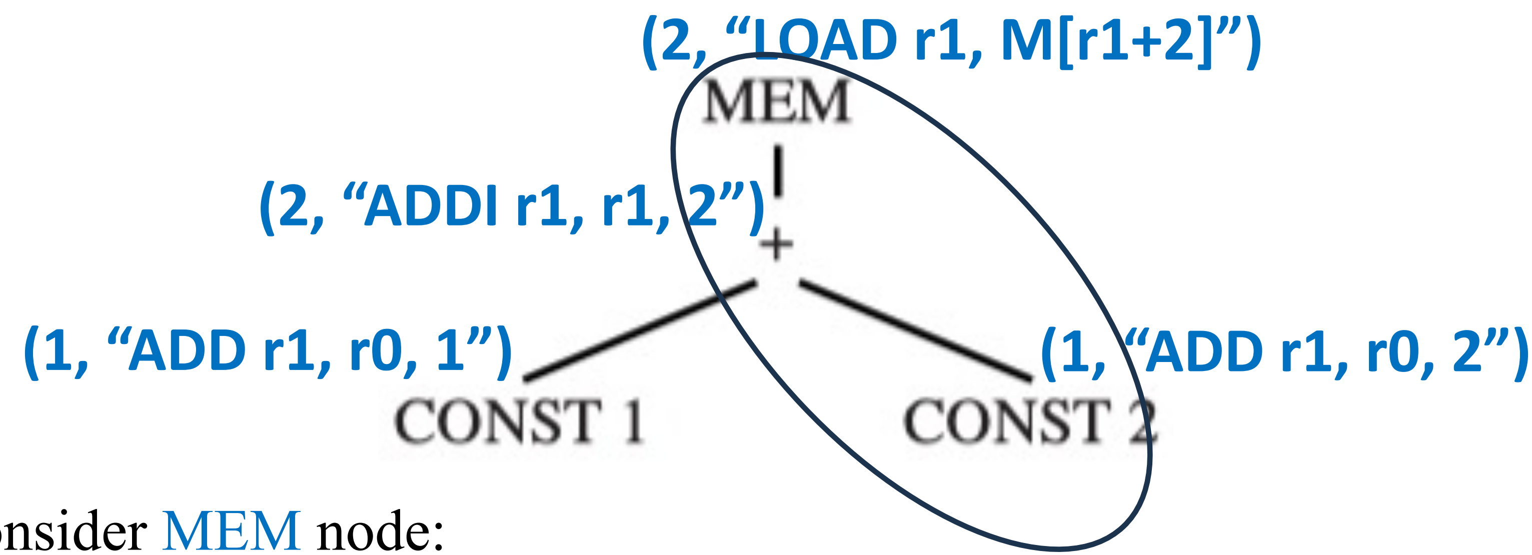


Consider + node:

- Several tiles match the + node

Tile	Instruction	Tile Cost	Leaves Cost	Total cost
+	ADD	1	1+1	3
+ CONST (right)	ADDI	1	1	2
+ CONST (left)	ADDI	1	1	2

Example: Dynamic Programming

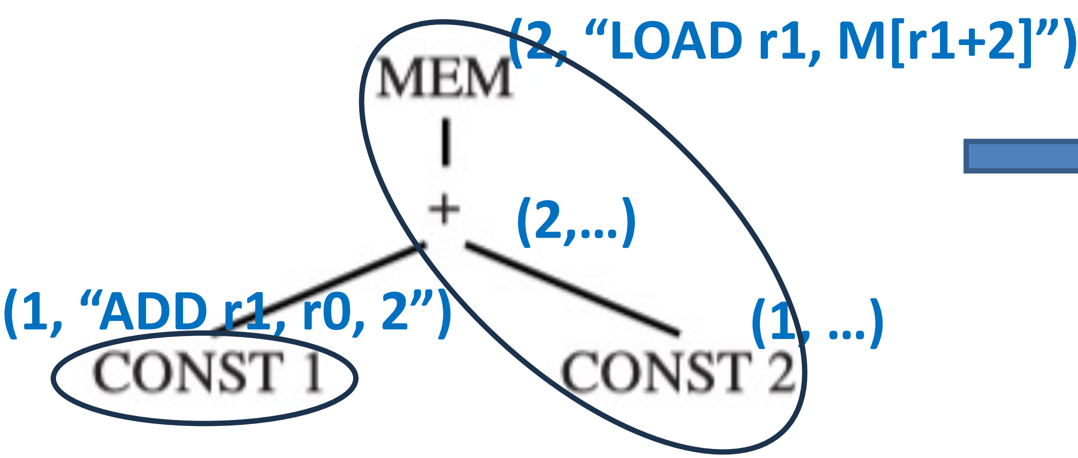


Consider MEM node:

Tile	Instruction	Tile Cost	Leaves Cost	Total cost
MEM |	LOAD	1	2	3
MEM | + / \\ CONST	LOAD	1	1	2
MEM | + / \\ CONST	LOAD	1	1	2

Dynamic Programming – Instruction Emission

- Once the cost of the root node (and thus the entire tree) is found, the instruction emission phase begins. The algorithm is as follows:
  - Emission(node n):
    - For each leaf li of the tile selected at node n, perform Emission(li).
    - Emit the instruction matched at node n.
- Emission(li): perform on the leaves of the tile that matched at n



ADDI r1 ← r0 + 1
LOAD r1 ← M[r1 + 2]

Efficiency of Tilling Algorithms

How expensive are maximal munch and dynamic programming?

- Suppose:
  - T: the number of different tiles
  - K: the average matching tile contains K nonleaf nodes.
  - K': the largest number of nodes that ever need to be examined to see which tiles match at a given subtree
  - T': the number of different patterns (tiles) match at each tree node, on the average
  - N: the number of nodes in the input tree

- Maximal munch: proportional to (K' + T') N / K
- Dynamic programming: proportional to (K' + T') N
- K, K' and T' are constant, the running time of all of these algorithms are linear.

2. 指令选择算法

- □ **Maximal Munch**
- □ **Dynamic Programming**
- □ **Tree Grammar (了解为主)**

Motivation

- **Problem**
  - For machines with complex instruction sets and several classes of registers and addressing modes (e.g., CISC)
  - It can be hard to use the simple tree patterns and tiling alg.
  - Hard-codes the tiles in the code generator: tedious, error-prune!
- **Idea**
  - Define tiles in a separate specification define tiles in a separate specification
  - Use a generic tree pattern matching algorithm to compute tiling

We want "instruction selector generators!"

Instruction Selection via Tree Grammars

- **Problem:** Hard-codes the tiles in the code generator can be tedious and error-prune
- **Idea:** instruction selector generators?
- **Solution**:
  - Use a **tree grammar** (a special context-free grammar) to describe the tiles,
  - Reduce instruction selection **to** a **parsing problem**!
  - Use a generalization of the dynamic programming algorithm for the "parsing"

Instruction Selection via Tree Grammars

- The relationships for tiles are encoded as **rewriting rules**. Each rule comprises of
  - A production in a tree grammar
  - An associated cost
  - A code generation template

Pattern, replacement	*Cost*	*Template*
$+(reg_1, reg_2) \rightarrow reg_2$	1	add r1, r2
$store(reg_1, load(reg_2)) \rightarrow$ *done*	5	movem r2, r1

Instruction Selection via Tree Grammars

- **The tree grammars can be ambiguous**
  - There are many different instruction sequences implementing the same expression.

- **Which “parsing algorithms” to use?**
  - The parsing techniques described in Chapter 3 are not very useful
  - However, a generalization of the dynamic-programming algorithm works quite well

Example

#	*Pattern, replacement*	*Cost*	*Template*
1	$+(reg_1,reg_2) \rightarrow reg_2$	1	`add r1, r2`
2	$\times(reg_1,reg_2) \rightarrow reg_2$	10	`mul r1, r2`
3	$+(num,reg_1) \rightarrow reg_2$	1	`addi num, r1`
4	$\times(num,reg_1) \rightarrow reg_2$	10	`muli num, r1`
5	$store(reg_1, load(reg_2)) \rightarrow$ *done*	5	`movem r2, r1`

	store(+(ra,×(+(ri,1),4)),load(+(rb, ×(rj, 4))))	
4	store(+(ra,×(+(ri,1),4)),load(+(rb, rj)))	`muli 4, rj`
1	store(+(ra,×(+(ri,1),4)),load(rb))	`add rj, rb`
3	store(+(ra,×(ri,4)),load(rb))	`addi 1, ri`
4	store(+(ra,ri),load(rb))	`muli 4, ri`
1	store(ra,load(rb))	`add ri, ra`
5	*done*	`movem rb, ra`

Instruction Selection via Tree Grammars

- Several compilation tasks: can be formally describe and their implementations can be automatically generated

Compilation Task	**Description Formulation**	**Acceptor**
Lexical analysis	Regular expressions	Finite automata
Syntax analysis	Context-free grammars	Pushdown automata
Instruction selection	Regular tree grammars	Finite tree automata

- Techniques for instruction selection (input: IR Tree)

Compilation Task	**Description Formulation**
Hard-coded matcher like Tile	Avoids large sparse table; lots of work
Use parsing techniques	Automation; ambiguous grammars
Linearize tree into string and …	..
…	…

3. CISC Machines

RISC vs. CISC

RISC machine	**CISC machine**
32 registers	few registers (16, or 8, or 6)
only one class of integer/pointer register	registers divided into different classes, with some operations available only on certain registers;
arithmetic operations only between registers	arithmetic operations can access registers or memory through "addressing modes";
"three-address" instructions of the form **r1** $\leftarrow$ **r2** $\oplus$ **r3**	"two-address" instructions of the form **r1** $\leftarrow$ **r1** $\oplus$ **r2**;
load and store instructions with only the M[reg+const] addressing mode	several different addressing modes;
Every instruction exactly 32 bits long	variable-length instructions, formed from variable-length opcode plus variable-length addressing modes;
one result or effect per instruction	instructions with side effects such as "autoincrement" addressing modes.

CISC can be hard to model via tree pattern-based tilings

Problems and Solutions of CISC

- **Few registers**:
  - Solution: generate TEMP nodes freely, and assume that the register allocator will do a good job
- **Classes of registers**:
  - E.g., Pentium的乘法指令要求将左操作数放入寄存器eax，结果的高位放入rdx
  - Solution: move the operands and result explicitly.
  - Example: to implement t1 ← t2 × t3:

mov eax, t2	eax t2	
mul t3	eax ← eax × t3;	edx ← garbage
mov t1, eax	t1 ← eax	

Problems and Solutions of CISC

- **Two-address instructions**:
  - Solution: Adding extra move instructions
  - Example: In order to implement t1 ← t2 + t3

    mov t1, t2   t1 ← t2

    add t1, t3   t1 ← t1 + t3

  - Then we hope that the register allocator will be able to allocate t1 and t2 to the same register, so that the move instruction will be deleted.

Problems and Solutions of CISC

- **Arithmetic operations can address memory:**
  - The instruction selection phase turns every TEMP node into a "register" reference. Many of these "registers" will actually turn out to be memory locations.
  - Solution: Fetch all the operands into registers before operating and store them back to memory afterwards.
  - Example: these two sequences compute the same thing:

mov eax, [ebp - 8]	
add eax, ecx	add [ebp - 8], ecx
mov [ebp - 8], eax	

Problems and Solutions of CISC

- **Several addressing modes:**
  - Two advantages:
    - They "trash" fewer registers
    - shorter instruction encoding
  - With some work, tree-matching instruction selection can be made to select CISC, but programs can be just as fast using the simple RISC-like instructions
- **Variable-length instructions:**
  - Not really a problem for the compiler;
  - Once the instructions are selected, it is a trivial (though tedious) matter for the assembler to emit the encodings.

Problems and Solutions of CISC

- **Instructions with side effects:**
  - Problem: some machines have an "autoincrement" memory fetch instruction whose effect is:

    $$r2 \leftarrow M[r1];\ r1 \leftarrow r1 + 4$$

  - Difficult to model using tree patterns, since it produces two results.
  - There are three solutions:
    - Ignore the autoincrement instructions, and hope they go away.
    - Try to match special idioms in an ad hoc way, within the context of a tree pattern-matching code generator.
    - Use a different instruction algorithm entirely, one based on DAG patterns instead of tree patterns.

Algorithms for Instruction Selection

- Algorithms for optimal tilings are simpler than optimum tilings.
- For **CISC**, the difference between optimum and optimal tilings is noticeable
  - Some instructions accomplish several operations each
- For **RISC**, there is usually no difference at all between optimum and optimal tilings
  - The tiles are small and of uniform cost.
- Thus, for RISC, the simpler tiling algorithms suffice

Instruction Selection for Modern Processor

- Execution time not sum of tile times
- Cost is an approximation
- E.g., instruction order also matters
  - Pipelining: parts of different instruction overlap
  - Bad ordering stalls the pipelining
  - Instruction scheduling helps

Thank you all for your attention

Generating Code

- Given a tiled tree, to generate code
  - Postorder treewalk; node-dependant order for children
  - Emit code sequences corresponding to tiles in order
  - Connect tiles by using same register name to tie boundaries together

Tiling Algorithms: Maximal Munch

- Maximal Munch
  - Start at root of tree, find largest tile that fits. Cover the root node and possibly other nearby nodes. Then repeat for each subtree
  - Generate instruction as each tile is placed
    - Generates instructions in reverse order
  - Generates an optimal tiling – but may not be optimum

Tiling Algorithms: Dynamic Programming

- Dynamic Programming
  - There may be many tiles that could match at a particular node
  - Idea: Walk the tree and accumulate the set of all possible tiles that could match at that point – Tiles(n)
  - Then: Select minimal cost for subtrees (bottom up), and go top-down to select and emit lowest-cost instructions