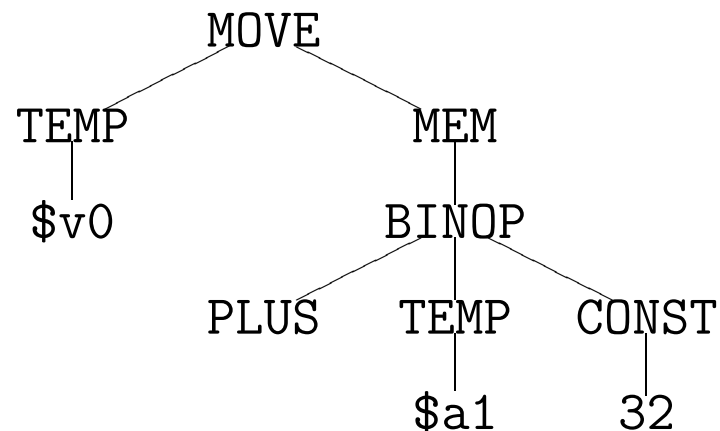


8 Instruction Selection

- The **IR code instructions** were designed to do exactly *one* operation: load/store, add, subtract, jump, etc.
- The **machine instructions** of a real CPU often perform *several* of these primitive operations.

Example: The MIPS machine instruction `lw $v0, 32($a1)` corresponds to the following IR tree:



- Finding the appropriate machine instructions to implement a given IR tree is called **instruction selection**.
- The instruction selection phase is supplied with a **machine description**, a set of IR **tree patterns** describing the machine instructions of the target CPU.

Instruction selection then amounts to **tiling** the IR tree with a (minimal) set of tree patterns.

- In this chapter we will use a hypothetical target CPU, the *Jouette*³² architecture.

N.B.

- In the following, we abbreviate the IR tree



³²French: toy.

- Machine description for *Jouette*:³³

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	$r_i \leftarrow r_j - c$	

³³In the *Jouette* CPU, register r_0 always contains the value 0 (MIPS: \$zero).

- Machine description for *Jouette* (continued):³⁴

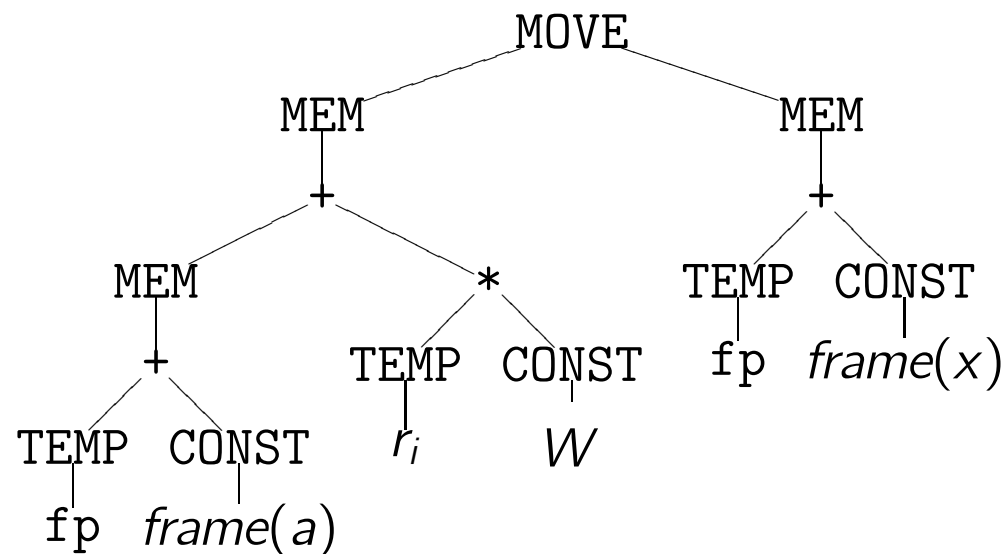
Instruction	Effect	IR Tree Pattern
load	$r_i \leftarrow M[r_j + c]$	
store	$M[r_j + c] \leftarrow r_i$	
movem	$M[r_j] \leftarrow M[r_i]$	

³⁴ $M[x]$ denotes the memory word at address x .

- Instruction selection means *tiling* the IR tree.
 - The **tiles** are the tree patterns available in the machine description.
 - The fundamental goal is to **cover the tree with non-overlapping tiles**.
- Example:** The Tiger assignment statement

$$a[i] := x$$

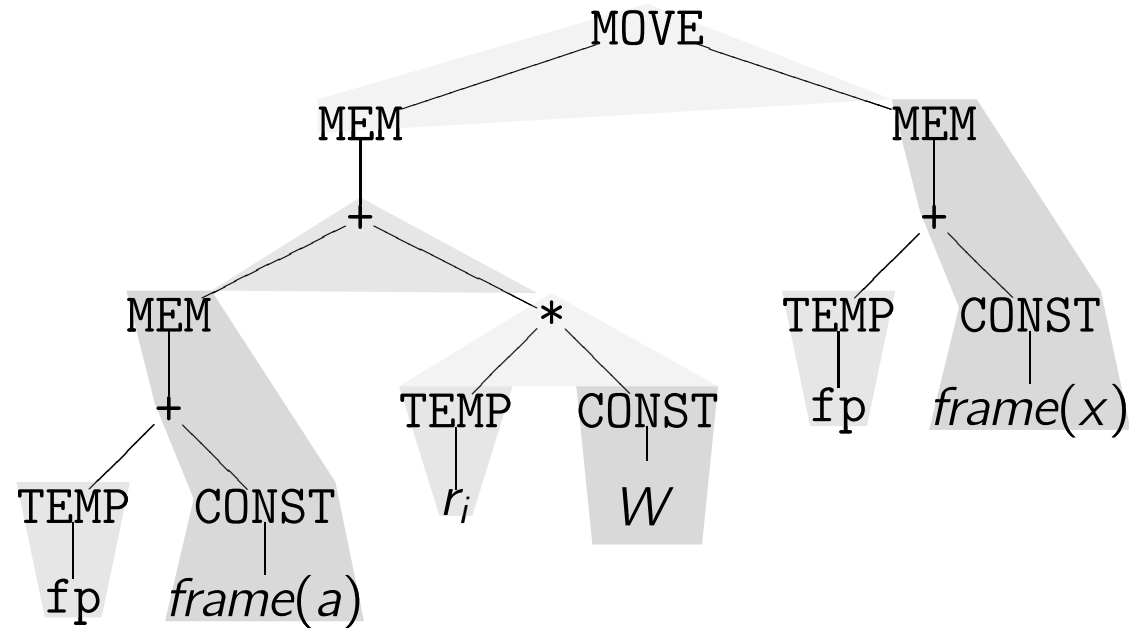
is translated into the following IR tree³⁵ (variables a and x are frame-resident, i lives in register r_i):



³⁵ $frame(v)$ denotes the offset of variable v 's slot in the frame, cf. access, Chapter 6.

- This tree has several valid tilings:

①

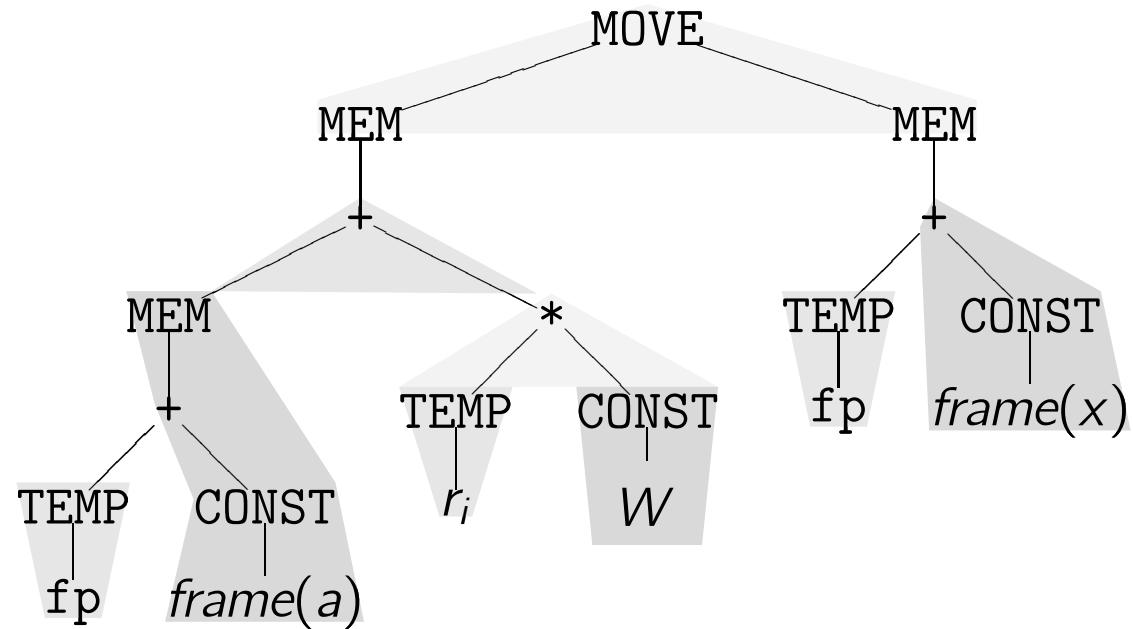


Jouette machine instructions

```

1  load  r1, frame(a)(fp)
2  addi  r2, r0, W
3  mul   r2, ri, r2
4  add   r1, r1, r2
5  load  r2, frame(x)(fp)
6  store 0(r1), r2
  
```

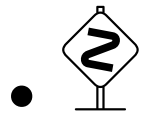
②



Jouette machine instructions

```

1  load  r1, frame(a)(fp)
2  addi  r2, r0, W
3  mul   r2, ri, r2
4  add   r1, r1, r2
5  addi  r2, fp, frame(x)
6  movem r1, r2
  
```



- Can we always find a tiling such that the given IR tree can be covered?
 - Yes, if the machine instruction set is “reasonable”, we can at least produce a tiling such that *each tile covers one IR tree node only*.

Example: naive tiling for the previous IR tree:

Jouette machine instructions	
1	addi $r_1, r_0, \text{frame}(a)$
2	add r_1, fp, r_1
3	load $r_1, 0(r_1)$
4	addi r_2, r_0, W
5	mul r_2, r_1, r_2
6	add r_1, r_1, r_2
7	addi $r_2, r_0, \text{frame}(x)$
8	load $r_2, 0(r_2)$
9	store $0(r_1), r_2$

- It is expected that the execution cost ($\approx \#$ of instructions) of the naive tiling is higher than for the two previous tilings.

8.1 Algorithms for Instruction Selection

- A number of reasonable algorithms exist to solve the IR tree tiling problem. We will first discuss an algorithm that produces an *optimal tiling*:

| In an **optimal tiling**, no two adjacent tiles can be combined into a single tile of lower cost.

- For simplicity, we can just set the cost of each instruction to 1 unit. For a modern CPU, nearby instructions interact in many ways such that a good cost function will actually be quite complicated to design.

Example: for the Intel Pentium 4, *instruction latency* is influenced by the following CPU features:

- *branch prediction, speculative execution*
- *non-blocking memory access*
- *pipelining*
- *multiple cache levels, temporal/spatial locality of data access*

8.1.1 Maximal Munch

- The simple algorithm **Maximal Munch** always finds an optimal tiling:
 - ① *Start at the root of the IR tree.*
 - ② *Find the largest (maximum number of covered IR tree nodes) tile t that fits.*
 - ③ *Record the machine instruction corresponding to t .*
 - ④ *t covers the root and (perhaps) several other nodes below the root. Tile t leaves several subtrees uncovered.*
 - ⑤ *Invoke **Maximal Munch** recursively on all subtrees.*
 - ⑥ *Emit the machine instructions recorded in step ③ in order of a postorder traversal of the tiled IR tree.*



Why does the algorithm order the machine instructions in a postorder fashion?

- Step ② of **Maximal Munch** involves **tree pattern matching**. In a C implementation, one will typically find code fragments like

C implementation of Maximal Munch

```
1  void maximal_munch (T_stm s)
2  { ...
3      switch (s->kind) {
4          case T_MOVE: {
5              T_exp dst = s->u.MOVE.dst;
6              T_exp src = s->u.MOVE.src;
7
8              if (dst->kind == T_MEM) {
9                  if (dst->u.MEM->kind == T_BINOP           &&
10                     dst->u.MEM->u.BINOP.op == T_PLUS       &&
11                     dst->u.MEM->u.BINOP.right->kind == T_CONST) {
12                      T_exp e1 = dst->u.MEM->u.BINOP.left;
13                      T_exp e2 = src;
14
15                      /* detected: MOVE (MEM (BINOP (PLUS, e1, CONST (c))), e2) */
16
17                      maximal_munch (e1);
18                      maximal_munch (e2);
19                      emit ("store");
20                  }
21              }
22              else
23                  ...
```

8.1.2 Dynamic Programming

- Maximal Munch makes a **local decision** when it selects and places the next tile.
- A more ambitious approach, based on **dynamic programming** techniques, takes a global view and can produce an **optimum tiling**:

| *In an **optimum tiling**, the sum of the overall tile costs is minimum.*

- The basic idea of **dynamic programming** is that an **optimum solution of a problem p is based on optimum solutions of the subproblems of p** .
 - If $p = \text{instruction selection for an IR tree}$, then the solutions to **subproblems** of p are tilings for the tree's **subtrees**.
 - Tiling based on dynamic programming thus proceeds **bottom-up**.

- The **cost** of placing a tile t is

$$c + \sum_{i=1}^{\#leaves(t)} c_i$$

where c denotes the cost for the tile itself (remember: we assume $c = 1$ for simplicity) and c_i is the cost of the i th tiled subtree attached to t .

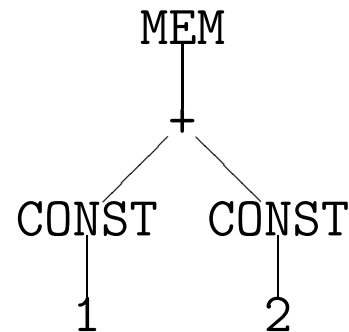
– **Note:**

We have, for example, $\#leaves \left(\begin{array}{c} + \\ \swarrow \quad \searrow \end{array} \right) = 2$ and $\#leaves \left(\begin{array}{c} \text{MEM} \\ | \\ + \\ \swarrow \quad \searrow \\ \text{CONST} \\ | \\ c \end{array} \right) = 1$.

- Instruction selection by dynamic programming:

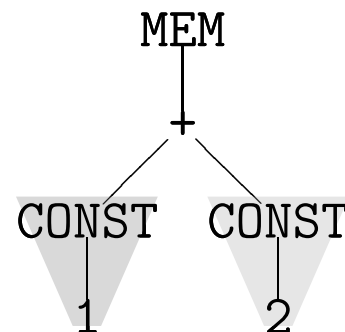
Of all the tiles that match at a node n —starting from the leaves of the IR tree—the one with **minimum cost** is chosen and the (minimum) cost of node n is remembered.

- **Example:** Select the optimum *Jouette* instructions to implement the IR tree

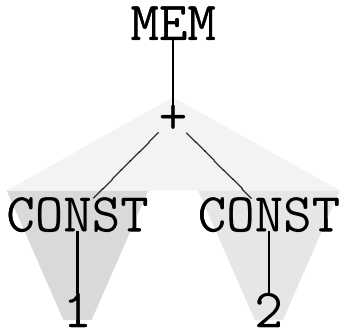
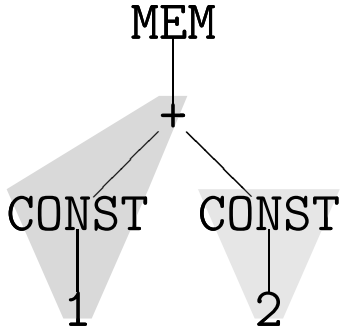
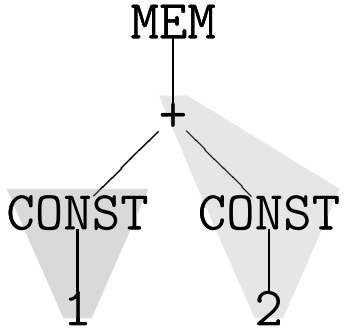


- ① Start at the leaves (the two CONST nodes). For both leaves, the only matching tile is an `addi` instruction with total cost

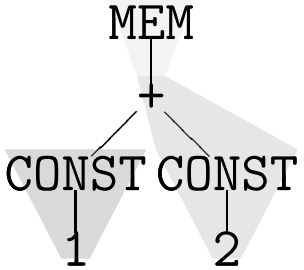
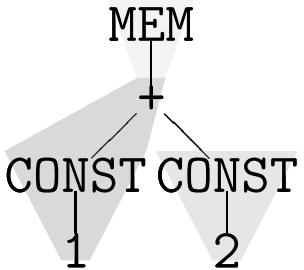
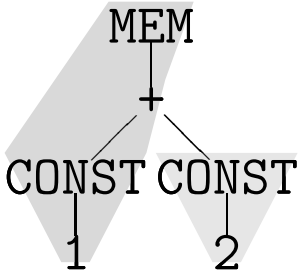
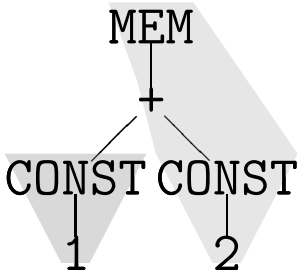
$$1 + \sum_{i=1}^0 c_i = 1 \ .$$



② Proceed up the IR tree. At the + node, several tiles match:

Tile	Instruction	Tile Cost c	Leaves Cost $\sum_i c_i$	Total Cost
	add	1	$1 + 1$	3
	addi	1	1	2
	addi	1	1	2

③ Proceed up the IR tree. Again, at the MEM node, several tiles match:

Tile	Instruction	Tile Cost c	Leaves Cost $\sum_i c_i$	Total Cost
	load	1	2	3
	load	1	2	3
	load	1	1	2
	load	1	1	2

- ④ When the tiling process has covered the IR tree root with tile t , switch to **code emission**:

emit(t):

foreach tile t_i attached to tile t do

└ $emit(t_i)$;

write instruction for tile t ;

- ⑤ The final *Jouette* assembly program is

Jouette machine instructions

1	<code>addi r_1, r_0, 1</code>
2	<code>load r_1, 2(r_1)</code>