Code generation: how?

1. IR tree nodes are like simple (3-address) machine instructions.
2. How can we generate *efficient* code for any instruction set?

Instruction selection

IR tree nodes are simple: close to machine instructions.

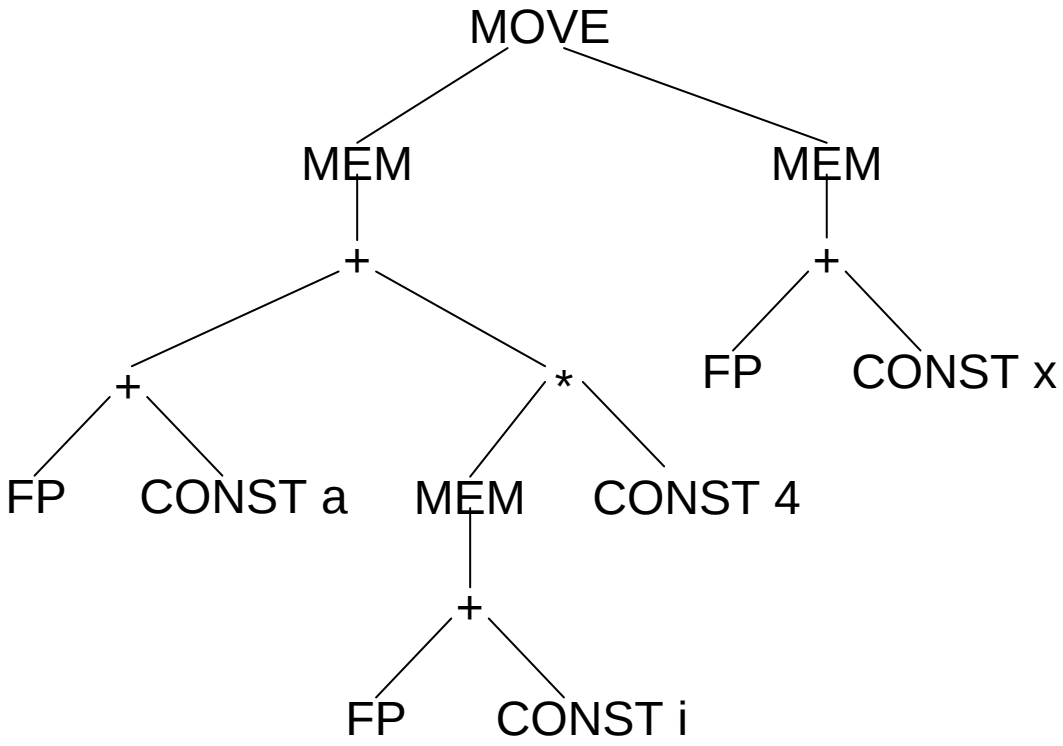
But don’t correspond exactly to specific machine’s instruction set.

Instruction selection:

- Generate code from IR tree for any instruction set.

- Try to find *best* instruction sequence.

Example tree



Generating code

For given instruction set:

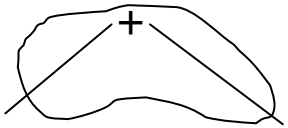
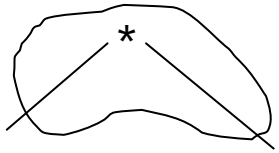
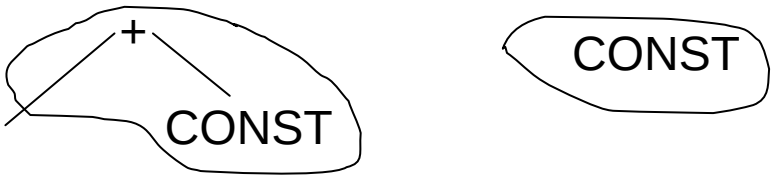
Each instruction implements small part (*tile*) of the IR tree.

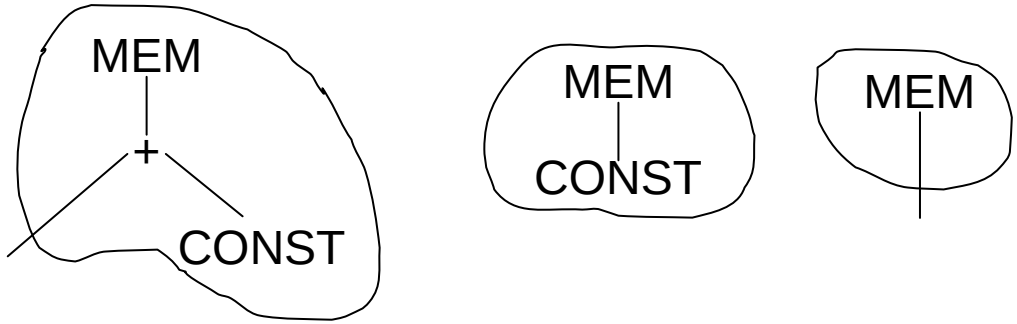
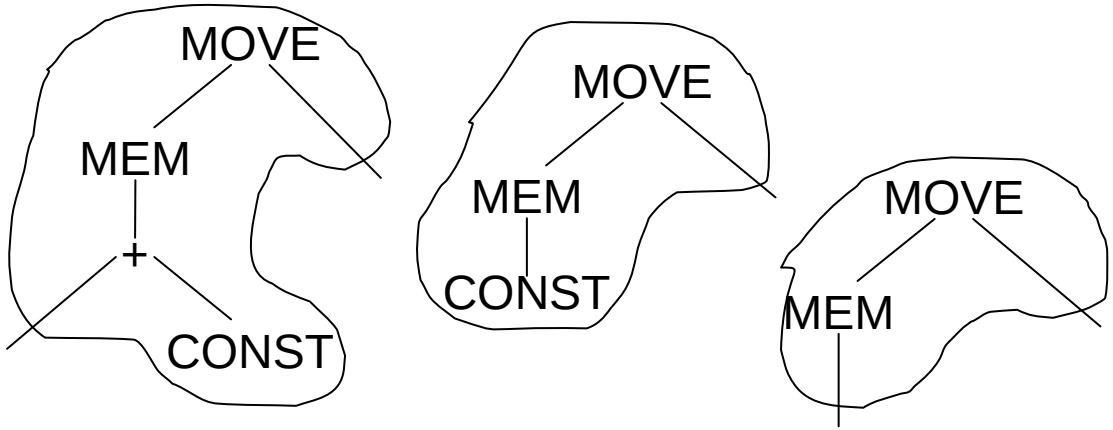
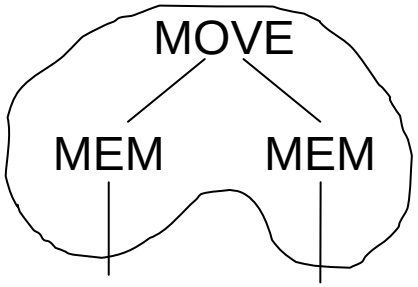
Aim of code generation:

Cover whole tree with non-overlapping tiles.

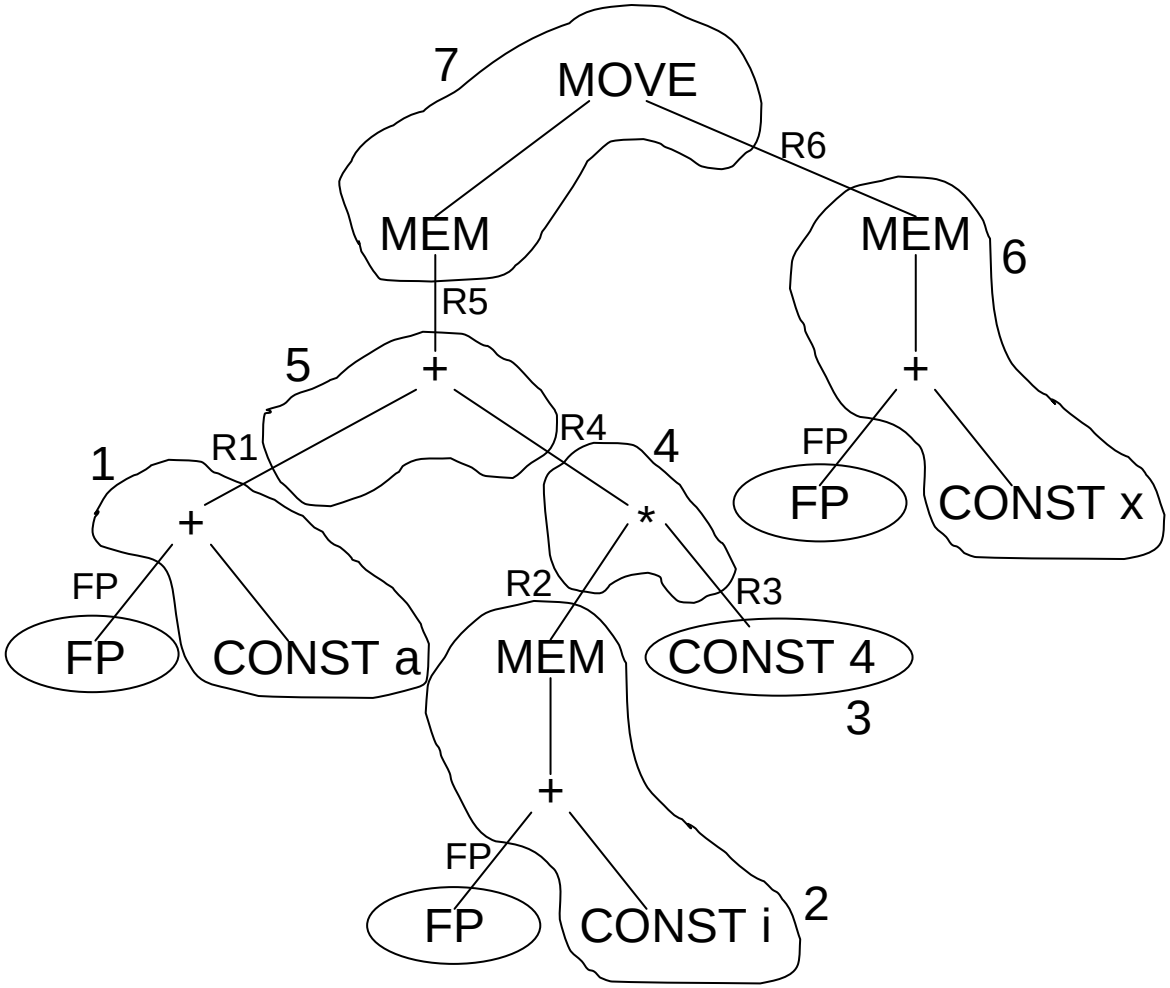
Example instruction set

Uses 3-address instructions.
Most instructions operate only on registers.

Instruction	Meaning	Tile
`ADD Ri Rj Rk`	`Ri ← Rj + Rk`	
`MUL Ri Rj Rk`	`Ri ← Rj * Rk`	
`ADDI Ri Rj c`	`Ri ← Rj + c`	

Instruction	Meaning	Tile
`LOAD Ri Rj c`	`Ri ← M[Rj+c]`	
`STORE Rj c Ri`	`M[Rj+c] ← Ri`	
`MOVEM Rj Ri`	`M[Rj] ← M[Ri]`	

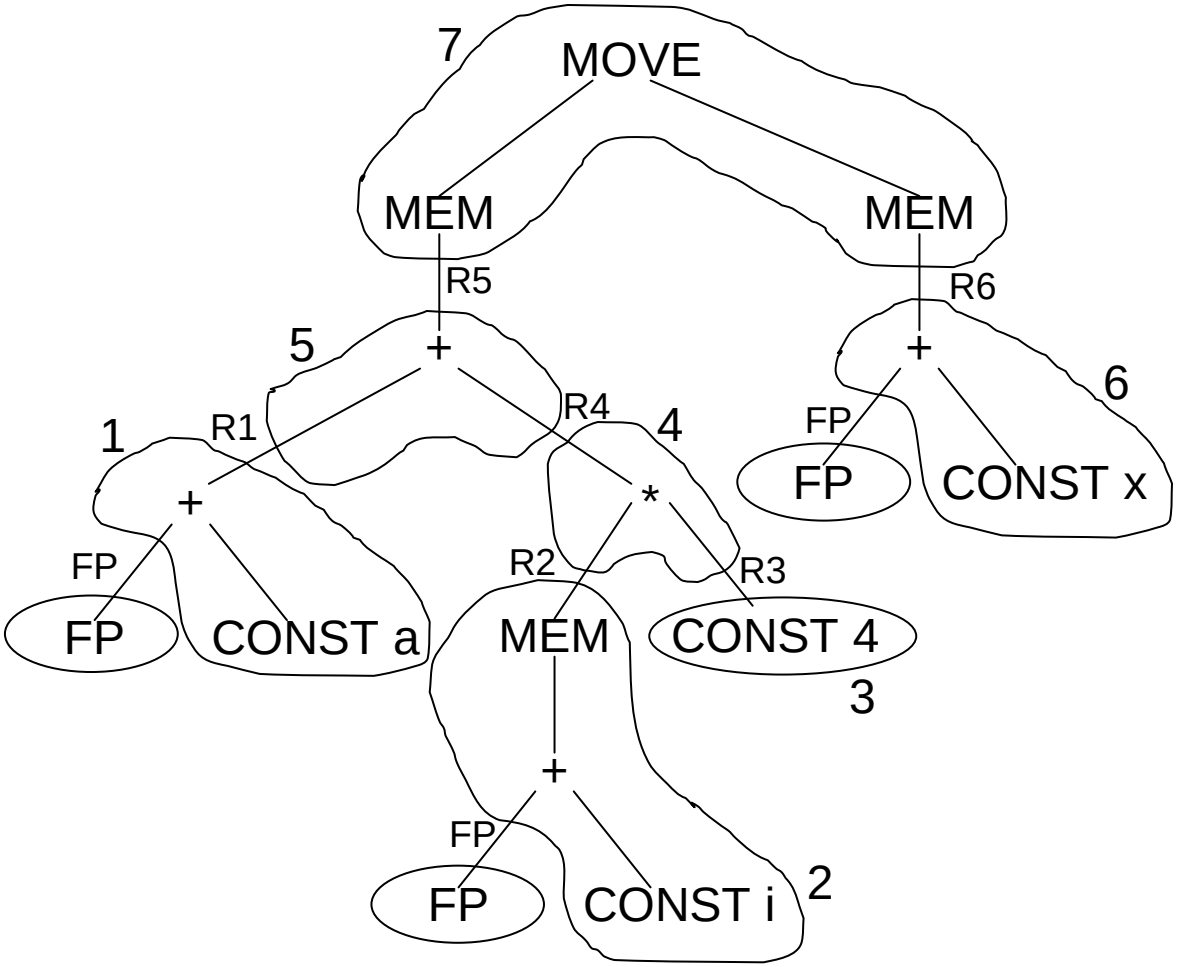
Example tiling 1



Corresponding instruction sequence:

```
1:  ADDI:  R1 ← FP + a
2:  LOAD:  R2 ← M[FP + i]
3:  ADDI:  R3 ← 4
4:  MUL:   R4 ← R2 * R3
5:  ADD:   R5 ← R1 + R4
6:  LOAD:  R6 ← M[FP + x]
7:  STORE: M[R5] ← R6
```

Example tiling 2



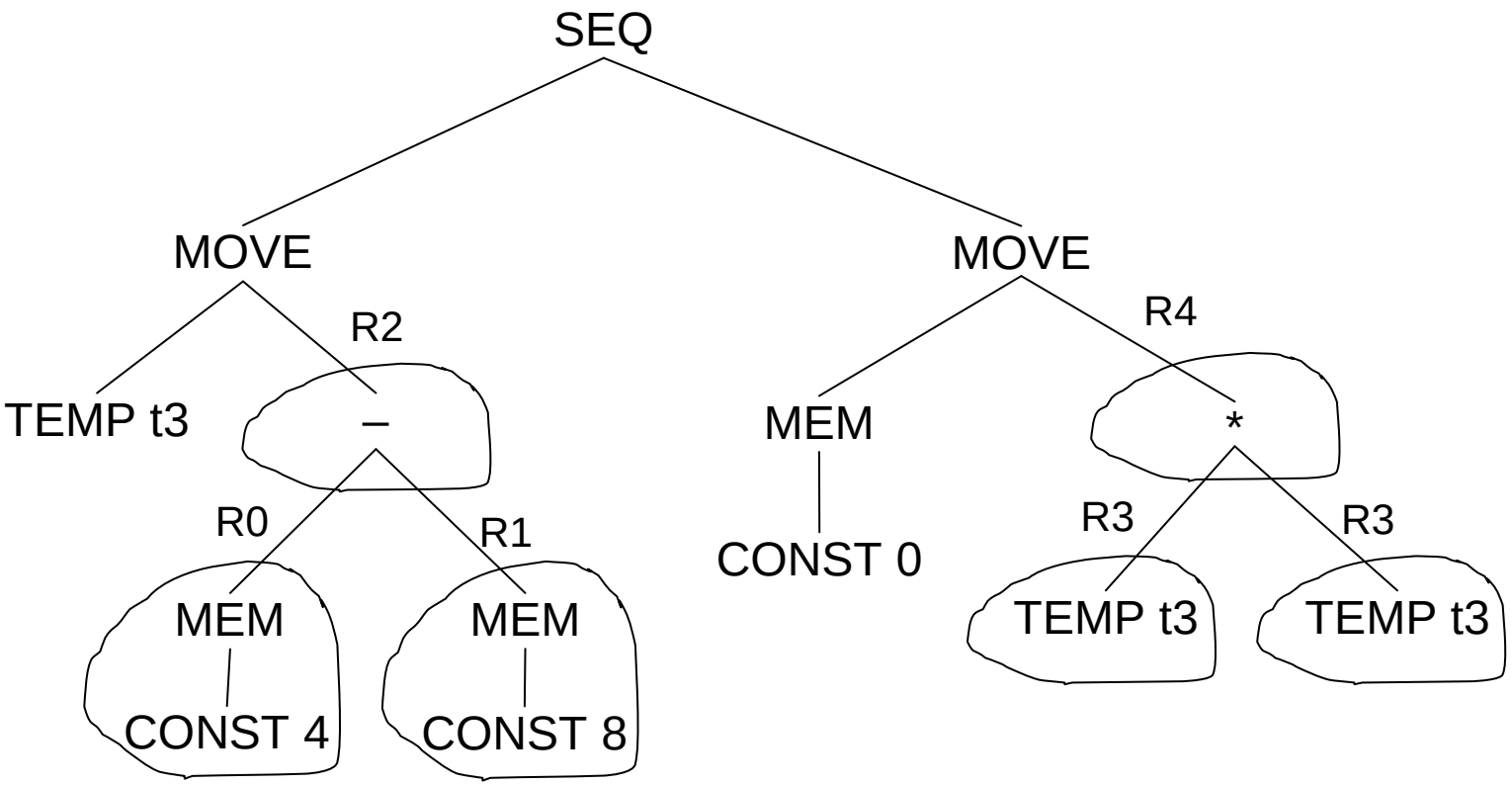
Corresponding instruction sequence:

```
1:  ADDI:  R1 ← FP + a
2:  LOAD:  R2 ← M[FP + i]
3:  ADDI:  R3 ← 4
4:  MUL:   R4 ← R2 * R3
5:  ADD:   R5 ← R1 + R4
6:  ADDI:  R6 ← FP + x
7:  MOVEM: M[R5] ← M[R6]
```

Registers

Each *tile* produces a value in a register.

TEMP leaf nodes are also allocated to (same) register. E.g.:



Unlimited number of registers available.

Separate register allocation phase will assign infinite set of registers to fixed set.

Two-address instructions

Some instruction sets are two-address:

```
ADD  Ri Rj                    Ri ← Ri + Rj
```

Needs another phase: convert each 3-address instruction to 2-address:

```
Ri ← Rj + Rk                  Ri ← Rj
                              Ri ← Ri + Rk

Ri ← Rj * Rk                  Ri ← Rj
                              Ri ← Ri * Rk

Ri ← Rj + c                   Ri ← Rj
                              Ri ← Ri + c
```

Some of the extra MOVE instructions can be eliminated during register allocation.

Finding best instruction sequence

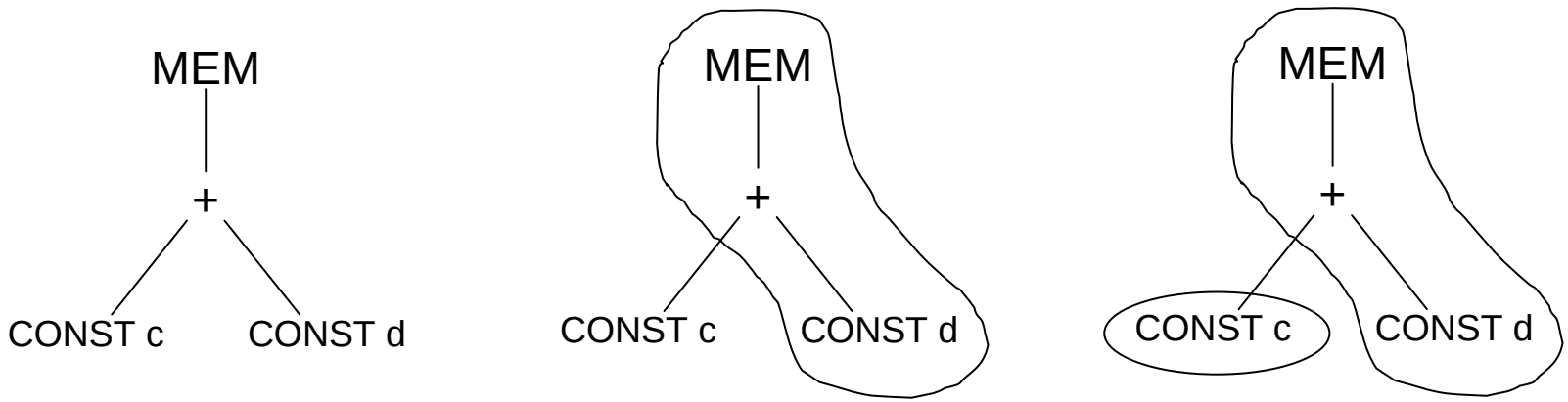
Assume larger tiles are best.

Maximal munch algorithm

Algorithm:

- Start at root of tree (top down)
- Place tile covering node and (possibly) near descendants
- Use largest tile each time
- Repeat recursively for each subtree of tile

Example:




```
1:  ADDI:  R1 ← c
2:  LOAD:  R2 ← M[R1 + d]
```

Problem:

- Doesn't necessarily produce the cheapest sequence of instructions.

Dynamic programming algorithm

Generates cheapest sequence of instructions.

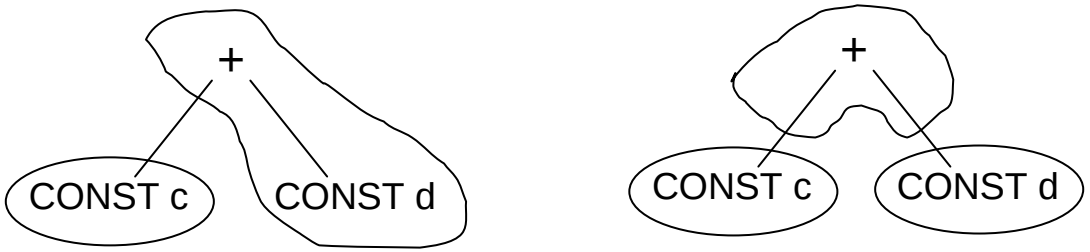
Looks at all possibilities to find cheapest one.

Algorithm:

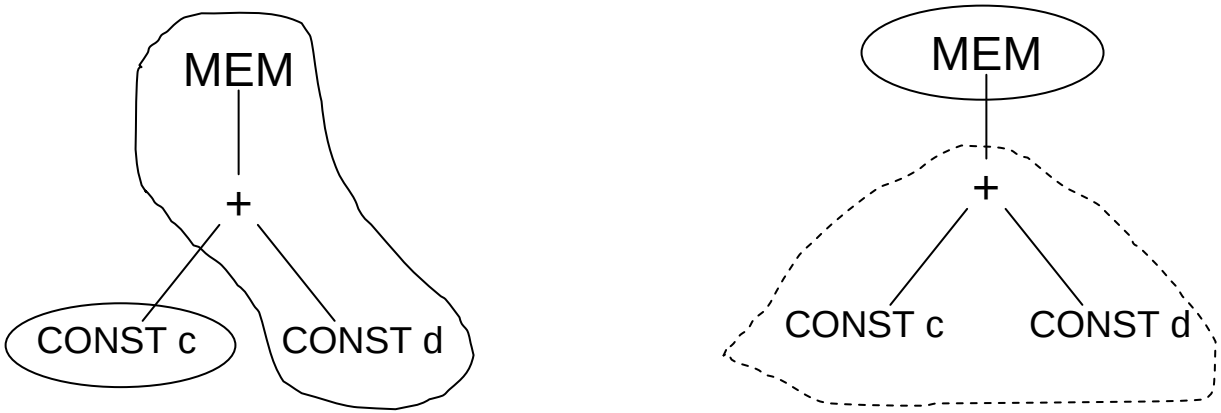
- Start at leaves of tree (bottom up)
- Try every tile matching this node
- For each tile, cost of subtree (rooted at this node) is 1 + cost of each subtree of tile
- Choose tile that minimizes cost of subtree rooted at this node

Example (step 1):

Example (step 2):



Example (step 3):



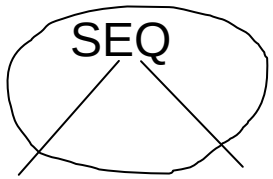
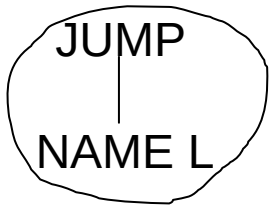
Problems:

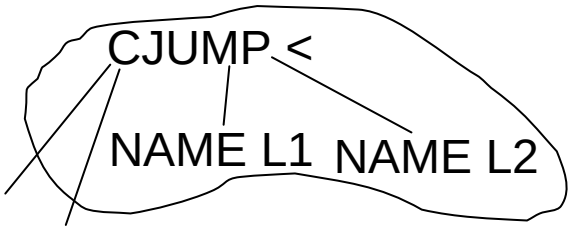
- More expensive.
- Not much improvement in practice.

Control flow statements

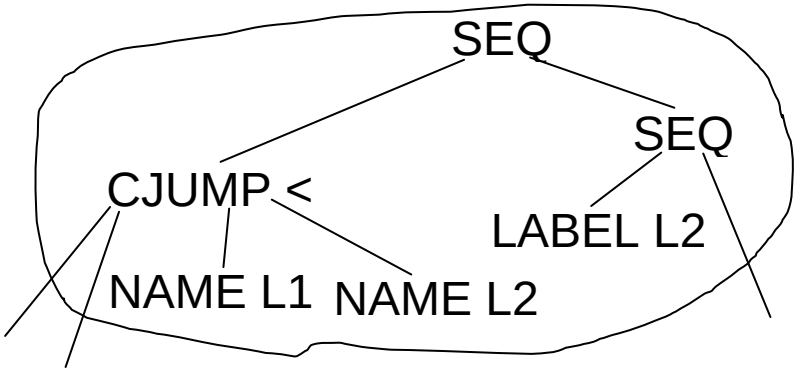
So far, we have handled expressions and MOVE statements.

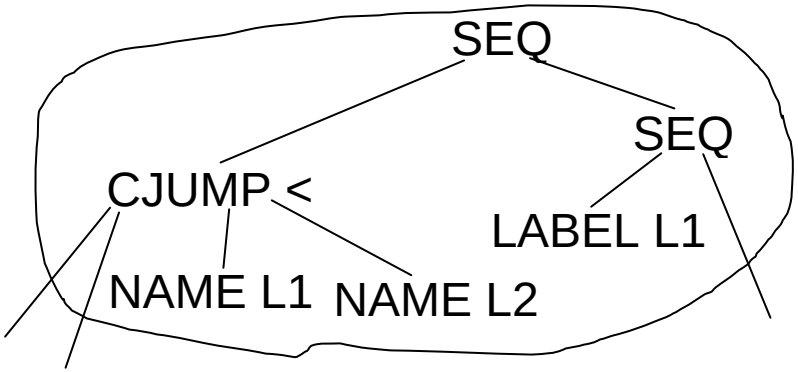
Other statements can be handled by other tiles.

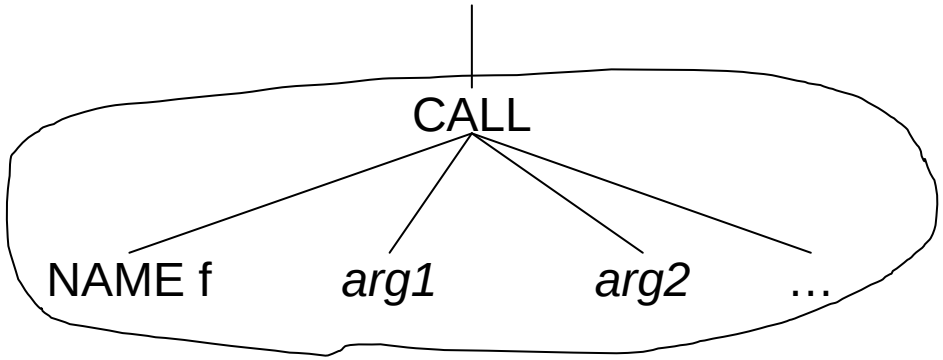
Instruction	Meaning	Tile
		
`JUMP L`	`goto L`	

`SUB Ri Rj Rk`
`BRANCHLT Ri L1`
`JUMP L2`
	`Ri ← Rj - Rk`
	`if Ri<0 goto L1`
	`goto L2`
		

Instruction	Meaning	Tile

`SUB Ri Rj Rk`
`BRANCHLT Ri L1`
`L2:`
	`Ri ← Rj - Rk`
	`if Ri<0 goto L1`
		

`SUB Ri Rj Rk`
`BRANCHGE Ri L2`
`L1:`
	`Ri ← Rj - Rk`
	`if Ri≥0 goto L2`
		

move arg1, arg2, ... to outgoing parameter registers `CALL f`		

More efficient code is generated from a CJUMP that is followed by one of its labels:

- if followed by false label, use conditional branch instruction
- if followed by true label, negate condition and use conditional branch

Worst case: translate to conditional branch and jump.