ID2202 Lecture 07
Instruction Selection

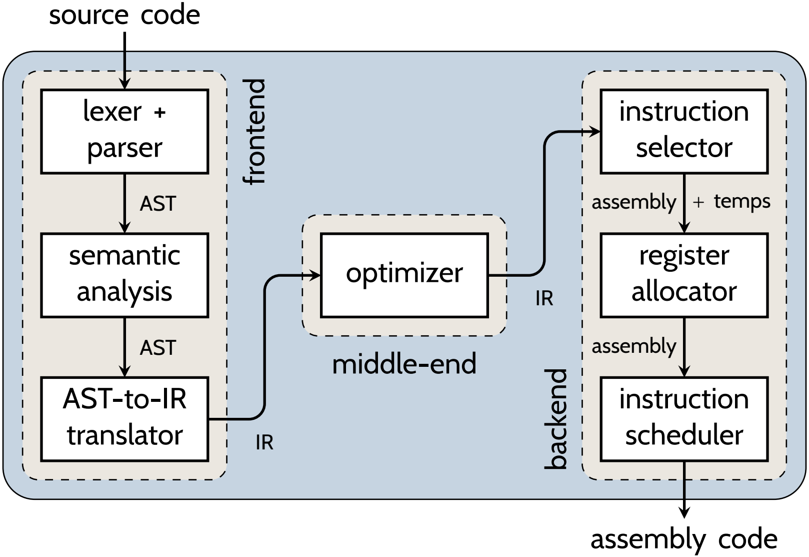
Gabriel Hjort Blindell

ghb@kth.se

Software and Computer Systems (SCS)
School of Information and Communication Technology (ICT)
KTH Royal Institute of Technology

November 21, 2016

Compilation stages



IR (Intermediate Representation)

Using terminology from Tiger book:

- IR code consists of a list of **basic blocks**
- Each basic block contains a list of **statements**
  - First statement is LABEL,
  - Last statement is either JUMP or CJUMP,
  - All other statements are either MOVEs or EXPs
- Every statement shaped like an **IR tree**

Task of instruction selection

To translate each IR tree into corresponding sequence of assembly instructions

As running example for rest of lecture . . .

```
int a[ ];
int b[ ];
    ⋮
int num = . . .
for (int i = 0; i < num; i++) {
  b[i] = a[i];
}
```

. . . will only consider this statement

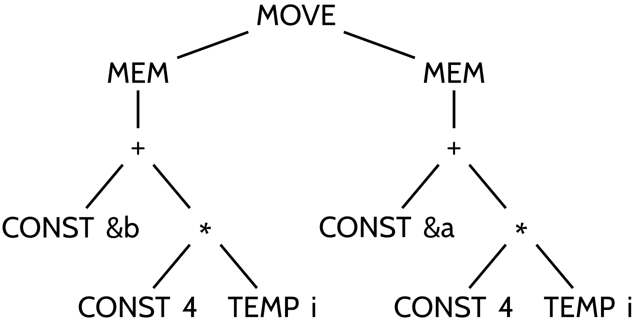
```
int a[ ];
int b[ ];
    ⋮
int num = . . .
for (int i = 0; i < num; i++) {
  b[i] = a[i];
}
```

IR tree of **b[i] = a[i];**

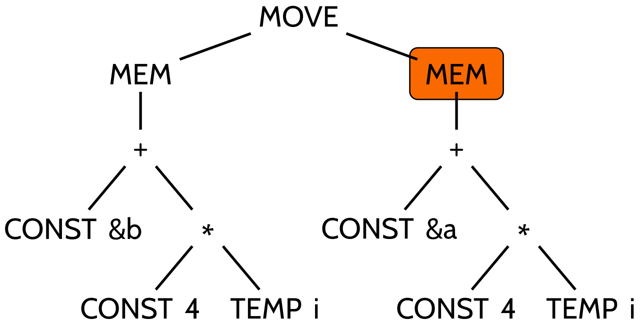
Assuming:

- Base memory address of **a** is CONST &a
- Base memory address of **b** is CONST &b
- Value of **i** is in TEMP i
- Size of **int** is CONST 4

IR tree of **b[i] = a[i];**

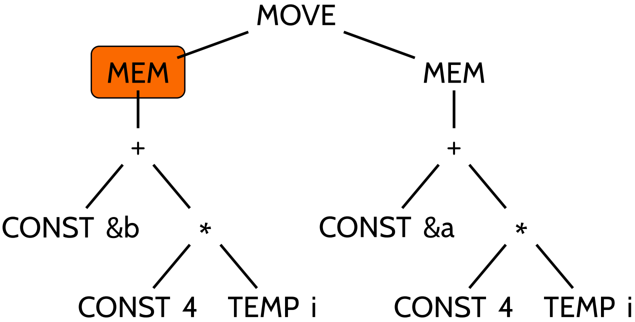


IR tree of **b[i] = a[i];**



MEM as **r-value** (right of MOVE) means “value of . . .”

IR tree of **b[i] = a[i];**



MEM as **l-value** (left of MOVE) means “location of . . .”

Target machine: *Jouette*

Notations:

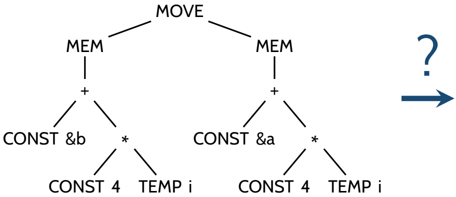
- r_i denotes “register *i*”
- t_i denotes “temporary *i*”
- *#c* denotes “integer constant *c*”
- *M*[*x*] denotes “memory value at address *x*”

Target machine: *Jouette*

Assembly instructions:		**Costs:**
ADD	$r_i \leftarrow r_j + r_k$	1
ADDI	$r_i \leftarrow r_j + \#c$	1
MUL	$r_i \leftarrow r_j * r_k$	2
LOAD	$r_i \leftarrow M[r_j + \#c]$	10
STORE	$M[r_i + \#c] \leftarrow r_j$	10
MOVEM	$M[r_i] \leftarrow M[r_j]$	12

- Jouette has more instructions (see Tiger book)
- r_0 is always contains value 0
- Cost could be number of cycles, code size, ...

Problem to solve



ADD	$r_i \leftarrow r_j + r_k$
ADDI	$r_i \leftarrow r_j + \#c$
MUL	$r_i \leftarrow r_j * r_k$
LOAD	$r_i \leftarrow M[r_j + \#c]$
STORE	$M[r_i + \#c] \leftarrow r_j$
MOVEM	$M[r_i] \leftarrow M[r_j]$

1st approach:

Macro Expansion

Fundamental idea

- Traverse IR tree bottom up
- For each IR operation *o*:
  - Emit assembly code implementing *o*
- Propagate values via temporaries
- Emission done through **expansion macros**

Macro for CONST

```
expand(CONST $c$) =
    $t_x$ = getNewTemp()
    emit("ADDI    $t_x \leftarrow r_0 + \#c$")
    setResultIsIn($t_x$)
```

- What if *c* is 0?

Better macro for CONST

```
expand(CONST $c$) =
    **if** $c$ == 0 **then**
        setResultIsIn($r_0$)
    **else**
        $t_x$ = getNewTemp()
        emit(“ADDI    $t_x \leftarrow r_0 + \#c$”)
        setResultIsIn($t_x$)
    **endif**
```

Macro for TEMP

```
expand(TEMP $t$) =
    setResultIsIn($t_t$)
```

Macros for + and $*$

```
expand(+ $E_{\text{lhs}}$ $E_{\text{rhs}}$) =
    $t_{\text{lhs}}$ = getResultOf($E_{\text{lhs}}$)
    $t_{\text{rhs}}$ = getResultOf($E_{\text{rhs}}$)
    $t_x$ = getNewTemp()
    emit(“ADD      $t_x \leftarrow t_{\text{lhs}} + t_{\text{rhs}}$”)
    setResultIsIn($t_x$)
```

- Likewise implementation for $*$

Macro for MEM

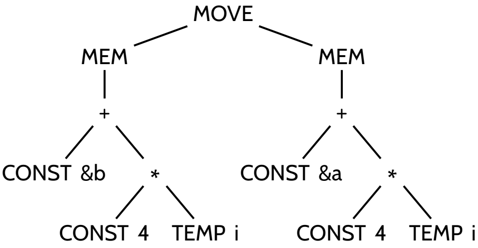
```
expand(MEM $E$) =
    if isRValue() then
        $t_x$ = getNewTemp()
        $t_y$ = getResultOf($E$)
        emit(“LOAD    $t_x \leftarrow M[t_y + \#0]$”)
        setResultIsIn($t_x$)
    else is L-value
        setResultIsIn(getResultOf($E$))
    endif
```

Macros for MOVE

```
expand(MOVE (MEM $E_{lhs}$) $E_{rhs}$) =
    $t_x$ = getResultOf($E_{lhs}$)
    $t_y$ = getResultOf($E_{rhs}$)
    emit(“STORE   $M[t_x + \#0] \leftarrow t_y$”)

expand(MOVE $E_{lhs}$ $E_{rhs}$) =
    $t_x$ = getResultOf($E_{lhs}$)
    $t_y$ = getResultOf($E_{rhs}$)
    emit(“ADD     $t_x \leftarrow r_0 + t_y$”)
```

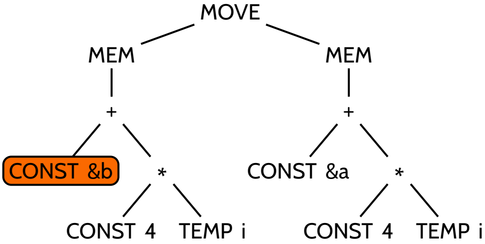
Running macro expansion on our IR tree



Assembly code:

Action:

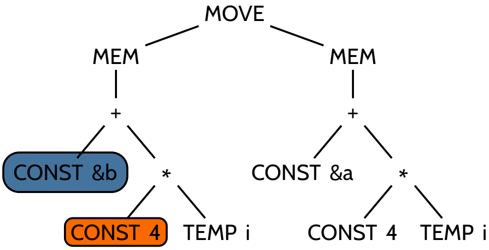
Running macro expansion on our IR tree



Assembly code:

Action: execute corresponding macro on each node

Running macro expansion on our IR tree

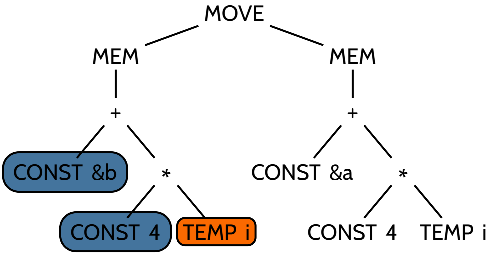


Assembly code:

ADDI $t_0 \leftarrow r_0 + $ #&b

Action: execute corresponding macro on each node

Running macro expansion on our IR tree

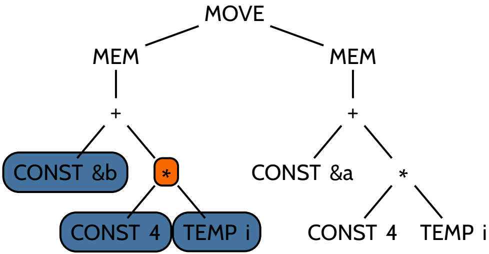


Assembly code:

ADDI $t_0 \leftarrow r_0 + \#\&b$
ADDI $t_1 \leftarrow r_0 + \#4$

Action: execute corresponding macro on each node

Running macro expansion on our IR tree

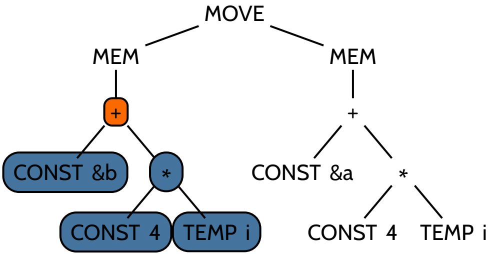


Assembly code:

ADDI $t_0 \leftarrow r_0 + \#\&b$
ADDI $t_1 \leftarrow r_0 + \#4$

Action: execute corresponding macro on each node

Running macro expansion on our IR tree



Assembly code:

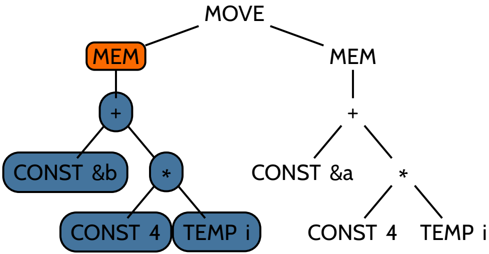
ADDI $t_0 \leftarrow r_0 + \#\&b$

ADDI $t_1 \leftarrow r_0 + \#4$

MUL $t_2 \leftarrow t_1 * t_i$

Action: execute corresponding macro on each node

Running macro expansion on our IR tree

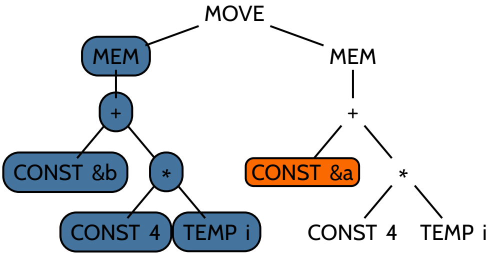


Assembly code:

ADDI	$t_0 \leftarrow r_0 + \#\&b$
ADDI	$t_1 \leftarrow r_0 + \#4$
MUL	$t_2 \leftarrow t_1 * t_i$
ADD	$t_3 \leftarrow t_0 + t_2$

Action: execute corresponding macro on each node

Running macro expansion on our IR tree



Assembly code:

ADDI $t_0 \leftarrow r_0 + \#\&b$

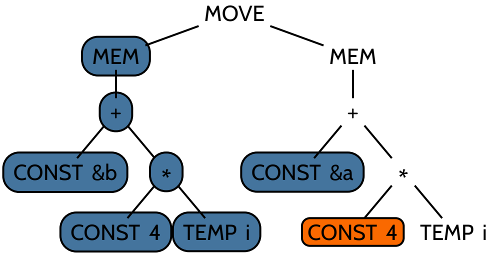
ADDI $t_1 \leftarrow r_0 + \#4$

MUL $t_2 \leftarrow t_1 * t_i$

ADD $t_3 \leftarrow t_0 + t_2$

Action: execute corresponding macro on each node

Running macro expansion on our IR tree

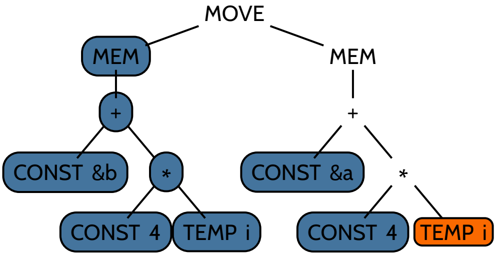


Assembly code:

ADDI	t_0	←	r_0	+	#&b
ADDI	t_1	←	r_0	+	#4
MUL	t_2	←	t_1	*	t_i
ADD	t_3	←	t_0	+	t_2
ADDI	t_4	←	r_0	+	#&a

Action: execute corresponding macro on each node

Running macro expansion on our IR tree

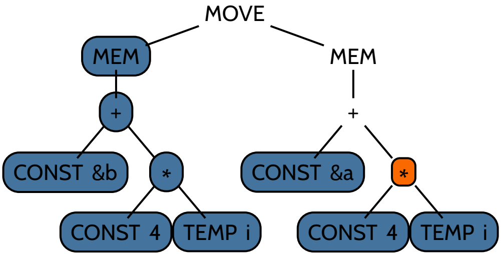


Assembly code:

ADDI $t_0 \leftarrow r_0 + \#\&b$
ADDI $t_1 \leftarrow r_0 + \#4$
MUL $t_2 \leftarrow t_1 * t_i$
ADD $t_3 \leftarrow t_0 + t_2$
ADDI $t_4 \leftarrow r_0 + \#\&a$
ADDI $t_5 \leftarrow r_0 + \#4$

Action: execute corresponding macro on each node

Running macro expansion on our IR tree

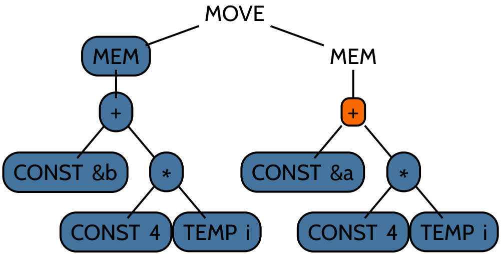


Assembly code:

```
ADDI    t0 ← r0 + #&b
ADDI    t1 ← r0 + #4
MUL     t2 ← t1 * ti
ADD     t3 ← t0 + t2
ADDI    t4 ← r0 + #&a
ADDI    t5 ← r0 + #4
```

Action: execute corresponding macro on each node

Running macro expansion on our IR tree

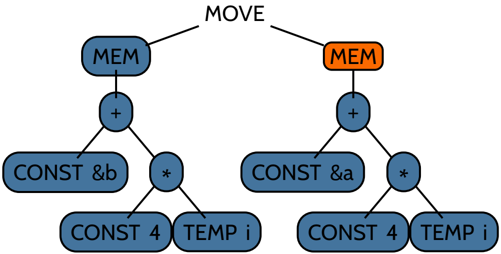


Assembly code:

ADDI $t_0 \leftarrow r_0 + \#\&b$
ADDI $t_1 \leftarrow r_0 + \#4$
MUL $t_2 \leftarrow t_1 * t_i$
ADD $t_3 \leftarrow t_0 + t_2$
ADDI $t_4 \leftarrow r_0 + \#\&a$
ADDI $t_5 \leftarrow r_0 + \#4$
MUL $t_6 \leftarrow t_5 * t_i$

Action: execute corresponding macro on each node

Running macro expansion on our IR tree

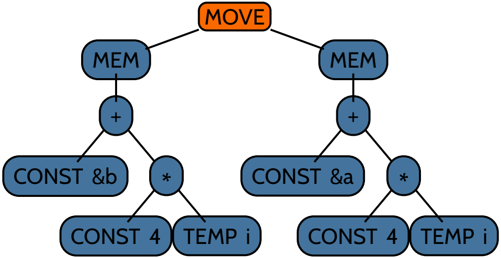


Assembly code:

ADDI $t_0 \leftarrow r_0 + \#\&b$
ADDI $t_1 \leftarrow r_0 + \#4$
MUL $t_2 \leftarrow t_1 * t_i$
ADD $t_3 \leftarrow t_0 + t_2$
ADDI $t_4 \leftarrow r_0 + \#\&a$
ADDI $t_5 \leftarrow r_0 + \#4$
MUL $t_6 \leftarrow t_5 * t_i$
ADD $t_7 \leftarrow t_4 + t_6$

Action: execute corresponding macro on each node

Running macro expansion on our IR tree



Assembly code:

ADDI $t_0 \leftarrow r_0 + \#\&b$

ADDI $t_1 \leftarrow r_0 + \#4$

MUL $t_2 \leftarrow t_1 * t_i$

ADD $t_3 \leftarrow t_0 + t_2$

ADDI $t_4 \leftarrow r_0 + \#\&a$

ADDI $t_5 \leftarrow r_0 + \#4$

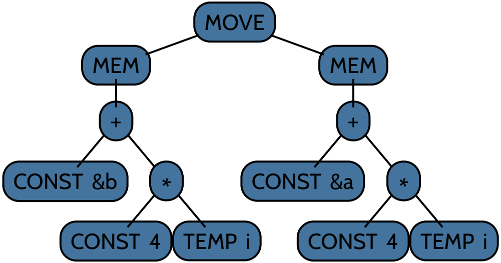
MUL $t_6 \leftarrow t_5 * t_i$

ADD $t_7 \leftarrow t_4 + t_6$

LOAD $t_8 \leftarrow M[t_7 + \#0]$

Action: execute corresponding macro on each node

Running macro expansion on our IR tree



Assembly code:

ADDI	$t_0 \leftarrow r_0 + \#\&b$
ADDI	$t_1 \leftarrow r_0 + \#4$
MUL	$t_2 \leftarrow t_1 * t_i$
ADD	$t_3 \leftarrow t_0 + t_2$
ADDI	$t_4 \leftarrow r_0 + \#\&a$
ADDI	$t_5 \leftarrow r_0 + \#4$
MUL	$t_6 \leftarrow t_5 * t_i$
ADD	$t_7 \leftarrow t_4 + t_6$
LOAD	$t_8 \leftarrow M[t_7 + \#0]$
STORE	$M[t_3 + \#0] \leftarrow t_8$

Action: done

Quality of emitted assembly code

		Costs:
ADDI	$t_0 \leftarrow r_0 + \#\&b$	1
ADDI	$t_1 \leftarrow r_0 + \#4$	1
MUL	$t_2 \leftarrow t_1 * t_i$	2
ADD	$t_3 \leftarrow t_0 + t_2$	1
ADDI	$t_4 \leftarrow r_0 + \#\&a$	1
ADDI	$t_5 \leftarrow r_0 + \#4$	1
MUL	$t_6 \leftarrow t_5 * t_i$	2
ADD	$t_7 \leftarrow t_4 + t_6$	1
LOAD	$t_8 \leftarrow M[t_7 + \#0]$	10
STORE	$M[t_3 + \#0] \leftarrow t_8$	10

$$\sum \text{cost} = 30$$

Can we do better?

		Costs:
ADDI	$t_0 \leftarrow r_0 + \#\&b$	1
ADDI	$t_1 \leftarrow r_0 + \#4$	1
MUL	$t_2 \leftarrow t_1 * t_i$	2
ADD	$t_3 \leftarrow t_0 + t_2$	1
ADDI	$t_4 \leftarrow r_0 + \#\&a$	1
ADDI	$t_5 \leftarrow r_0 + \#4$	1
MUL	$t_6 \leftarrow t_5 * t_i$	2
ADD	$t_7 \leftarrow t_4 + t_6$	1
LOAD	$t_8 \leftarrow M[t_7 + \#0]$	10
STORE	$M[t_3 + \#0] \leftarrow t_8$	10

$$\sum \text{cost} = 30$$

Suggested improvement

		Costs:
ADDI	$t_0 \leftarrow r_0 + \#\&b$	1
ADDI	$t_1 \leftarrow r_0 + \#4$	1
MUL	$t_2 \leftarrow t_1 * t_i$	2
ADD	$t_3 \leftarrow t_0 + t_2$	1
ADDI	$t_4 \leftarrow r_0 + \#\&a$	1
ADDI	$t_5 \leftarrow r_0 + \#4$	1
MUL	$t_6 \leftarrow t_5 * t_i$	2
ADD	$t_7 \leftarrow t_4 + t_6$	1
LOAD	$t_8 \leftarrow M[t_7 + \#0]$	10
STORE	$M[t_3 + \#0] \leftarrow t_8$	10

$$\sum \text{cost} = 30$$

Result of improvement

		Costs:
ADDI	$t_0 \leftarrow r_0 + \#\&b$	1
ADDI	$t_1 \leftarrow r_0 + \#4$	1
MUL	$t_2 \leftarrow t_1 * t_i$	2
ADD	$t_3 \leftarrow t_0 + t_2$	1
ADDI	$t_5 \leftarrow r_0 + \#4$	1
MUL	$t_6 \leftarrow t_5 * t_i$	2
LOAD	$t_8 \leftarrow M[t_6 + \#\&a]$	10
STORE	$M[t_3 + \#0] \leftarrow t_8$	10

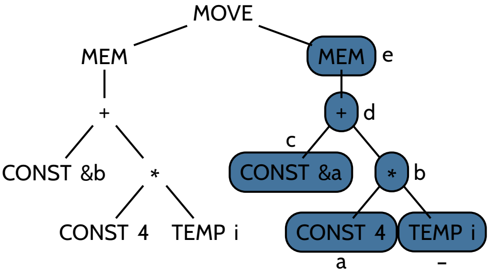
$$\sum \text{cost} = 28$$

Why did not macro expansion emit this?

		Costs:
ADDI	$t_0 \leftarrow r_0 + \#\&b$	1
ADDI	$t_1 \leftarrow r_0 + \#4$	1
MUL	$t_2 \leftarrow t_1 * t_i$	2
ADD	$t_3 \leftarrow t_0 + t_2$	1
ADDI	$t_5 \leftarrow r_0 + \#4$	1
MUL	$t_6 \leftarrow t_5 * t_i$	2
LOAD	$t_8 \leftarrow M[t_6 + \#\&a]$	10
STORE	$M[t_3 + \#0] \leftarrow t_8$	10

$$\sum \text{cost} = 28$$

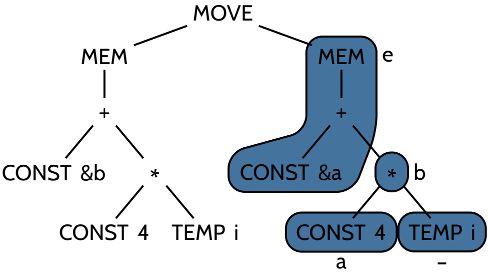
Limitation: Macro expansion emits assembly code *one* IR operation at a time . . .



Assembly code:

	ADDI	$t_0 \leftarrow r_0 + \#\&b$
	ADDI	$t_1 \leftarrow r_0 + \#4$
	MUL	$t_2 \leftarrow t_1 * t_i$
	ADD	$t_3 \leftarrow t_0 + t_2$
c	ADDI	$t_4 \leftarrow r_0 + \#\&a$
a	ADDI	$t_5 \leftarrow r_0 + \#4$
b	MUL	$t_6 \leftarrow t_5 * t_i$
d	ADD	$t_7 \leftarrow t_4 + t_6$
e	LOAD	$t_8 \leftarrow M[t_7 + \#0]$

. . . but some assembly instructions can implement *multiple* IR operations



Assembly code:

ADDI $t_0 \leftarrow r_0 + \#\&b$
ADDI $t_1 \leftarrow r_0 + \#4$
MUL $t_2 \leftarrow t_1 * t_i$
ADD $t_3 \leftarrow t_0 + t_2$

a ADDI $t_5 \leftarrow r_0 + \#4$
b MUL $t_6 \leftarrow t_5 * t_i$

e LOAD $t_8 \leftarrow M[t_6 + \#\&a]$

Instruction Selection as a Covering Problem

Let's refine the task of instruction selection

Before:

- To translate each IR tree into corresponding sequence of assembly instructions

Now:

- To **cover** each IR tree using set of **tiles** (often called **patterns**), such that:
    - every IR operation is covered by exactly one tile (no tiles overlap)
- Tile set derived from instruction set
- Valid cover is called a **tiling**
- Prefer tiling T_1 over T_2 if

$$\sum_{p \in T_1} \mathrm{cost}(p) < \sum_{p \in T_2} \mathrm{cost}(p)$$

Optimal and optimum tilings

- **Optimal tiling:**
  - If two adjacent tiles cannot be combined into single tile with lower cost
  - Can be found using greedy target algorithms
  - Often sufficient for simple architectures
- **Optimum tiling:**
  - If tiling has least cost
  - Requires non-greedy algorithms
  - Beneficial when significant cost difference between optimum and optimal tilings

In literature only Tiger book uses these notions

Subproblems to solve

- **Tile matching:**
  - Which tiles could cover what parts of the IR tree?
- **Tile selection:**
  - Which tiles to choose to form a tiling?
- **Optimality:**
  - How to find optimal/optimum tiling?

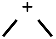
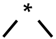
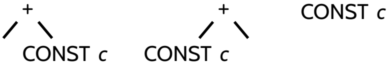
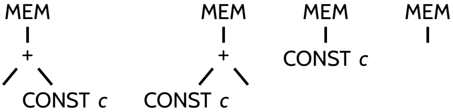
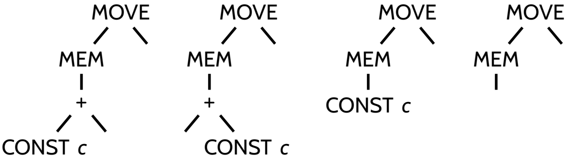
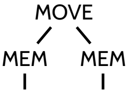
Revisiting macro expansion

- Requires all tiles to consist of single IR operation
  - Trivial to match tiles
- Existed only one tile per IR operation
  - Trivial to form tilings
- Can only find one tiling
  - Suboptimal by design

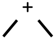
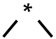
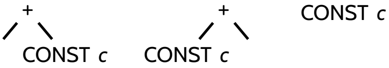
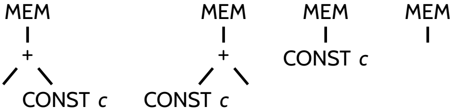
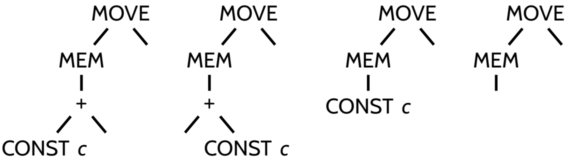
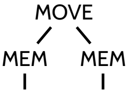
Tiles set of macro expansion

Tile	Instructions	
TEMP *c*	–	
CONST *c*	ADDI	$t_x \leftarrow r_0 + \#c$
+	ADD	$t_x \leftarrow t_y + t_z$
*	MUL	$t_x \leftarrow t_y * t_z$
MEM	– *or* LOAD	 $t_x \leftarrow M[t_y + \#0]$
MOVE	STORE *or* ADD	$M[t_x + \#0] \leftarrow t_y$ $t_x \leftarrow r_0 + t_y$

Full tile set for our *Jouette* instructions

Instruction		Tiles
–		TEMP *c*
ADD	$t_x \leftarrow t_y + t_z$	
MUL	$t_x \leftarrow t_y * t_z$	
ADDI	$t_x \leftarrow t_y + \#c$	
LOAD	$t_x \leftarrow M[t_y + \#c]$	 MEM + CONST *c* MEM + CONST *c* MEM CONST *c* MEM
STORE	$M[t_x + \#c] \leftarrow t_y$	 MOVE MEM + CONST *c* MOVE MEM + CONST *c* MOVE MEM CONST *c* MOVE MEM
MOVEM	$M[t_x] \leftarrow M[t_y]$	

Problem: How to use these efficiently?

Instruction		Tiles
–		TEMP *c*
ADD	$t_x \leftarrow t_y + t_z$	
MUL	$t_x \leftarrow t_y * t_z$	
ADDI	$t_x \leftarrow t_y + \#c$	
LOAD	$t_x \leftarrow M[t_y + \#c]$	 MEM + CONST c MEM + CONST c MEM CONST c MEM
STORE	$M[t_x + \#c] \leftarrow t_y$	 MOVE MEM + CONST c MOVE MEM + CONST c MOVE MEM CONST c MOVE MEM
MOVEM	$M[t_x] \leftarrow M[t_y]$	

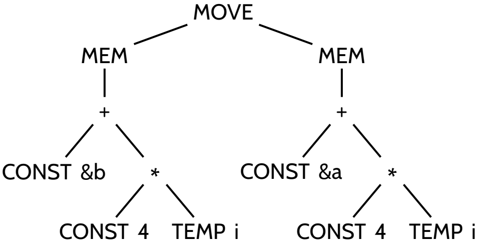
2nd approach:

Maximum Munch

Fundamental idea

- To find optimal tiling:
    1. Start at root node
    2. Find largest tile that matches at root
    3. Cover nodes matched by tile
    4. Repeat for all subtrees
- To emit assembly code:
    - Traverse IR tree bottom up
    - For each tile *t* in tiling:
        - Emit instruction corresponding to *t*

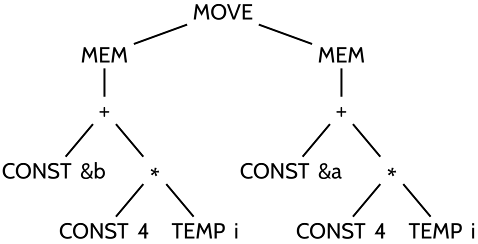
Running maximum munch on our IR tree



Assembly code:

Action:

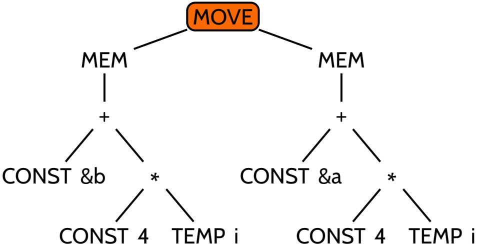
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

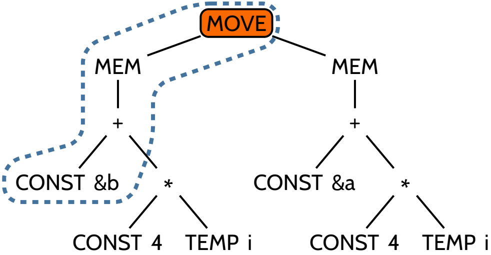
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

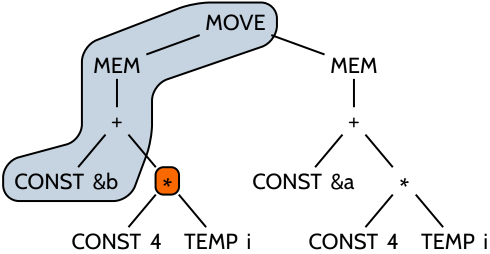
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

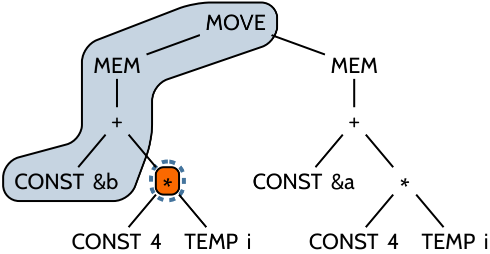
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

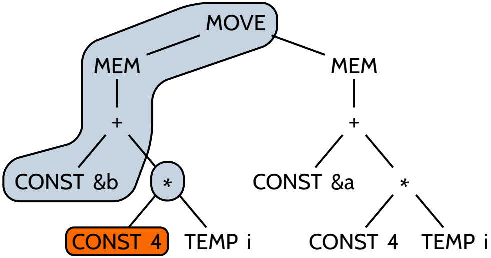
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

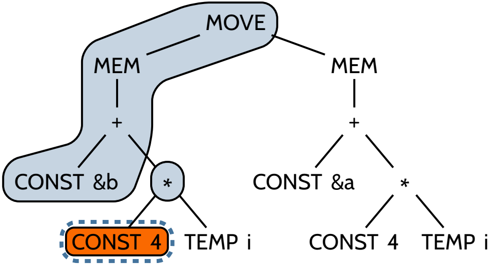
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

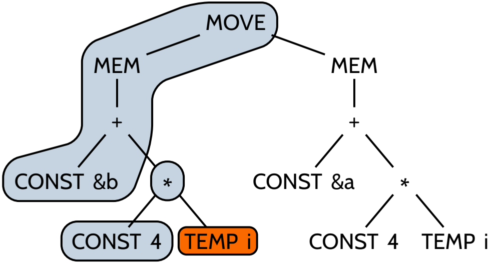
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

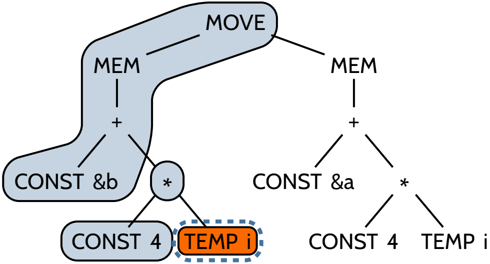
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

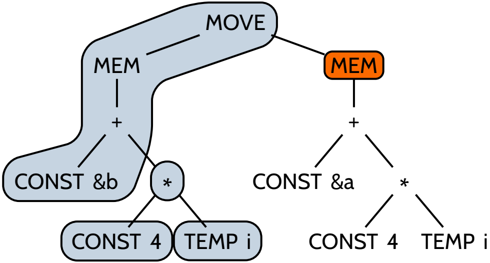
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

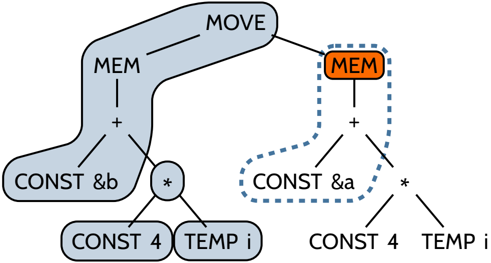
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

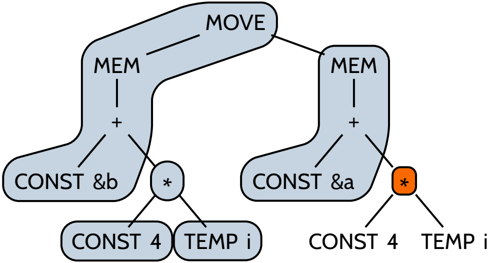
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

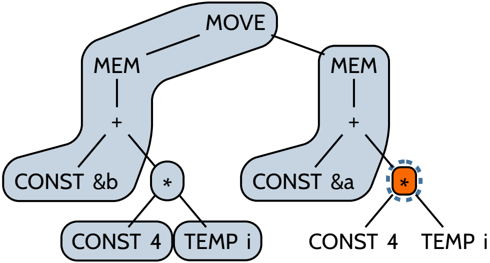
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

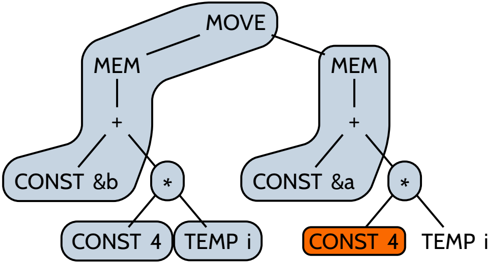
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

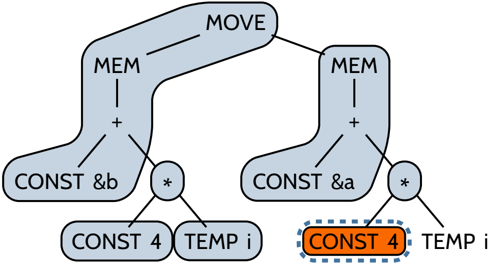
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

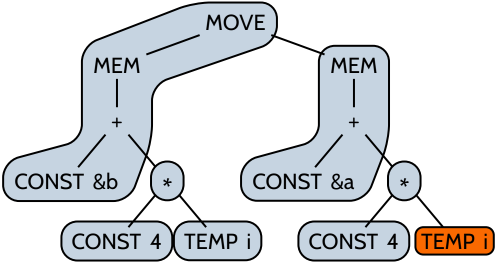
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

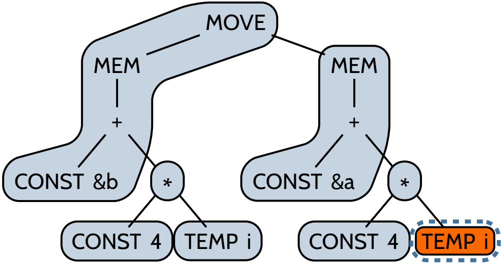
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

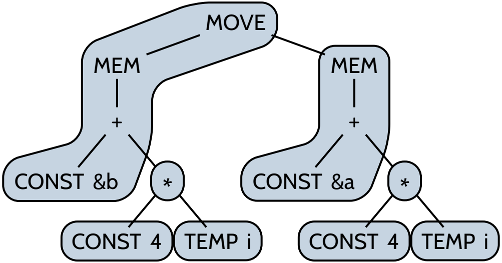
Running maximum munch on our IR tree



Assembly code:

Action: find largest matches

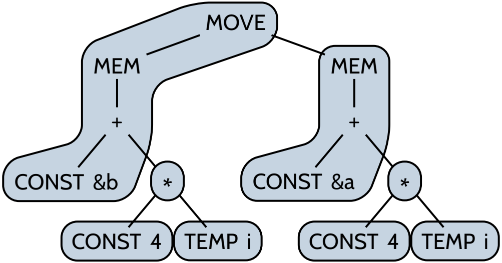
Running maximum munch on our IR tree



Assembly code:

Action: done finding matches

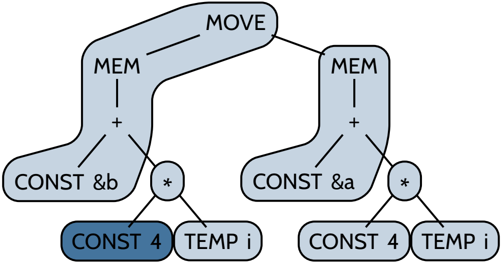
Running maximum munch on our IR tree



Assembly code:

Action: emit assembly instructions

Running maximum munch on our IR tree

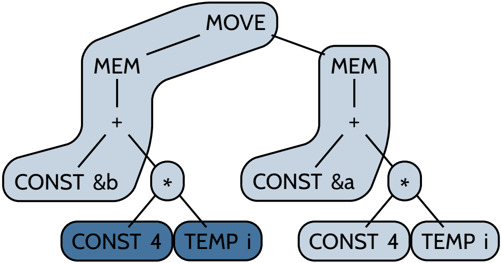


Assembly code:

ADDI $t_0 \leftarrow r_0 + \#4$

Action: emit assembly instructions

Running maximum munch on our IR tree

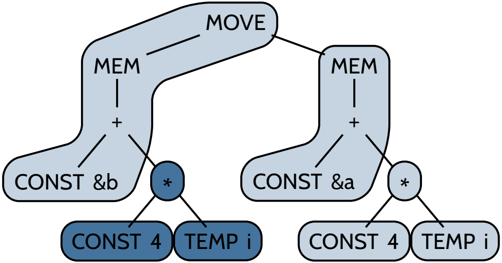


Assembly code:

ADDI $t_0 \leftarrow r_0 + \#4$

Action: emit assembly instructions

Running maximum munch on our IR tree

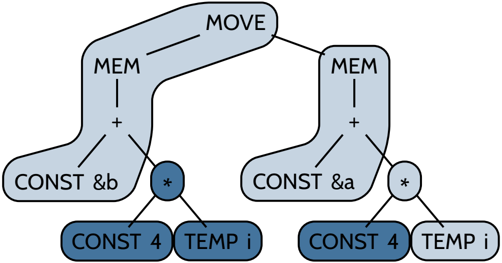


Assembly code:

ADDI	$t_0 \leftarrow r_0 + \#4$
MUL	$t_1 \leftarrow t_0 * t_i$

Action: emit assembly instructions

Running maximum munch on our IR tree

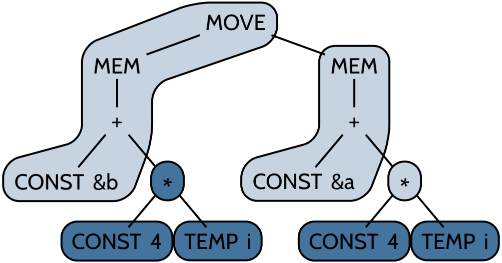


Assembly code:

```
ADDI    t0 ← r0 + #4
MUL     t1 ← t0 * ti
ADDI    t2 ← r0 + #4
```

Action: emit assembly instructions

Running maximum munch on our IR tree

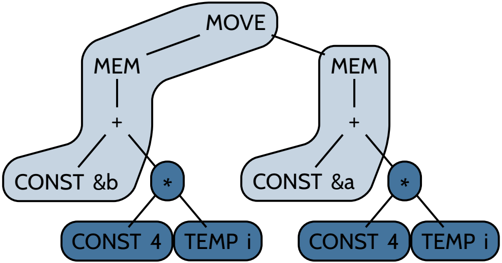


Assembly code:

```
ADDI    t₀ ← r₀ + #4
MUL     t₁ ← t₀ * tᵢ
ADDI    t₂ ← r₀ + #4
```

Action: emit assembly instructions

Running maximum munch on our IR tree

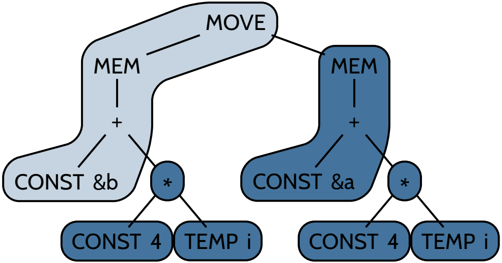


Assembly code:

ADDI	$t_0 \leftarrow r_0 + \#4$
MUL	$t_1 \leftarrow t_0 * t_i$
ADDI	$t_2 \leftarrow r_0 + \#4$
MUL	$t_3 \leftarrow t_2 * t_i$

Action: emit assembly instructions

Running maximum munch on our IR tree

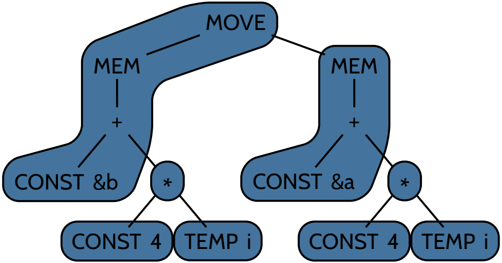


Assembly code:

ADDI	$t_0 \leftarrow r_0 + \#4$
MUL	$t_1 \leftarrow t_0 * t_i$
ADDI	$t_2 \leftarrow r_0 + \#4$
MUL	$t_3 \leftarrow t_2 * t_i$
LOAD	$t_4 \leftarrow M[t_3 + \#\&a]$

Action: emit assembly instructions

Running maximum munch on our IR tree



Assembly code:

ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$
ADDI $t_2 \leftarrow r_0 + \#4$
MUL $t_3 \leftarrow t_2 * t_i$
LOAD $t_4 \leftarrow M[t_3 + \#\&a]$
STORE $M[t_1 + \#\&b] \leftarrow t_4$

Action: done

Quality of emitted assembly code

		Costs:
ADDI	$t_0 \leftarrow r_0 + \#4$	1
MUL	$t_1 \leftarrow t_0 * t_i$	2
ADDI	$t_2 \leftarrow r_0 + \#4$	1
MUL	$t_3 \leftarrow t_2 * t_i$	2
LOAD	$t_4 \leftarrow M[t_3 + \#\&a]$	10
STORE	$M[t_1 + \#\&b] \leftarrow t_4$	10

$$\sum \text{cost} = 26$$

Cost Reduced By 4. So What?

```
for (int i = 0; i < num; i++) {
  b[i] = a[i];
}
```

- If program dominated by **b[i] = a[i]**:
  - 13% cost reduction → 13% execution time reduction

3rd approach:

Tree Parsing

Fundamental idea

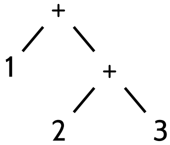
- Derive **tree grammar** from tile set:
    - Each tile yields a production
- **Generate** LR parser from tree grammar
- To find tile matches and optimal tiling:
    1. Transform IR tree into an **IR string**
    2. Run LR parser on IR string
- To emit assembly code:
    - When performing a reduction:
        - Emit instruction corresponding to reduced production

Transforming trees into strings

- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)

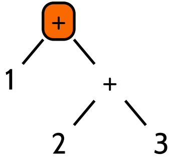
Transforming trees into strings

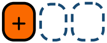
- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



Transforming trees into strings

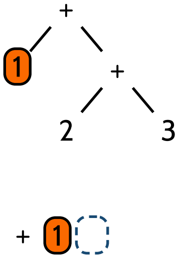
- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)





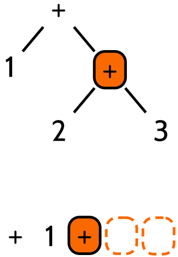
Transforming trees into strings

- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



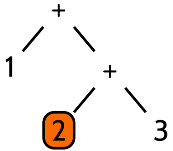
Transforming trees into strings

- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



Transforming trees into strings

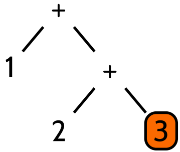
- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



$+\ \ 1\ \ +\ \ \boxed{2}\ \square$

Transforming trees into strings

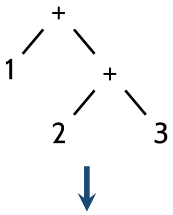
- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



+ 1 + 2 3

Transforming trees into strings

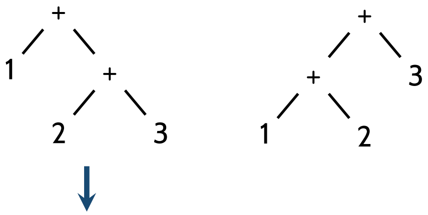
- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



+ 1 + 2 3

Transforming trees into strings

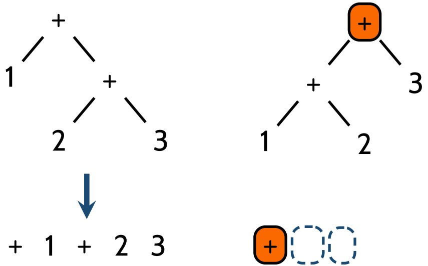
- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



+ 1 + 2 3

Transforming trees into strings

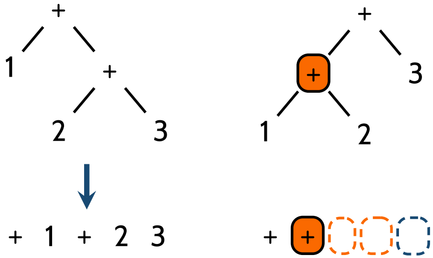
- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



+ 1 + 2 3

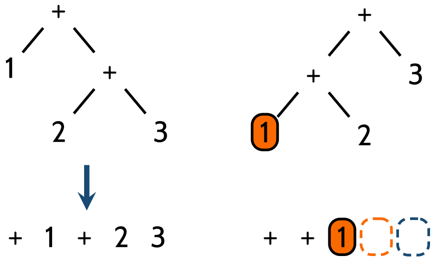
Transforming trees into strings

- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



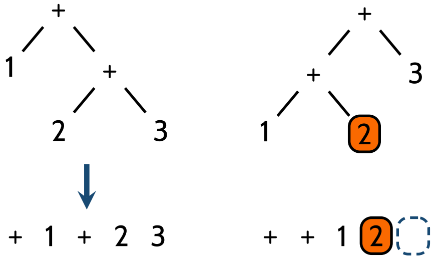
Transforming trees into strings

- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



Transforming trees into strings

- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)

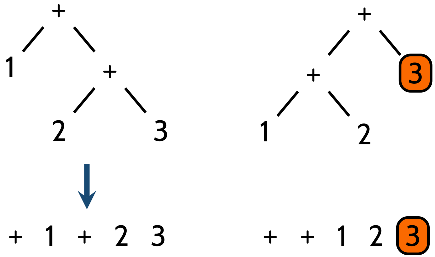


+ 1 + 2 3

+ + 1 2

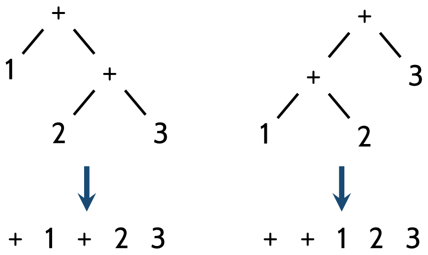
Transforming trees into strings

- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



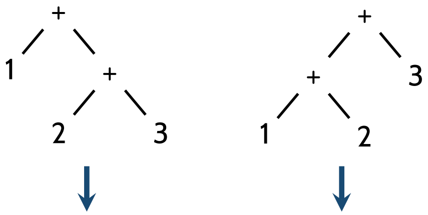
Transforming trees into strings

- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



Transforming trees into strings

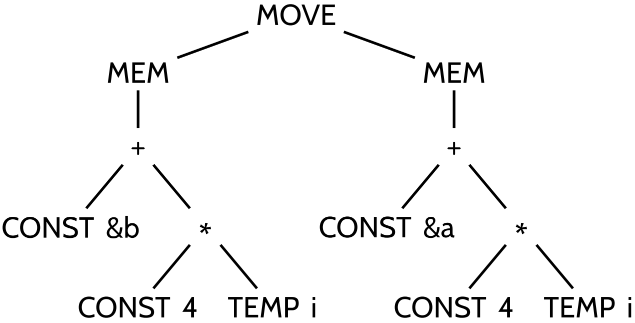
- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



+ 1 + 2 3   + + 1 2 3

- Equivalent to depth-first, left-to-right tree traversal

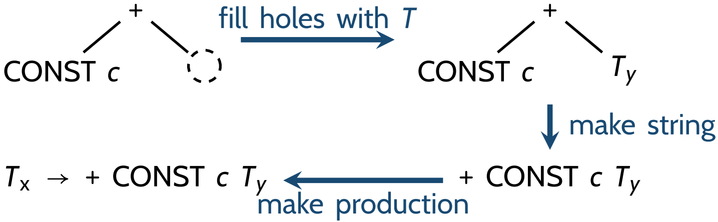
Transforming our IR tree into an IR string



```
MOVE MEM + CONST &b * CONST 4 TEMP i
     MEM + CONST &a * CONST 4 TEMP i
```

Deriving the tree grammar

- Introduce nonterminal T to represent temporaries
  - T_x refers to temporary x
- For each tile:



+

CONST c

fill holes with T

+

CONST c T_y

make string

$T_x \rightarrow + \text{ CONST } c\ T_y$

make production

$+ \text{ CONST } c\ T_y$

  - In relation to expansion macros:
    - RHS T corresponds to: t = getResultOf(E_{rhs})
    - LHS T corresponds to: t = getNewTemp()

      . . .

      setResultIsIn(t)
- Introduce start symbol:
  - $S \rightarrow T\$$

Tree grammar for *Jouette*

Instruction		Productions
–		$T_x \rightarrow \text{TEMP } x$
ADD	$t_{new} \leftarrow t_y + t_z$	$T \rightarrow + \ T_y \ T_z$
MUL	$t_{new} \leftarrow t_y * t_z$	$T \rightarrow * \ T_y \ T_z$
ADDI	$t_{new} \leftarrow t_y + \#c$	$T \rightarrow + \ T_y \ \text{CONST } c$ $T \rightarrow + \ \text{CONST } c \ T_y$ $T \rightarrow \text{CONST } c$
LOAD	$t \leftarrow M[t_y + \#c]$	$T \rightarrow \text{MEM} + T_y \ \text{CONST } c$ $T \rightarrow \text{MEM} + \text{CONST } c \ T_y$ $T \rightarrow \text{MEM CONST } c$ $T \rightarrow \text{MEM } T_y$
STORE	$M[t_x + \#c] \leftarrow t_y$	$T \rightarrow \text{MOVE MEM} + \text{CONST } c \ T_x \ T_y$ $T \rightarrow \text{MOVE MEM} + T_x \ \text{CONST } c \ T_y$ $T \rightarrow \text{MOVE MEM CONST } c \ T_y$ $T \rightarrow \text{MOVE MEM } T_x \ T_y$
MOVEM	$M[t_x] \leftarrow M[t_y]$	$T \rightarrow \text{MOVE MEM } T_x \ \text{MEM } T_y$

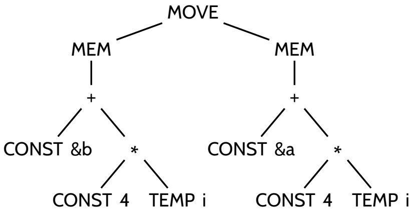
Tree grammars derived from assembly instructions often highly ambiguous

Means more than one correct sequence of instructions

- Resolving shift-reduce conflicts:
    - Always shift
- Resolving reduce-reduce conflicts:
    - Choose longest production

Heuristic above equivalent to maximum munch

Running tree parsing on our IR tree



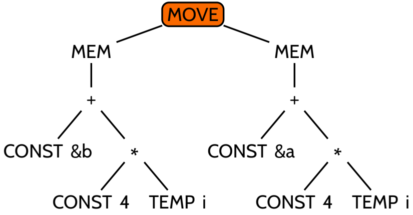
Assembly code:

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action:

Stack:

Running tree parsing on our IR tree



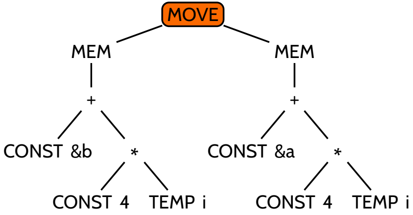
Assembly code:

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: move to next symbol

Stack:

Running tree parsing on our IR tree



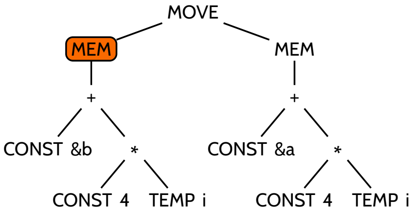
Assembly code:

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: shift

Stack: MOVE

Running tree parsing on our IR tree



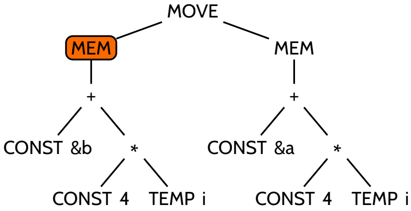
Assembly code:

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: move to next symbol

Stack: MOVE

Running tree parsing on our IR tree



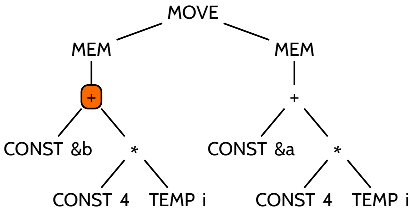
Assembly code:

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: shift

Stack: MOVE MEM

Running tree parsing on our IR tree



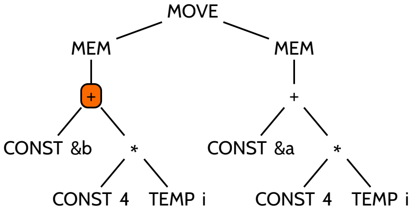
Assembly code:

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: move to next symbol

Stack: MOVE MEM

Running tree parsing on our IR tree



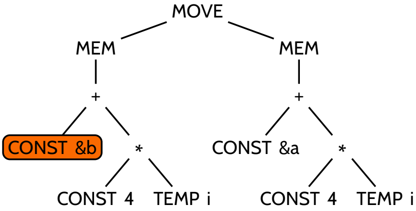
Assembly code:

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: shift

Stack: MOVE MEM +

Running tree parsing on our IR tree



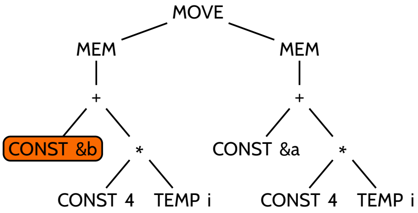
Assembly code:

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: move to next symbol

Stack: MOVE MEM +

Running tree parsing on our IR tree



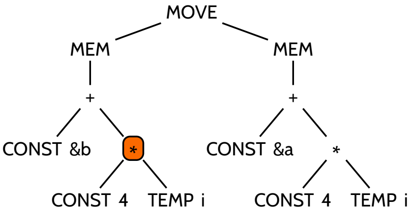
Assembly code:

MOVE MEM + CONST &b ∗ CONST 4 TEMP i
MEM + CONST &a ∗ CONST 4 TEMP i $

Action: shift

Stack: MOVE MEM + CONST &b

Running tree parsing on our IR tree



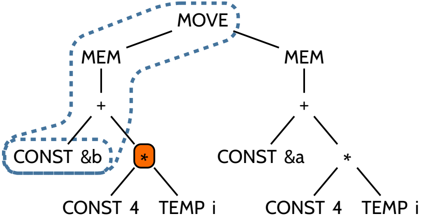
Assembly code:

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: move to next symbol

Stack: MOVE MEM + CONST &b

Running tree parsing on our IR tree



Assembly code:

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

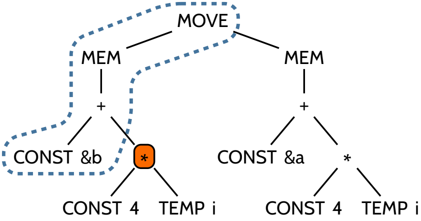
Action: shift-reduce conflict!

Stack: MOVE MEM + CONST &b *

T → CONST *c* *reducible now*

T → MOVE MEM + CONST *c* T_x T_y *may be reducible later*

Running tree parsing on our IR tree



Assembly code:

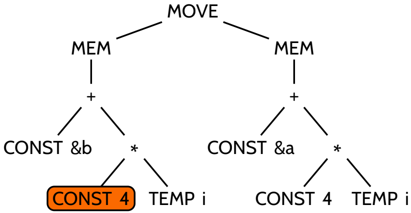
MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: always shift

Stack: MOVE MEM + CONST &b *

$T \rightarrow$ MOVE MEM + CONST c T_x T_y *hope for later reduction*

Running tree parsing on our IR tree



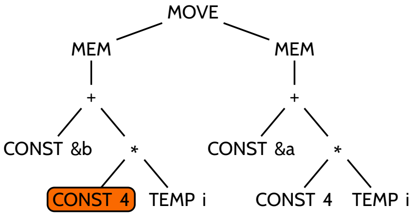
Assembly code:

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: move to next symbol

Stack: MOVE MEM + CONST &b *

Running tree parsing on our IR tree



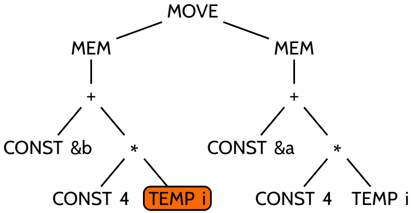
Assembly code:

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: shift

Stack: MOVE MEM + CONST &b * CONST 4

Running tree parsing on our IR tree



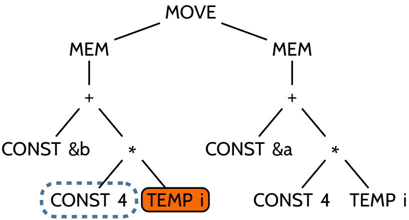
Assembly code:

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: move to next symbol

Stack: MOVE MEM + CONST &b * CONST 4

Running tree parsing on our IR tree



Assembly code:

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

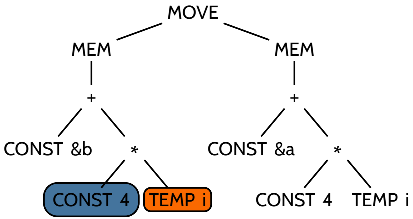
Action: reduce

Stack: MOVE MEM + CONST &b * CONST 4

$T \rightarrow$ CONST c

due to $T \rightarrow * \ T_y \ T_z$

Running tree parsing on our IR tree



MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

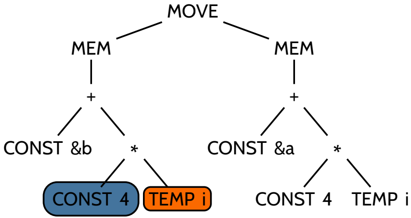
Action:

Stack: MOVE MEM + CONST &b * T_0

Assembly code:

ADDI $t_0 \leftarrow r_0 + \#4$

Running tree parsing on our IR tree



Assembly code:

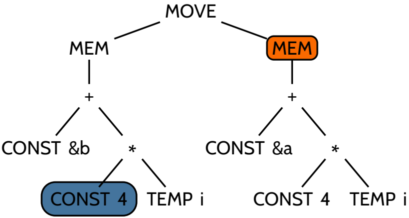
ADDI $t_0 \leftarrow r_0 + \#4$

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: shift

Stack: MOVE MEM + CONST &b * T_0 TEMP i

Running tree parsing on our IR tree



Assembly code:

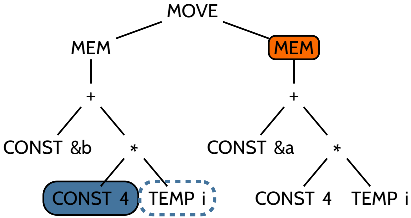
ADDI $t_0 \leftarrow r_0 + \#4$

MOVE MEM + CONST &b $*$ CONST 4 TEMP i
MEM + CONST &a $*$ CONST 4 TEMP i $

Action: move to next symbol

Stack: MOVE MEM + CONST &b $*$ T_0 TEMP i

Running tree parsing on our IR tree



Assembly code:

ADDI $t_0 \leftarrow r_0 + \#4$

MOVE MEM + CONST &b ∗ CONST 4 TEMP i
MEM + CONST &a ∗ CONST 4 TEMP i $

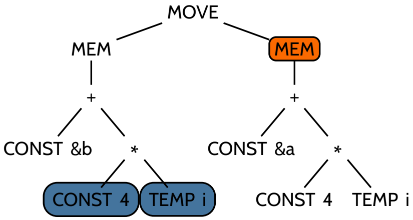
Action: reduce

Stack: MOVE MEM + CONST &b ∗ T_0 TEMP i

$T_t \longrightarrow$ TEMP t

due to $T \longrightarrow * \ T_y \ T_z$

Running tree parsing on our IR tree



Assembly code:

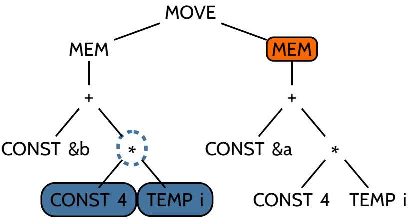
ADDI $t_0 \leftarrow r_0 + \#4$

MOVE MEM + CONST &b ∗ CONST 4 TEMP i
MEM + CONST &a ∗ CONST 4 TEMP i $

Action:

Stack: MOVE MEM + CONST &b ∗ T_0 T_i

Running tree parsing on our IR tree



Assembly code:

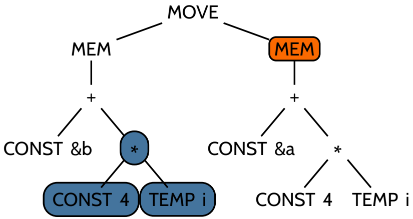
ADDI $t_0 \leftarrow r_0 + \#4$

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: reduce

Stack: MOVE MEM + CONST &b $* \; T_0 \; T_i$
$T \rightarrow * \; T_y \; T_z$

Running tree parsing on our IR tree



Assembly code:

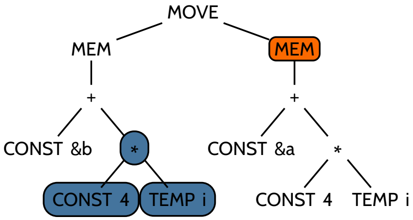
ADDI	t_0	←	r_0	+	#4
MUL	t_1	←	t_0	$*$	t_i

MOVE MEM + CONST &b $*$ CONST 4 TEMP i
MEM + CONST &a $*$ CONST 4 TEMP i $

Action:

Stack: MOVE MEM + CONST &b T_1

Running tree parsing on our IR tree



Assembly code:

ADDI	t_0	←	r_0	+	#4	
MUL	t_1	←	t_0	$*$	t_i	

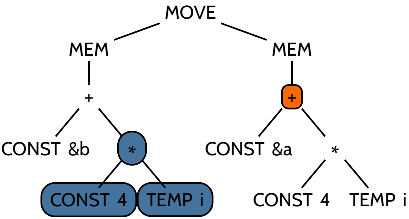
MOVE MEM + CONST &b $*$ CONST 4 TEMP i

MEM + CONST &a $*$ CONST 4 TEMP i $

Action: shift

Stack: MOVE MEM + CONST &b T_1 MEM

Running tree parsing on our IR tree



Assembly code:

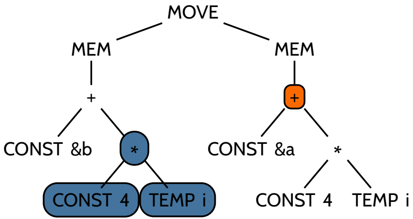
ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: move to next symbol

Stack: MOVE MEM + CONST &b T_1 MEM

Running tree parsing on our IR tree



Assembly code:

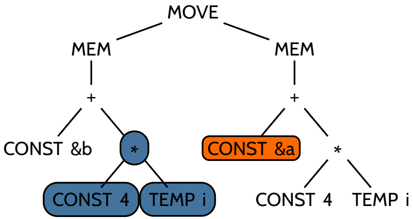
ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: shift

Stack: MOVE MEM + CONST &b T_1 MEM +

Running tree parsing on our IR tree



Assembly code:

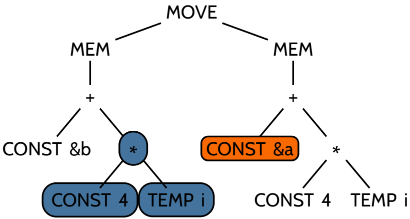
```
ADDI    t0 ← r0 + #4
MUL     t1 ← t0 * ti
```

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: move to next symbol

Stack: MOVE MEM + CONST &b T_1 MEM +

Running tree parsing on our IR tree



Assembly code:

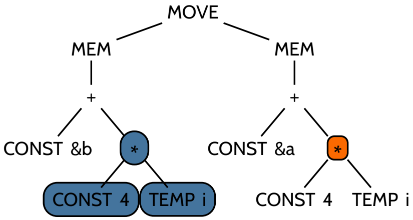
ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: shift

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a

Running tree parsing on our IR tree



Assembly code:

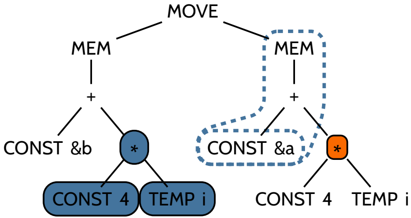
ADDI	$t_0 \leftarrow r_0 + \#4$
MUL	$t_1 \leftarrow t_0 * t_i$

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: move to next symbol

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a

Running tree parsing on our IR tree



Assembly code:

```
ADDI    t0 ← r0 + #4
MUL     t1 ← t0 * ti
```

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

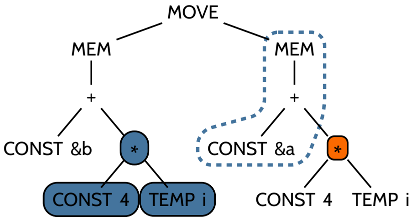
Action: shift-reduce conflict!

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a *

$T \rightarrow$ CONST c *reducible now*

$T_x \rightarrow$ MEM + CONST c T_y *may be reducible later*

Running tree parsing on our IR tree



Assembly code:

ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$

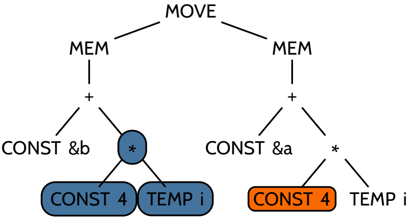
MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: always shift

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a *

$T_x \rightarrow$ MEM + CONST c T_y *hope for later reduction*

Running tree parsing on our IR tree



Assembly code:

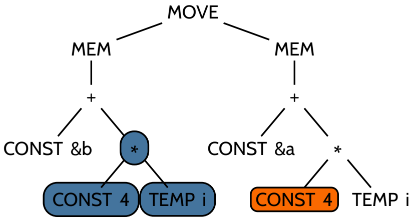
ADDI	$t_0 \leftarrow r_0 + \#4$
MUL	$t_1 \leftarrow t_0 * t_i$

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: move to next symbol

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a *

Running tree parsing on our IR tree



Assembly code:

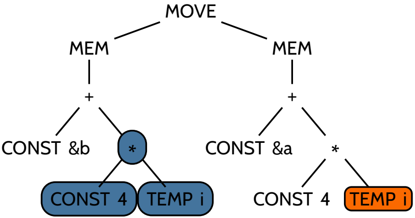
ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: shift

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a * CONST 4

Running tree parsing on our IR tree



Assembly code:

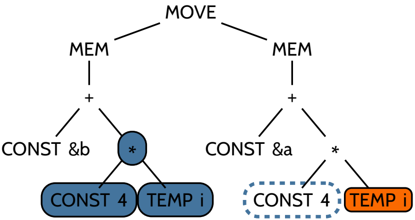
ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: move to next symbol

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a * CONST 4

Running tree parsing on our IR tree



Assembly code:

ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

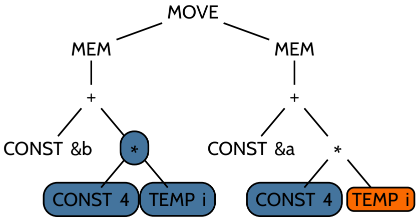
Action: reduce

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a * CONST 4

$T \rightarrow$ CONST c

due to $T \rightarrow * \; T_y \; T_z$

Running tree parsing on our IR tree



Assembly code:

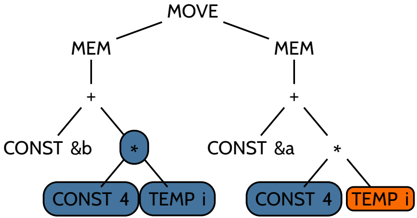
ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$
ADDI $t_2 \leftarrow r_0 + \#4$

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action:

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a * T_2

Running tree parsing on our IR tree



Assembly code:

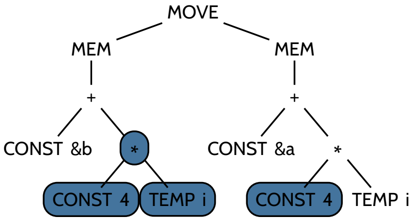
ADDI	t_0	←	r_0	+	#4
MUL	t_1	←	t_0	*	t_i
ADDI	t_2	←	r_0	+	#4

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: shift

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a * T_2 TEMP i

Running tree parsing on our IR tree



Assembly code:

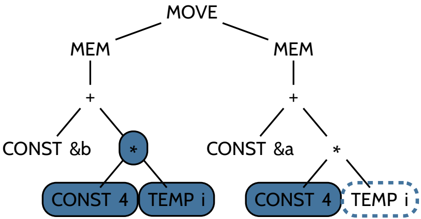
ADDI	$t_0 \leftarrow r_0 + \#4$
MUL	$t_1 \leftarrow t_0 * t_i$
ADDI	$t_2 \leftarrow r_0 + \#4$

MOVE MEM + CONST &b ∗ CONST 4 TEMP i
MEM + CONST &a ∗ CONST 4 TEMP i $

Action: move to next symbol

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a ∗ T_2 TEMP i

Running tree parsing on our IR tree



Assembly code:

ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$
ADDI $t_2 \leftarrow r_0 + \#4$

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

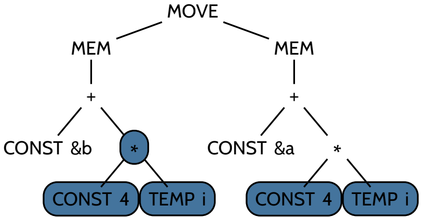
Action: reduce

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a * T_2 TEMP i

$T_t \rightarrow$ TEMP t

due to $T \rightarrow * \; T_y \; T_z$

Running tree parsing on our IR tree



Assembly code:

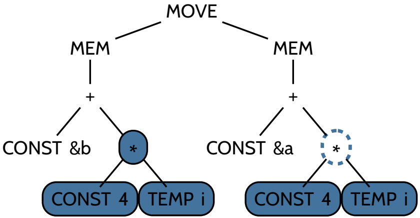
ADDI	t_0	←	r_0	+	#4
MUL	t_1	←	t_0	$*$	t_i
ADDI	t_2	←	r_0	+	#4

MOVE MEM + CONST &b $*$ CONST 4 TEMP i
MEM + CONST &a $*$ CONST 4 TEMP i $

Action:

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a $*$ T_2 T_i

Running tree parsing on our IR tree



Assembly code:

ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$
ADDI $t_2 \leftarrow r_0 + \#4$

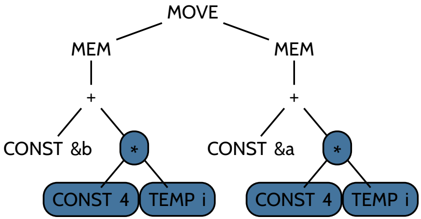
MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $\boxed{\$}$

Action: reduce

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a $\boxed{* \; T_2 \; T_i}$

$\boxed{T \longrightarrow * \; T_y \; T_z}$

Running tree parsing on our IR tree



Assembly code:

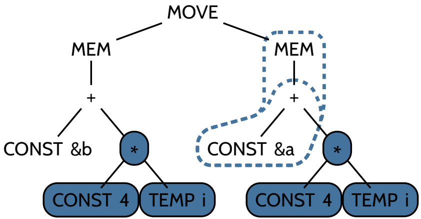
ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$
ADDI $t_2 \leftarrow r_0 + \#4$
MUL $t_3 \leftarrow t_2 * t_i$

MOVE MEM + CONST &b ∗ CONST 4 TEMP i
MEM + CONST &a ∗ CONST 4 TEMP i $

Action:

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a T_3

Running tree parsing on our IR tree



Assembly code:

ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$
ADDI $t_2 \leftarrow r_0 + \#4$
MUL $t_3 \leftarrow t_2 * t_i$

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

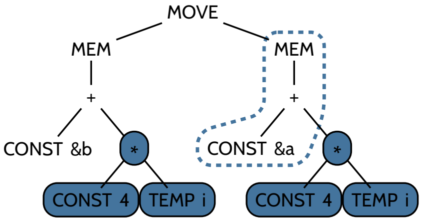
Action: reduce-reduce conflict!

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a T_3

$T \rightarrow$ + CONST c T_y

$T \rightarrow$ MEM + CONST c T_y

Running tree parsing on our IR tree



Assembly code:

```
ADDI   t0 ← r0 + #4
MUL    t1 ← t0 * ti
ADDI   t2 ← r0 + #4
MUL    t3 ← t2 * ti
```

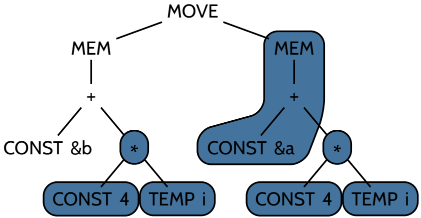
MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: reduce longest production

Stack: MOVE MEM + CONST &b T_1 MEM + CONST &a T_3

$T \rightarrow$ MEM + CONST c T_y

Running tree parsing on our IR tree



Assembly code:

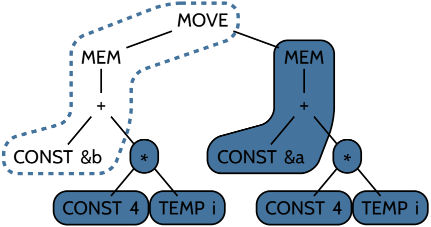
```
ADDI   t0 ← r0 + #4
MUL    t1 ← t0 * ti
ADDI   t2 ← r0 + #4
MUL    t3 ← t2 * ti
LOAD   t4 ← M[t3 + #&a]
```

MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action:

Stack: MOVE MEM + CONST &b T_1 T_4

Running tree parsing on our IR tree



MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: reduce

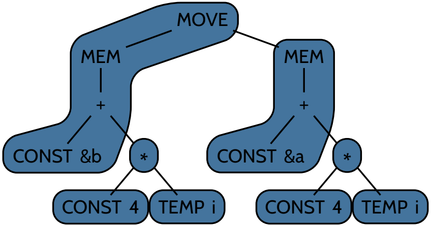
Stack: MOVE MEM + CONST &b T_1 T_4

$T \rightarrow$ MOVE MEM + CONST c T_x T_y

Assembly code:

```
ADDI   t0 ← r0 + #4
MUL    t1 ← t0 * ti
ADDI   t2 ← r0 + #4
MUL    t3 ← t2 * ti
LOAD   t4 ← M[t3 + #&a]
```

Running tree parsing on our IR tree



MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

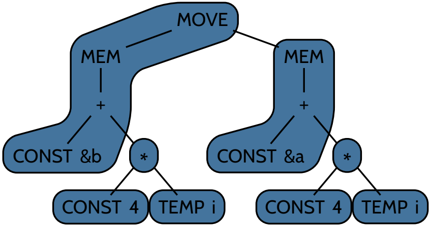
Action:

Stack: T

Assembly code:

```
ADDI    t0 ← r0 + #4
MUL     t1 ← t0 * ti
ADDI    t2 ← r0 + #4
MUL     t3 ← t2 * ti
LOAD    t4 ← M[t3 + #&a]
STORE   M[t1 + #&b] ← t4
```

Running tree parsing on our IR tree



MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: accept

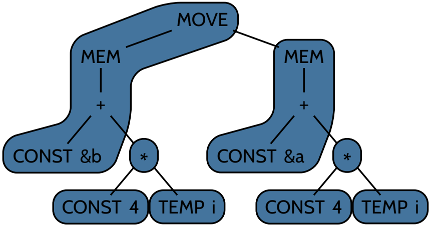
Stack: T

$S \rightarrow T\ \$$

Assembly code:

ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$
ADDI $t_2 \leftarrow r_0 + \#4$
MUL $t_3 \leftarrow t_2 * t_i$
LOAD $t_4 \leftarrow M[t_3 + \#\&a]$
STORE $M[t_1 + \#\&b] \leftarrow t_4$

Running tree parsing on our IR tree



MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

Action: done

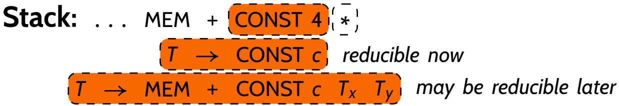
Stack:

Assembly code:

```
ADDI    t0 ← r0 + #4
MUL     t1 ← t0 * ti
ADDI    t2 ← r0 + #4
MUL     t3 ← t2 * ti
LOAD    t4 ← M[t3 + #&a]
STORE   M[t1 + #&b] ← t4
```

Limitation: Tree parsing could fail

- **Syntactic blocking:**
  - Always shifting in shift-reduce conflicts is a *guess* that may prove wrong
- **Stack at conflict:**

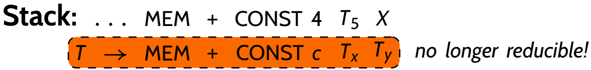


Stack: ... MEM + CONST 4 *

$T \rightarrow$ CONST c *reducible now*

$T \rightarrow$ MEM + CONST c T_x T_y *may be reducible later*

- **Stack some time after shifting:**



Stack: ... MEM + CONST 4 T_5 X

$T \rightarrow$ MEM + CONST c T_x T_y *no longer reducible!*

- **Solution:**
  - Add auxiliary productions that fix the stack (in other words, “undo” erroneous guesses)

Quality of emitted assembly code

		Costs:
ADDI	$t_0 \leftarrow r_0 + \#4$	1
MUL	$t_1 \leftarrow t_0 * t_i$	2
ADDI	$t_2 \leftarrow r_0 + \#4$	1
MUL	$t_3 \leftarrow t_2 * t_i$	2
LOAD	$t_4 \leftarrow M[t_3 + \#\&a]$	10
STORE	$M[t_1 + \#\&b] \leftarrow t_4$	10

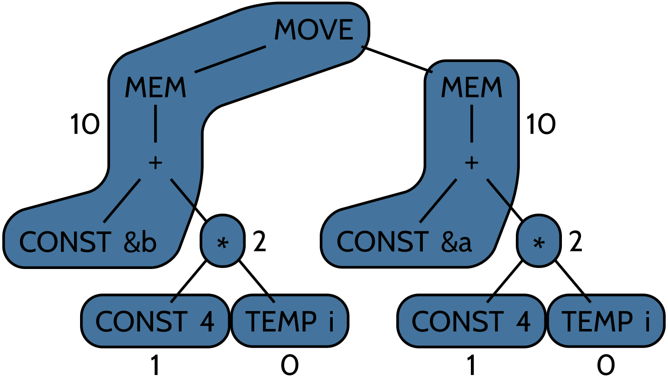
$$\sum \text{cost} = 26$$

Can we do better?

		Costs:
ADDI	$t_0 \leftarrow r_0 + \#4$	1
MUL	$t_1 \leftarrow t_0 * t_i$	2
ADDI	$t_2 \leftarrow r_0 + \#4$	1
MUL	$t_3 \leftarrow t_2 * t_i$	2
LOAD	$t_4 \leftarrow M[t_3 + \#\&a]$	10
STORE	$M[t_1 + \#\&b] \leftarrow t_4$	10

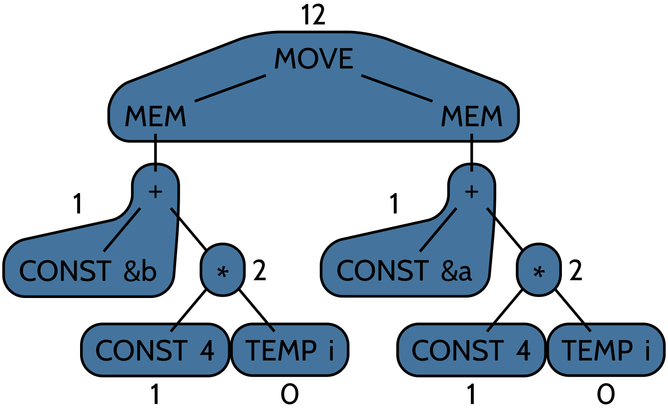
$$\sum \text{cost} = 26$$

Optimal tiling found with tree parsing



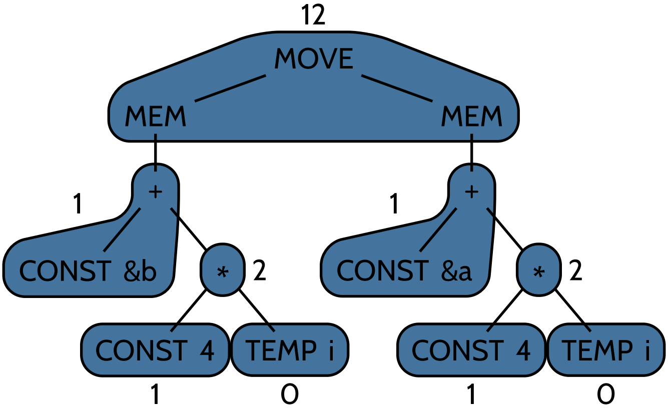
$\sum$ cost = 26

Optimum tiling



$\sum$ cost = 20

Need non-greedy approaches to find this tiling



$\sum$ cost = 20

4th approach:

Dynamic Programming

Fundamental idea

- Derive tree grammar from tile set
- To find optimum tiling:
  1. Find all tiles that match IR tree
  2. Traverse IR tree bottom up:
     - Record least cost of reducing current IR operation to a particular nonterminal
  3. Traverse IR tree top down:
     - Select production that produces nonterminal at least cost
     - Repeat for all subtrees
- To emit assembly code:
  - Traverse IR tree bottom up
  - For each tile *t* in tiling:
    - Emit instruction corresponding to *t*

Finding tiles that match

- Perform **bottom-up tree labeling**
  - See paper by Hoffmann & O'Donnell "Pattern Matching in Trees" (1982) `http://dx.doi.org/10.1145/322290.322295`
- Can be done in linear time $\mathcal{O}(n)$

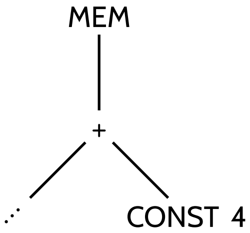
Cost of reduction

- Cost of reducing nonterminal using production *P*:

$$c_P + \sum_{1}^{n} c_i$$

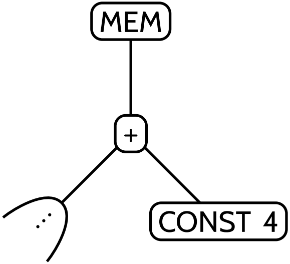
c_P = cost of *P* itself
c_i = cost of *i*th nonterminal appearing in RHS of *P*

Computing costs on example



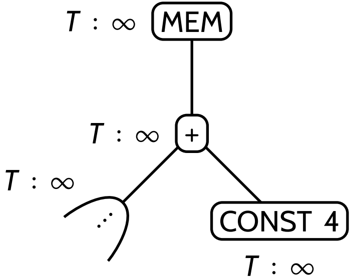
Action:

Computing costs on example



Action: added boxes for better readability

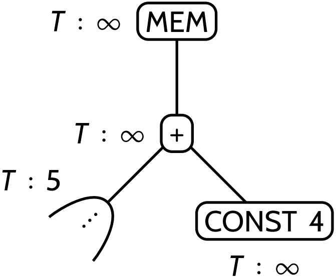
Computing costs on example



Action: initialize all costs to ∞

- $T : c$ denotes “reducible to nonterminal T at cost c”

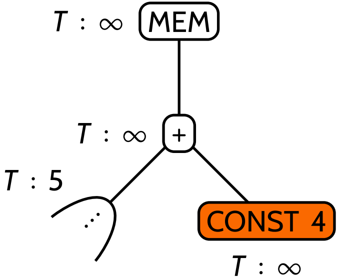
Computing costs on example



Action: assume $T : 5$ already found for subtree

- $T : c$ denotes "reducible to nonterminal T at cost c"

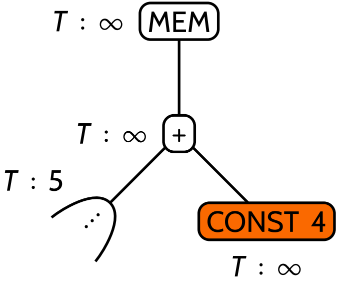
Computing costs on example



Action: start at CONST 4 node

- T : c denotes “reducible to nonterminal T at cost c”

Computing costs on example

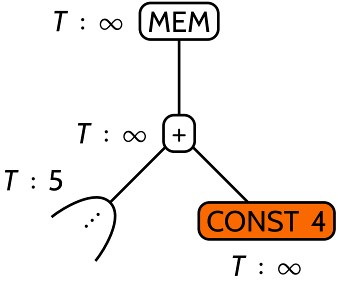


Productions:	**Costs:**
$(T \rightarrow \text{CONST } c)$	1

Action: get productions of tiles that match

- $T : c$ denotes “reducible to nonterminal T at cost c”

Computing costs on example

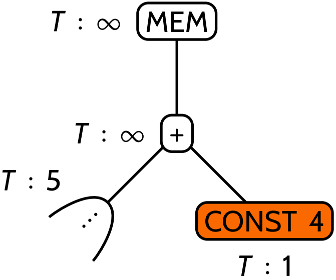


Productions:	**Costs:**
$(T \rightarrow \text{CONST } c)$	$1 + 0$

Action: compute costs of reduction

- $T : c$ denotes "reducible to nonterminal T at cost c"

Computing costs on example

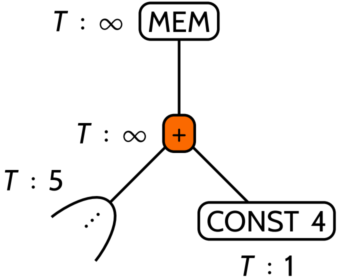


Productions:	**Costs:**
(T → CONST c)	1

Action: select production with least cost

- T : c denotes “reducible to nonterminal T at cost c”

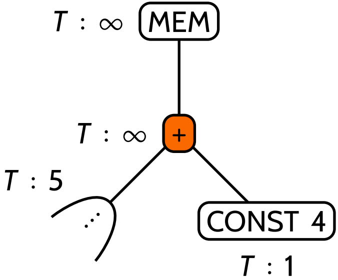
Computing costs on example



Action: continue to + node

- $T : c$ denotes "reducible to nonterminal T at cost c"

Computing costs on example

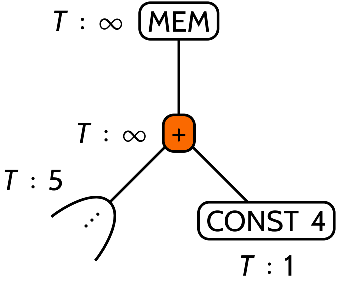


Productions:	**Costs:**
$(T \rightarrow + \; T_y \; T_z)$	1
$(T \rightarrow + \; T_y$ CONST $c)$	1

Action: get productions of tiles that match

- $T : c$ denotes “reducible to nonterminal T at cost c”

Computing costs on example

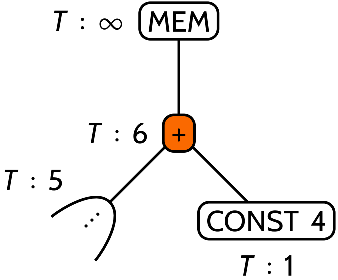


Productions:	**Costs:**
$(T \rightarrow + \; T_y \; T_z)$	$1 + 5 + 1$
$(T \rightarrow + \; T_y$ CONST $c)$	$1 + 5$

Action: compute costs of reduction

- $T : c$ denotes "reducible to nonterminal T at cost c"

Computing costs on example

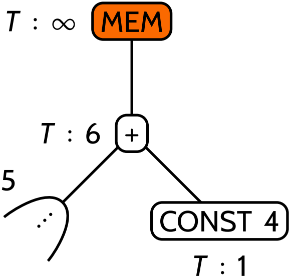


Productions:	**Costs:**
$(T \rightarrow + \ T_y \ T_z)$	7
$(T \rightarrow + \ T_y$ CONST $c)$	6

Action: select production with least cost

- $T : c$ denotes “reducible to nonterminal T at cost c”

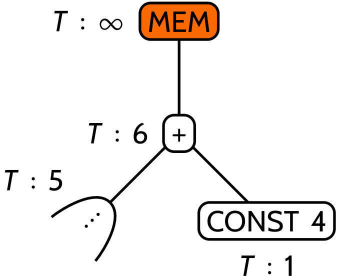
Computing costs on example



Action: continue to *MEM* node

- $T : c$ denotes “reducible to nonterminal T at cost c”

Computing costs on example

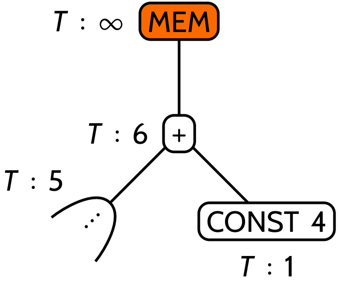


Productions:	**Costs:**
(T → MEM T_y)	10
(T → MEM + T_y CONST c)	10

Action: get productions of tiles that match

- $T : c$ denotes "reducible to nonterminal T at cost c"

Computing costs on example

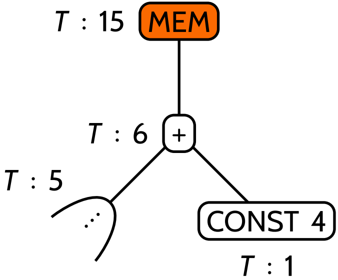


Productions:	**Costs:**
$(T \rightarrow \text{MEM } T_y)$	$10 + 6$
$(T \rightarrow \text{MEM} + T_y \text{ CONST } c)$	$10 + 5$

Action: compute costs of reduction

- $T : c$ denotes “reducible to nonterminal T at cost c”

Computing costs on example

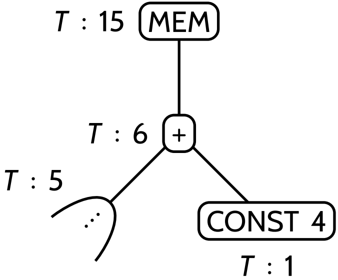


Productions:	**Costs:**
$(T \rightarrow$ MEM $T_y)$	16
$(T \rightarrow$ MEM + T_y CONST $c)$	15

Action: select production with least cost

- $T : c$ denotes "reducible to nonterminal T at cost c"

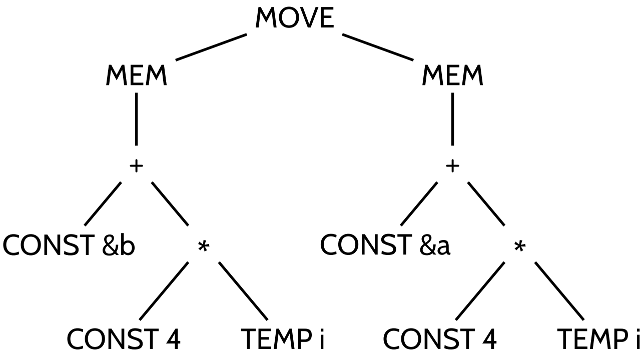
Computing costs on example



Action: done

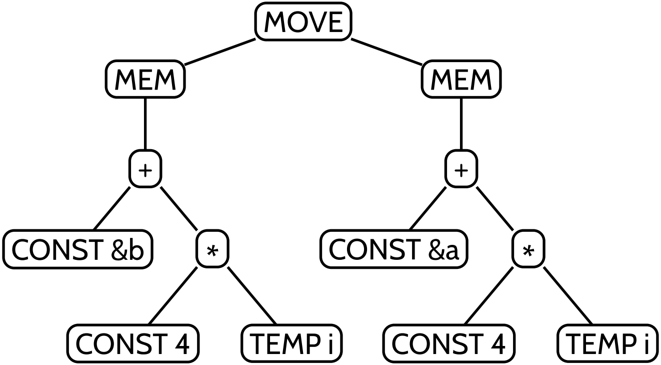
- $T : c$ denotes “reducible to nonterminal T at cost c”

Running dynamic programming on our IR tree



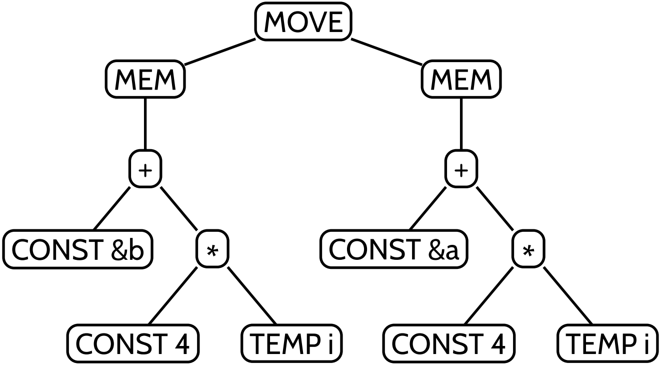
Action:

Running dynamic programming on our IR tree



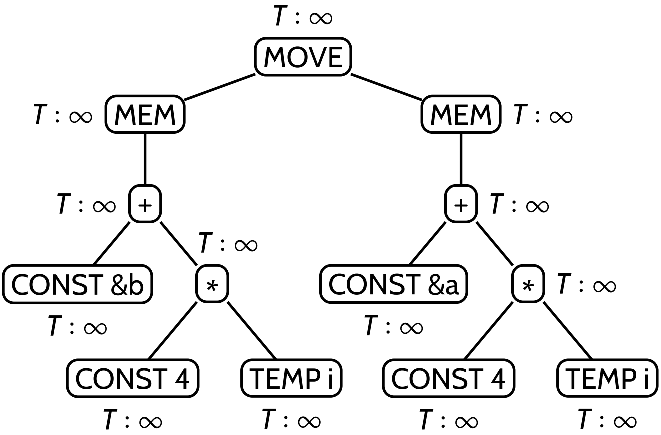
Action: added boxes for better readability

Running dynamic programming on our IR tree



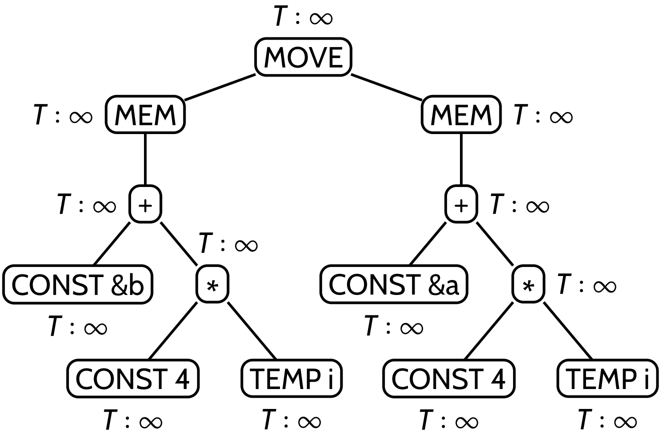
Action: assume tile matches already found

Running dynamic programming on our IR tree



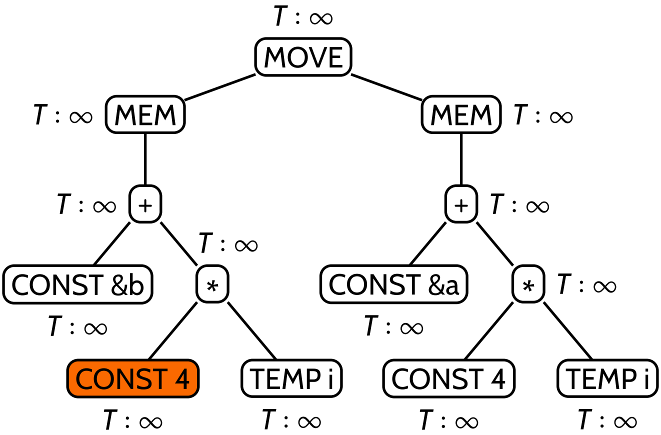
Action: initialize reduction costs

Running dynamic programming on our IR tree



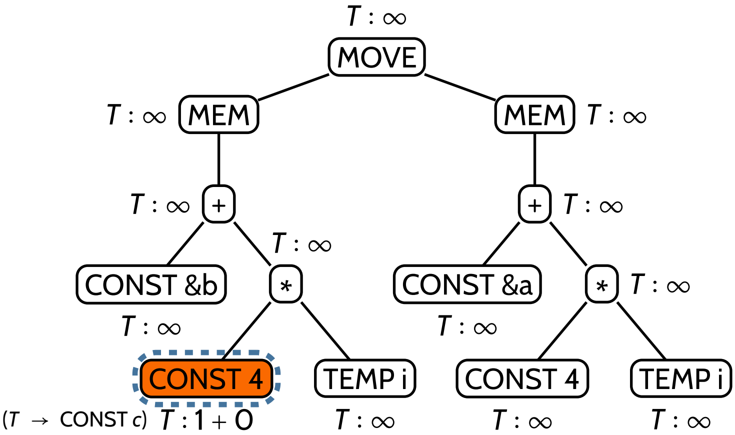
Action: compute least reduction costs

Running dynamic programming on our IR tree



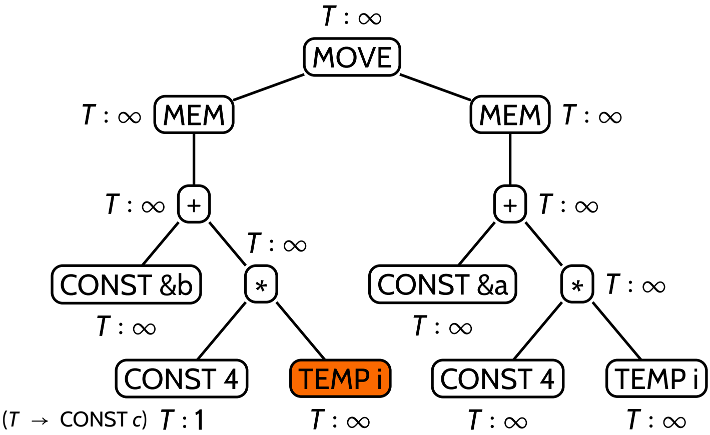
Action: compute least reduction costs

Running dynamic programming on our IR tree



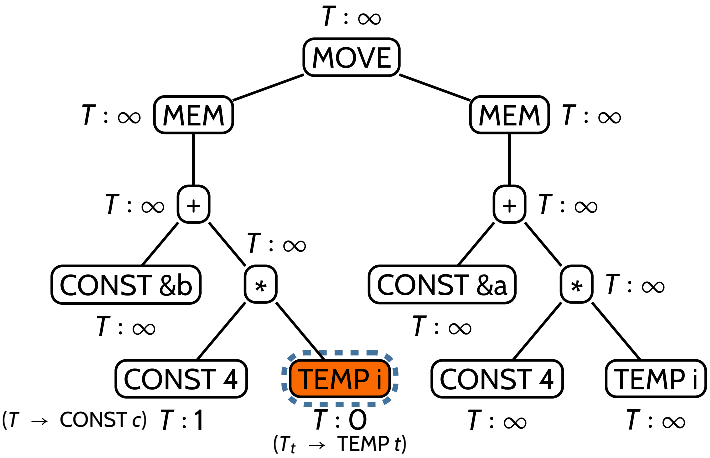
Action: compute least reduction costs

Running dynamic programming on our IR tree



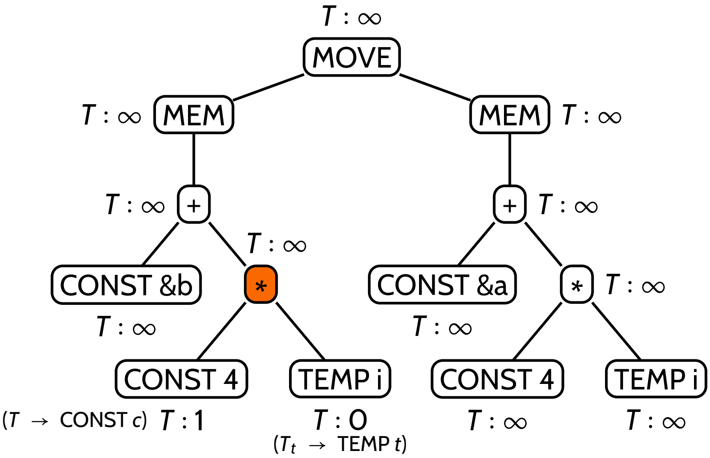
Action: compute least reduction costs

Running dynamic programming on our IR tree



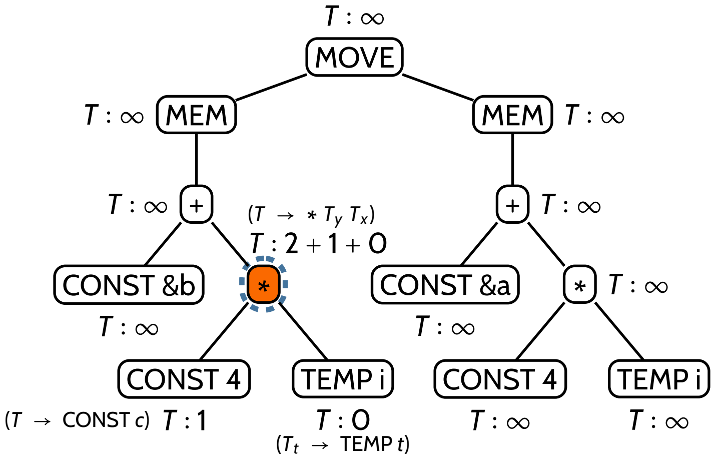
Action: compute least reduction costs

Running dynamic programming on our IR tree



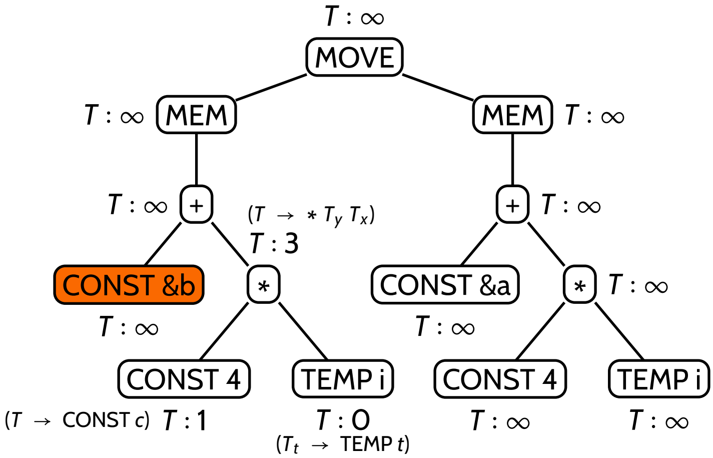
Action: compute least reduction costs

Running dynamic programming on our IR tree



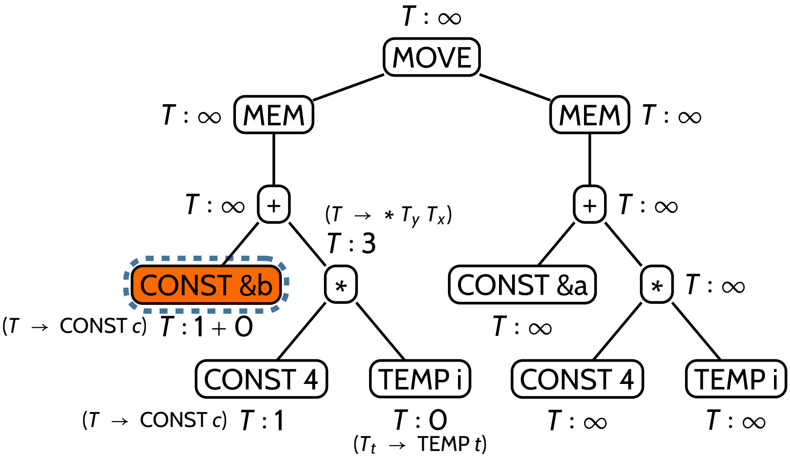
Action: compute least reduction costs

Running dynamic programming on our IR tree



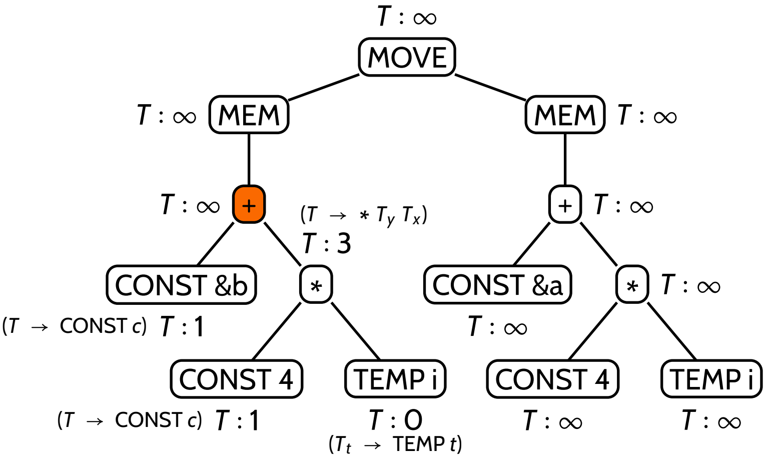
Action: compute least reduction costs

Running dynamic programming on our IR tree



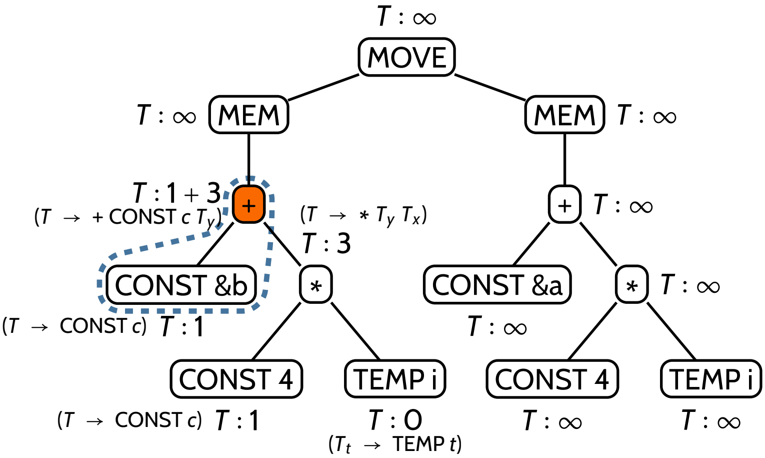
Action: compute least reduction costs

Running dynamic programming on our IR tree



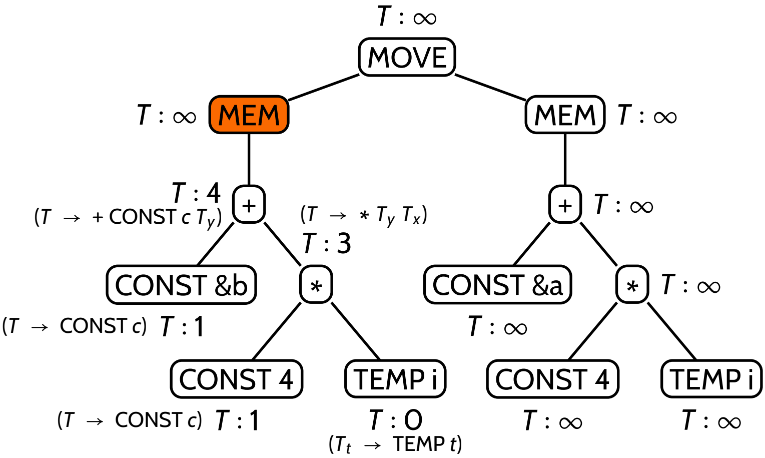
Action: compute least reduction costs

Running dynamic programming on our IR tree


$T : \infty$
MOVE
$T : \infty$
MEM
MEM
$T : \infty$
$T : 1 + 3$
$(T \rightarrow + \text{CONST } c \ T_y)$
+
$(T \rightarrow * \ T_y \ T_x)$
$T : 3$
+
$T : \infty$
CONST &b
*
CONST &a
*
$T : \infty$
$(T \rightarrow \text{CONST } c)$ $T : 1$
$T : \infty$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow \text{CONST } c)$ $T : 1$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$
$T : \infty$
$T : \infty$

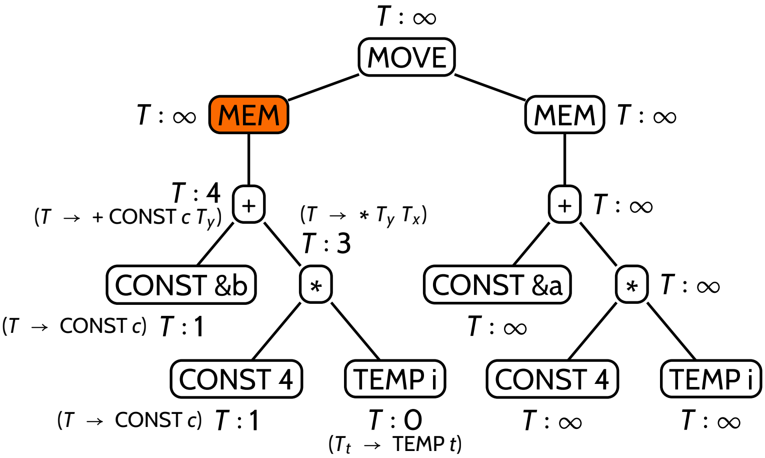
Action: compute least reduction costs

Running dynamic programming on our IR tree



Action: compute least reduction costs

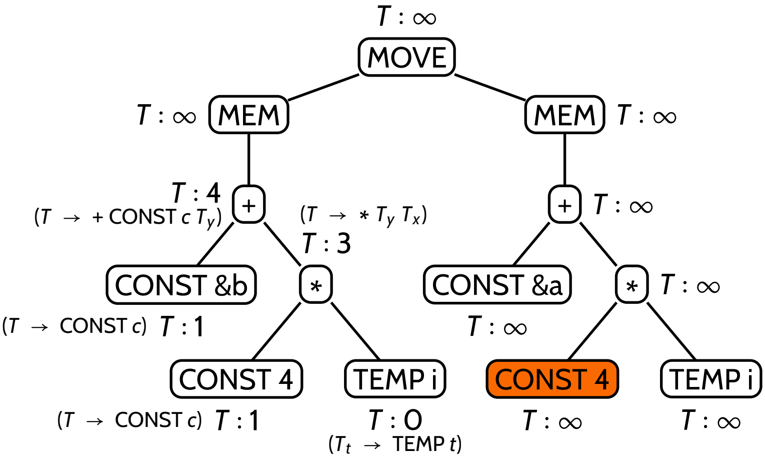
Running dynamic programming on our IR tree



$T : \infty$
MOVE
$T : \infty$
MEM
MEM
$T : \infty$
$T : 4$
$(T \rightarrow + \text{CONST}\ c\ T_y)$
+
$(T \rightarrow * T_y\ T_x)$
$T : 3$
+
$T : \infty$
CONST &b
*
CONST &a
*
$T : \infty$
$(T \rightarrow \text{CONST}\ c)$ $T : 1$
$T : \infty$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow \text{CONST}\ c)$ $T : 1$
$T : 0$
$(T_t \rightarrow \text{TEMP}\ t)$
$T : \infty$
$T : \infty$

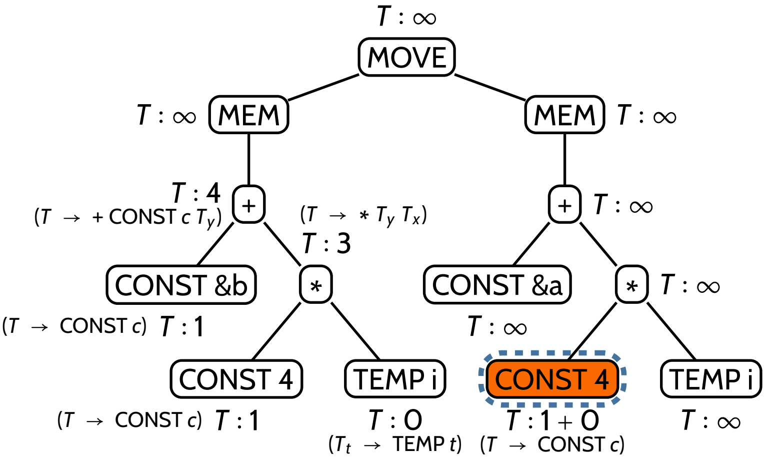
Action: LOAD instructions not allowed here (l-value)

Running dynamic programming on our IR tree


$T : \infty$
MOVE
$T : \infty$
MEM
MEM
$T : \infty$
$T : 4$
$(T \rightarrow + \text{CONST } c\ T_y)$
+
$(T \rightarrow * \ T_y\ T_x)$
$T : 3$
+
$T : \infty$
CONST &b
*
CONST &a
*
$T : \infty$
$(T \rightarrow \text{CONST } c)$ $T : 1$
$T : \infty$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow \text{CONST } c)$ $T : 1$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$
$T : \infty$
$T : \infty$

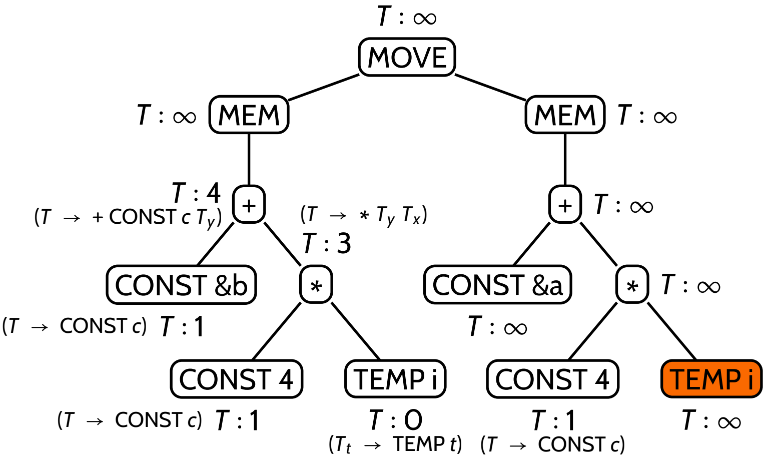
Action: compute least reduction costs

Running dynamic programming on our IR tree


$T : \infty$
MOVE
$T : \infty$
MEM
MEM
$T : \infty$
$T : 4$
$(T \rightarrow + \text{CONST}\ c\ T_y)$
+
$(T \rightarrow * T_y\ T_x)$
$T : 3$
+
$T : \infty$
CONST &b
*
CONST &a
*
$T : \infty$
$(T \rightarrow \text{CONST}\ c)$ $T : 1$
$T : \infty$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow \text{CONST}\ c)$ $T : 1$
$T : 0$
$(T_t \rightarrow \text{TEMP}\ t)$
$T : 1 + 0$
$(T \rightarrow \text{CONST}\ c)$
$T : \infty$

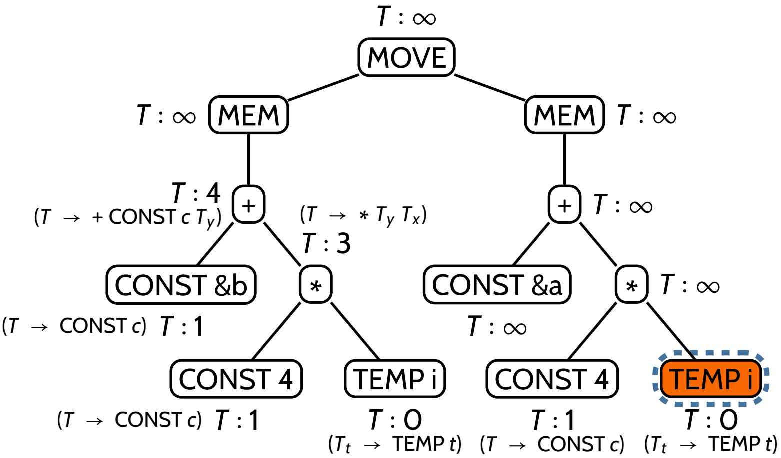
Action: compute least reduction costs

Running dynamic programming on our IR tree



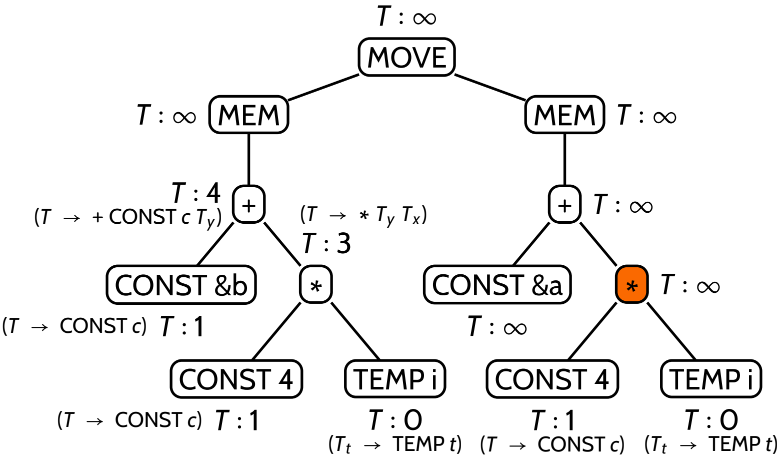
Action: compute least reduction costs

Running dynamic programming on our IR tree


$T : \infty$
MOVE
$T : \infty$
MEM
MEM
$T : \infty$
$T : 4$
$(T \rightarrow + \text{CONST}\ c\ T_y)$
+
$(T \rightarrow * \ T_y\ T_x)$
$T : 3$
+
$T : \infty$
CONST &b
*
CONST &a
*
$T : \infty$
$(T \rightarrow \text{CONST}\ c)$ $T : 1$
$T : \infty$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow \text{CONST}\ c)$ $T : 1$
$T : 0$
$(T_t \rightarrow \text{TEMP}\ t)$
$T : 1$
$(T \rightarrow \text{CONST}\ c)$
$T : 0$
$(T_t \rightarrow \text{TEMP}\ t)$

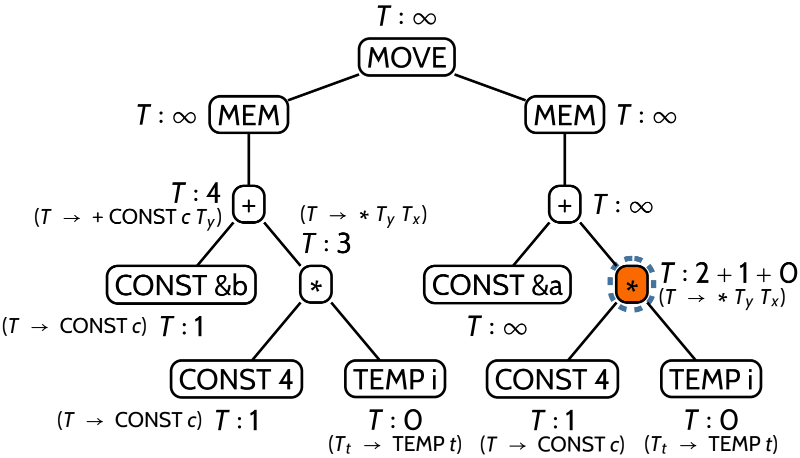
Action: compute least reduction costs

Running dynamic programming on our IR tree


$T : \infty$
MOVE
$T : \infty$
MEM
MEM
$T : \infty$
$T : 4$
$(T \rightarrow + \text{CONST}\ c\ T_y)$
+
$(T \rightarrow * T_y\ T_x)$
$T : 3$
+
$T : \infty$
CONST &b
*
CONST &a
*
$T : \infty$
$(T \rightarrow \text{CONST}\ c)$ $T : 1$
$T : \infty$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow \text{CONST}\ c)$ $T : 1$
$T : 0$
$(T_t \rightarrow \text{TEMP}\ t)$
$T : 1$
$(T \rightarrow \text{CONST}\ c)$
$T : 0$
$(T_t \rightarrow \text{TEMP}\ t)$

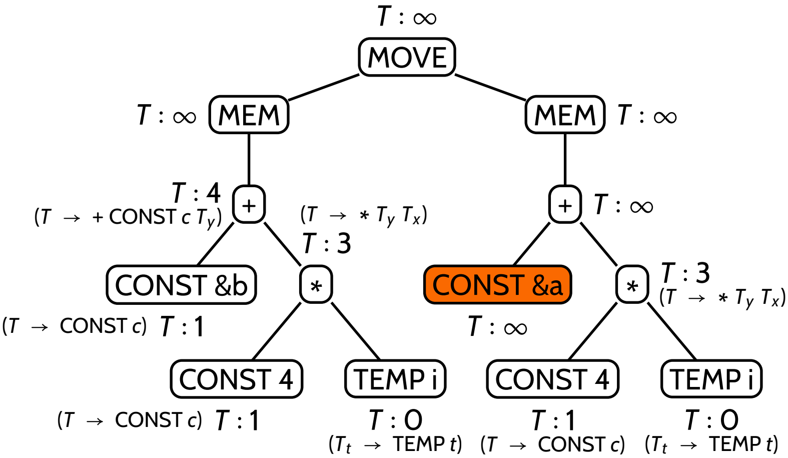
Action: compute least reduction costs

Running dynamic programming on our IR tree


$T : \infty$
MOVE
$T : \infty$
MEM
MEM
$T : \infty$
$T : 4$
$(T \rightarrow + \text{CONST } c\ T_y)$
+
$(T \rightarrow * T_y\ T_x)$
$T : 3$
+
$T : \infty$
CONST &b
*
CONST &a
*
$T : 2 + 1 + 0$
$(T \rightarrow * T_y\ T_x)$
$(T \rightarrow \text{CONST } c)$ $T : 1$
$T : \infty$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow \text{CONST } c)$ $T : 1$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$
$T : 1$
$(T \rightarrow \text{CONST } c)$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$

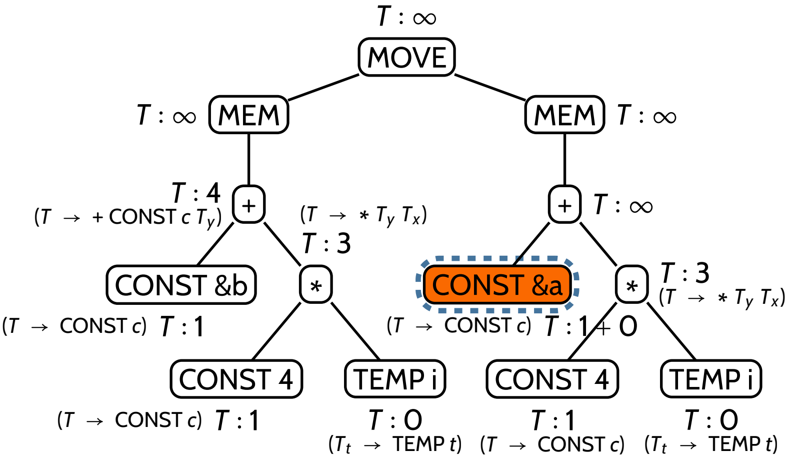
Action: compute least reduction costs

Running dynamic programming on our IR tree



Action: compute least reduction costs

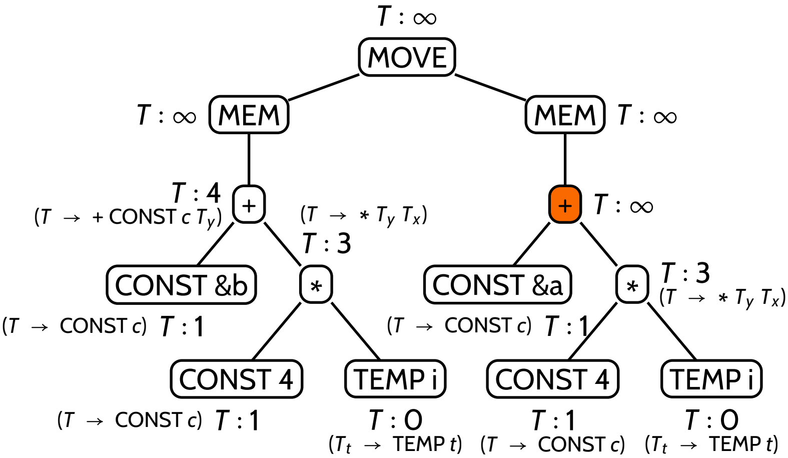
Running dynamic programming on our IR tree



$T : \infty$
MOVE
$T : \infty$
MEM
MEM
$T : \infty$
$T : 4$
$(T \rightarrow + \text{CONST}\ c\ T_y)$
+
$(T \rightarrow * T_y\ T_x)$
$T : 3$
+
$T : \infty$
CONST &b
$(T \rightarrow \text{CONST}\ c)$ $T : 1$
*
CONST &a
$(T \rightarrow \text{CONST}\ c)$ $T : 1 + 0$
*
$T : 3$
$(T \rightarrow * T_y\ T_x)$
CONST 4
$(T \rightarrow \text{CONST}\ c)$ $T : 1$
TEMP i
$T : 0$
$(T_t \rightarrow \text{TEMP}\ t)$
CONST 4
$T : 1$
$(T \rightarrow \text{CONST}\ c)$
TEMP i
$T : 0$
$(T_t \rightarrow \text{TEMP}\ t)$

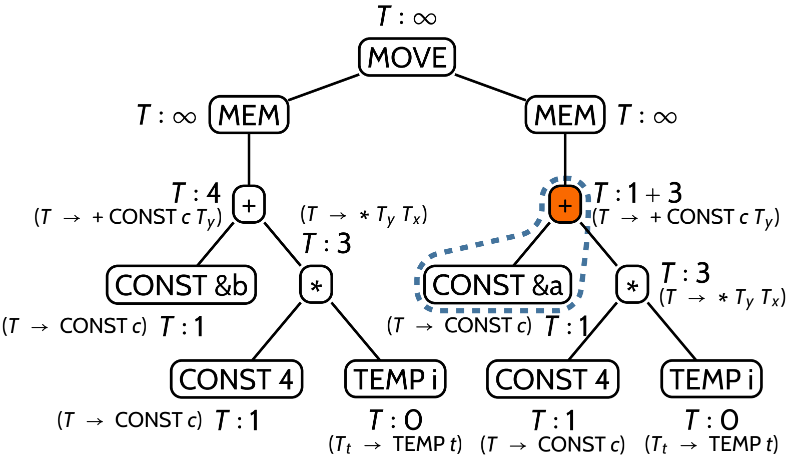
Action: compute least reduction costs

Running dynamic programming on our IR tree


$T : \infty$
MOVE
$T : \infty$
MEM
MEM
$T : \infty$
$T : 4$
$(T \rightarrow + \text{CONST}\ c\ T_y)$
+
$(T \rightarrow * T_y\ T_x)$
$T : 3$
+
$T : \infty$
CONST &b
*
CONST &a
*
$T : 3$
$(T \rightarrow * T_y\ T_x)$
$(T \rightarrow \text{CONST}\ c)$ $T : 1$
$(T \rightarrow \text{CONST}\ c)$ $T : 1$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow \text{CONST}\ c)$ $T : 1$
$T : 0$
$(T_t \rightarrow \text{TEMP}\ t)$
$T : 1$
$(T \rightarrow \text{CONST}\ c)$
$T : 0$
$(T_t \rightarrow \text{TEMP}\ t)$

Action: compute least reduction costs

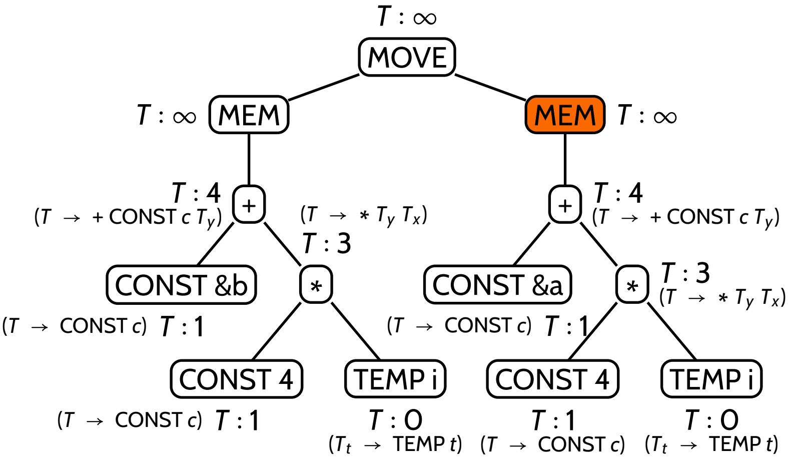
Running dynamic programming on our IR tree



$T:\infty$
MOVE
$T:\infty$
MEM
MEM
$T:\infty$
$T:4$
+
$(T \rightarrow +\,\text{CONST}\ c\ T_y)$
$(T \rightarrow *\ T_y\ T_x)$
$T:3$
+
$T:1+3$
$(T \rightarrow +\,\text{CONST}\ c\ T_y)$
CONST &b
*
CONST &a
*
$T:3$
$(T \rightarrow *\ T_y\ T_x)$
$(T \rightarrow \text{CONST}\ c)$ $T:1$
$(T \rightarrow \text{CONST}\ c)$ $T:1$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow \text{CONST}\ c)$ $T:1$
$T:0$
$(T_t \rightarrow \text{TEMP}\ t)$
$T:1$
$(T \rightarrow \text{CONST}\ c)$
$T:0$
$(T_t \rightarrow \text{TEMP}\ t)$

Action: compute least reduction costs

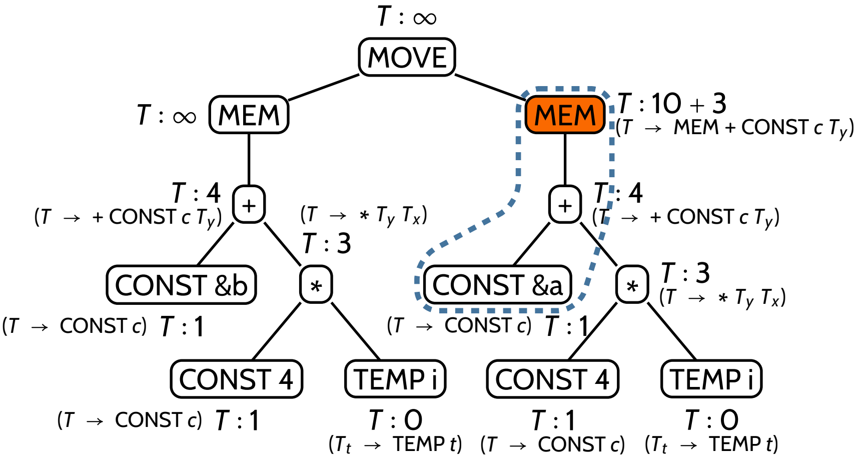
Running dynamic programming on our IR tree



$T : \infty$
MOVE
$T : \infty$
MEM
MEM
$T : \infty$
$T : 4$
$(T \rightarrow + \text{CONST } c\ T_y)$
+
$(T \rightarrow * T_y\ T_x)$
$T : 3$
+
$T : 4$
$(T \rightarrow + \text{CONST } c\ T_y)$
CONST &b
*
CONST &a
*
$T : 3$
$(T \rightarrow * T_y\ T_x)$
$(T \rightarrow \text{CONST } c)$ $T : 1$
$(T \rightarrow \text{CONST } c)$ $T : 1$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow \text{CONST } c)$ $T : 1$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$
$T : 1$
$(T \rightarrow \text{CONST } c)$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$

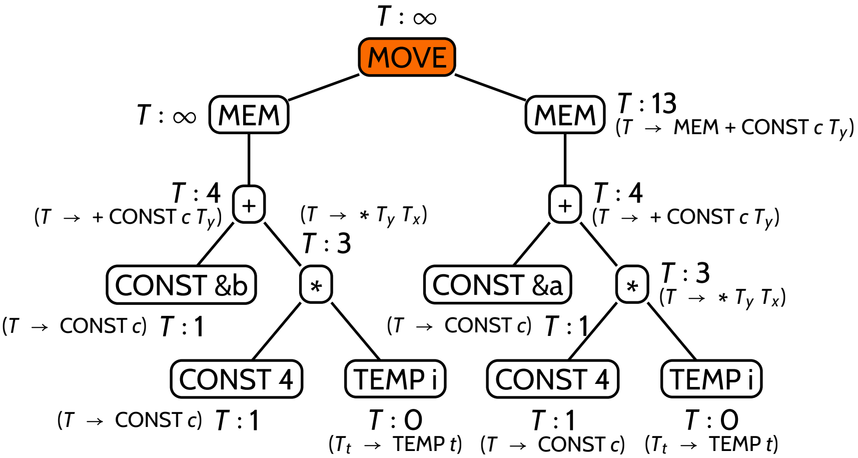
Action: compute least reduction costs

Running dynamic programming on our IR tree


$T : \infty$
MOVE
$T : \infty$
MEM
MEM
$T : 10 + 3$
$(T \rightarrow$ MEM + CONST $c\ T_y)$
$T : 4$
$(T \rightarrow$ + CONST $c\ T_y)$
+
$(T \rightarrow *\ T_y\ T_x)$
$T : 3$
+
$T : 4$
$(T \rightarrow$ + CONST $c\ T_y)$
CONST &b
*
CONST &a
*
$T : 3$
$(T \rightarrow *\ T_y\ T_x)$
$(T \rightarrow$ CONST $c)$ $T : 1$
$(T \rightarrow$ CONST $c)$ $T : 1$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow$ CONST $c)$ $T : 1$
$T : 0$
$(T_t \rightarrow$ TEMP $t)$
$T : 1$
$(T \rightarrow$ CONST $c)$
$T : 0$
$(T_t \rightarrow$ TEMP $t)$

Action: compute least reduction costs

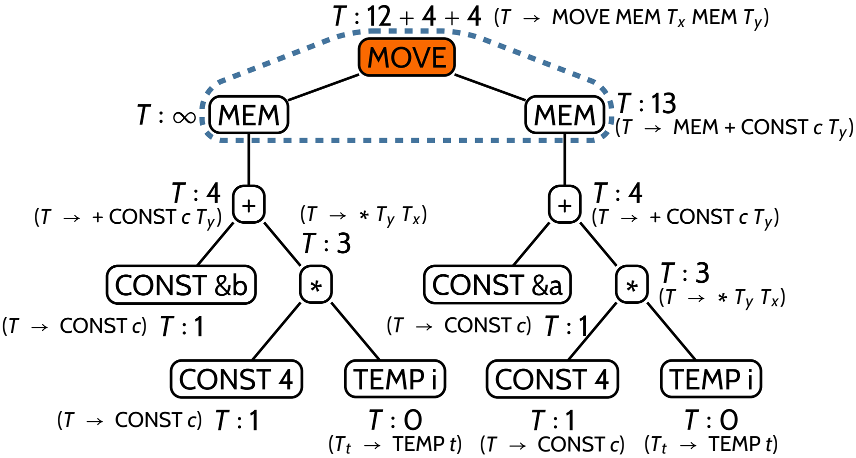
Running dynamic programming on our IR tree



$T : \infty$
MOVE
$T : \infty$
MEM
MEM
$T : 13$
$(T \rightarrow \text{MEM} + \text{CONST } c\ T_y)$
$T : 4$
$(T \rightarrow + \text{CONST } c\ T_y)$
+
$(T \rightarrow * \ T_y\ T_x)$
$T : 3$
+
$T : 4$
$(T \rightarrow + \text{CONST } c\ T_y)$
CONST &b
*
CONST &a
*
$T : 3$
$(T \rightarrow * \ T_y\ T_x)$
$(T \rightarrow \text{CONST } c)$ $T : 1$
$(T \rightarrow \text{CONST } c)$ $T : 1$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow \text{CONST } c)$ $T : 1$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$
$T : 1$
$(T \rightarrow \text{CONST } c)$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$

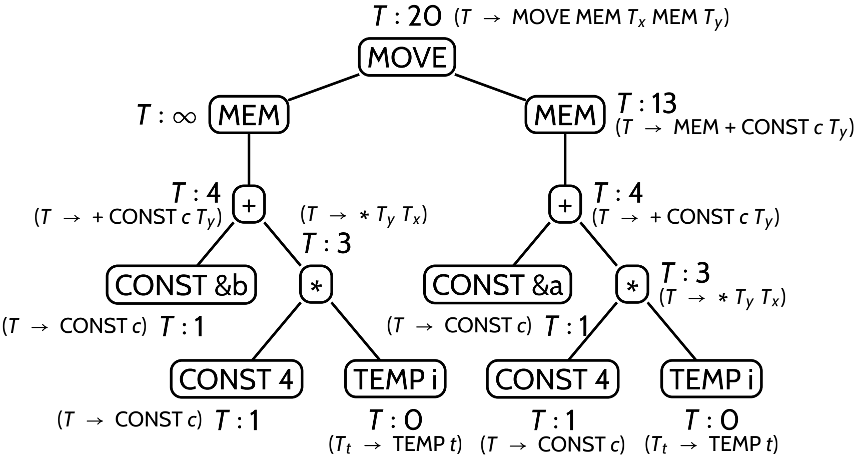
Action: compute least reduction costs

Running dynamic programming on our IR tree


$T : 12 + 4 + 4$ ($T \rightarrow$ MOVE MEM T_x MEM T_y)
MOVE
$T : \infty$
MEM
MEM
$T : 13$
($T \rightarrow$ MEM + CONST c T_y)
$T : 4$
($T \rightarrow$ + CONST c T_y)
+
($T \rightarrow * T_y T_x$)
$T : 3$
+
$T : 4$
($T \rightarrow$ + CONST c T_y)
CONST &b
*
CONST &a
*
$T : 3$
($T \rightarrow * T_y T_x$)
($T \rightarrow$ CONST c) $T : 1$
($T \rightarrow$ CONST c) $T : 1$
CONST 4
TEMP i
CONST 4
TEMP i
($T \rightarrow$ CONST c) $T : 1$
$T : 0$
($T_t \rightarrow$ TEMP t)
$T : 1$
($T \rightarrow$ CONST c)
$T : 0$
($T_t \rightarrow$ TEMP t)

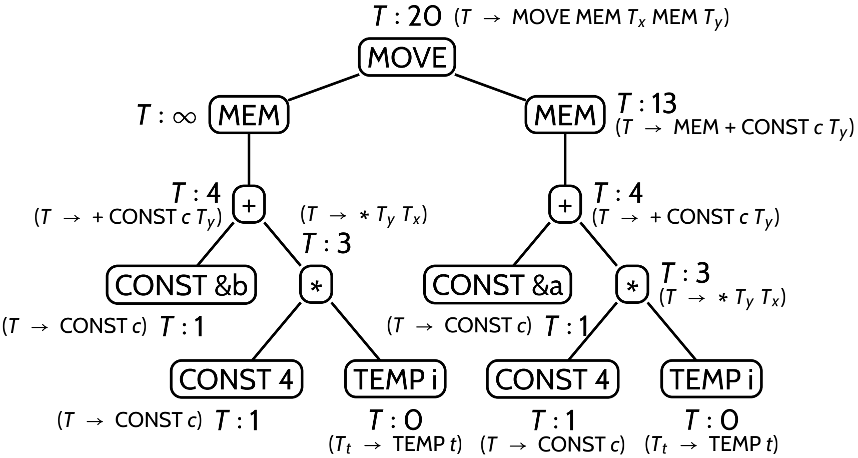
Action: compute least reduction costs

Running dynamic programming on our IR tree


$T : 20$ $(T \rightarrow \text{MOVE MEM } T_x \text{ MEM } T_y)$
MOVE
$T : \infty$
MEM
MEM
$T : 13$
$(T \rightarrow \text{MEM + CONST } c\ T_y)$
$T : 4$
$(T \rightarrow \text{+ CONST } c\ T_y)$
+
$(T \rightarrow *\ T_y\ T_x)$
$T : 3$
+
$T : 4$
$(T \rightarrow \text{+ CONST } c\ T_y)$
CONST &b
*
CONST &a
*
$T : 3$
$(T \rightarrow *\ T_y\ T_x)$
$(T \rightarrow \text{CONST } c)$ $T : 1$
$(T \rightarrow \text{CONST } c)$ $T : 1$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow \text{CONST } c)$ $T : 1$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$
$T : 1$
$(T \rightarrow \text{CONST } c)$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$

Action: done computing costs

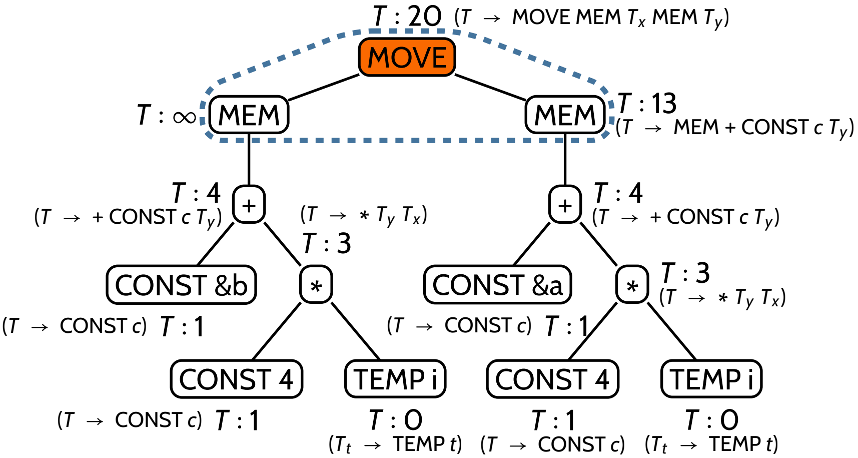
Running dynamic programming on our IR tree



$T: 20$ ($T \rightarrow$ MOVE MEM T_x MEM T_y)
MOVE
$T: \infty$
MEM
MEM
$T: 13$
($T \rightarrow$ MEM + CONST c T_y)
$T: 4$
($T \rightarrow$ + CONST c T_y)
+
($T \rightarrow * T_y T_x$)
$T: 3$
+
$T: 4$
($T \rightarrow$ + CONST c T_y)
CONST &b
*
CONST &a
*
$T: 3$
($T \rightarrow * T_y T_x$)
($T \rightarrow$ CONST c) $T: 1$
($T \rightarrow$ CONST c) $T: 1$
CONST 4
TEMP i
CONST 4
TEMP i
($T \rightarrow$ CONST c) $T: 1$
$T: 0$
($T_t \rightarrow$ TEMP t)
$T: 1$
($T \rightarrow$ CONST c)
$T: 0$
($T_t \rightarrow$ TEMP t)

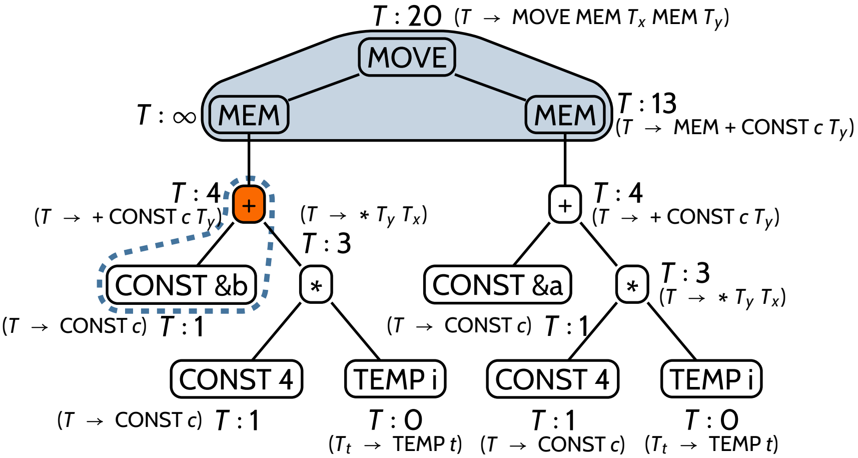
Action: select productions

Running dynamic programming on our IR tree


$T : 20$ $(T \rightarrow$ MOVE MEM T_x MEM $T_y)$
MOVE
$T : \infty$
MEM
MEM
$T : 13$
$(T \rightarrow$ MEM + CONST c $T_y)$
$T : 4$
$(T \rightarrow$ + CONST c $T_y)$
+
$(T \rightarrow * T_y T_x)$
$T : 3$
+
$T : 4$
$(T \rightarrow$ + CONST c $T_y)$
CONST &b
*
CONST &a
*
$T : 3$
$(T \rightarrow * T_y T_x)$
$(T \rightarrow$ CONST $c)$ $T : 1$
$(T \rightarrow$ CONST $c)$ $T : 1$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow$ CONST $c)$ $T : 1$
$T : 0$
$(T_t \rightarrow$ TEMP $t)$
$T : 1$
$(T \rightarrow$ CONST $c)$
$T : 0$
$(T_t \rightarrow$ TEMP $t)$

Action: select productions

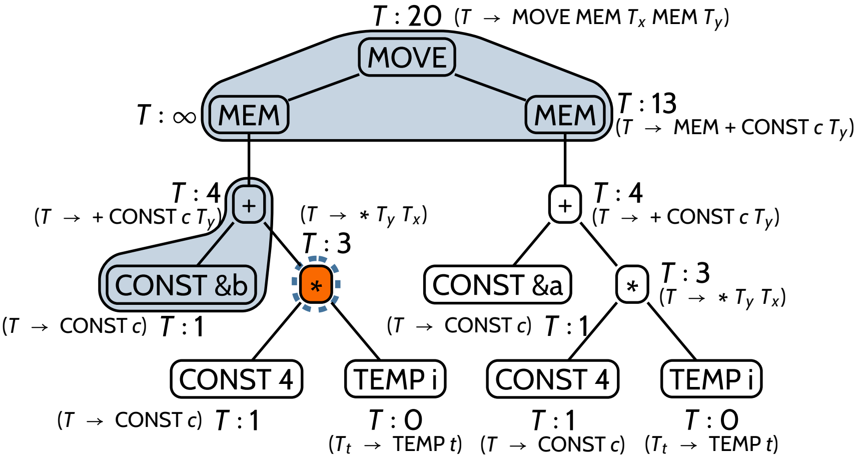
Running dynamic programming on our IR tree



$T:20$ ($T \rightarrow$ MOVE MEM T_x MEM T_y)
MOVE
$T:\infty$
MEM
MEM
$T:13$
($T \rightarrow$ MEM + CONST c T_y)
$T:4$
($T \rightarrow$ + CONST c T_y)
+
($T \rightarrow * T_y T_x$)
$T:3$
+
$T:4$
($T \rightarrow$ + CONST c T_y)
CONST &b
*
CONST &a
*
$T:3$
($T \rightarrow * T_y T_x$)
($T \rightarrow$ CONST c) $T:1$
($T \rightarrow$ CONST c) $T:1$
CONST 4
TEMP i
CONST 4
TEMP i
($T \rightarrow$ CONST c) $T:1$
$T:0$
($T_t \rightarrow$ TEMP t)
$T:1$
($T \rightarrow$ CONST c)
$T:0$
($T_t \rightarrow$ TEMP t)

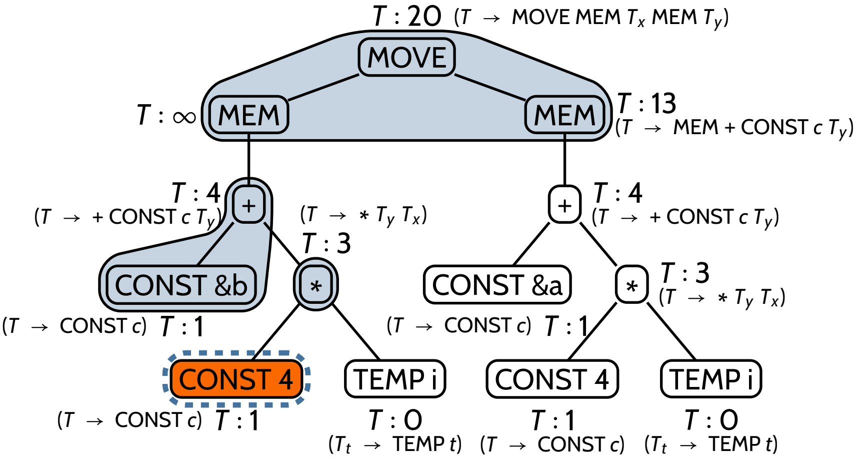
Action: select productions

Running dynamic programming on our IR tree


$T : 20$ $(T \rightarrow \text{MOVE MEM } T_x \text{ MEM } T_y)$
MOVE
$T : \infty$
MEM
MEM
$T : 13$
$(T \rightarrow \text{MEM + CONST } c\ T_y)$
$T : 4$
$(T \rightarrow \text{+ CONST } c\ T_y)$
+
$(T \rightarrow * \ T_y\ T_x)$
$T : 3$
+
$T : 4$
$(T \rightarrow \text{+ CONST } c\ T_y)$
CONST &b
*
CONST &a
*
$T : 3$
$(T \rightarrow * \ T_y\ T_x)$
$(T \rightarrow \text{CONST } c)$ $T : 1$
$(T \rightarrow \text{CONST } c)$ $T : 1$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow \text{CONST } c)$ $T : 1$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$
$T : 1$
$(T \rightarrow \text{CONST } c)$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$

Action: select productions

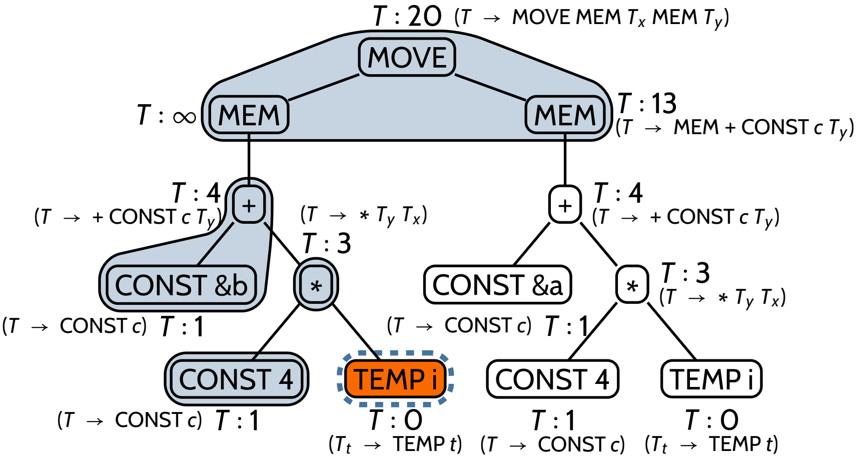
Running dynamic programming on our IR tree



$T:20$ ($T \rightarrow$ MOVE MEM T_x MEM T_y)
MOVE
$T:\infty$
MEM
MEM
$T:13$
($T \rightarrow$ MEM + CONST c T_y)
$T:4$
($T \rightarrow$ + CONST c T_y)
+
($T \rightarrow * \, T_y \, T_x$)
$T:3$
+
$T:4$
($T \rightarrow$ + CONST c T_y)
CONST &b
*
CONST &a
*
$T:3$
($T \rightarrow * \, T_y \, T_x$)
($T \rightarrow$ CONST c) $T:1$
($T \rightarrow$ CONST c) $T:1$
CONST 4
TEMP i
CONST 4
TEMP i
($T \rightarrow$ CONST c) $T:1$
$T:0$
($T_t \rightarrow$ TEMP t)
$T:1$
($T \rightarrow$ CONST c)
$T:0$
($T_t \rightarrow$ TEMP t)

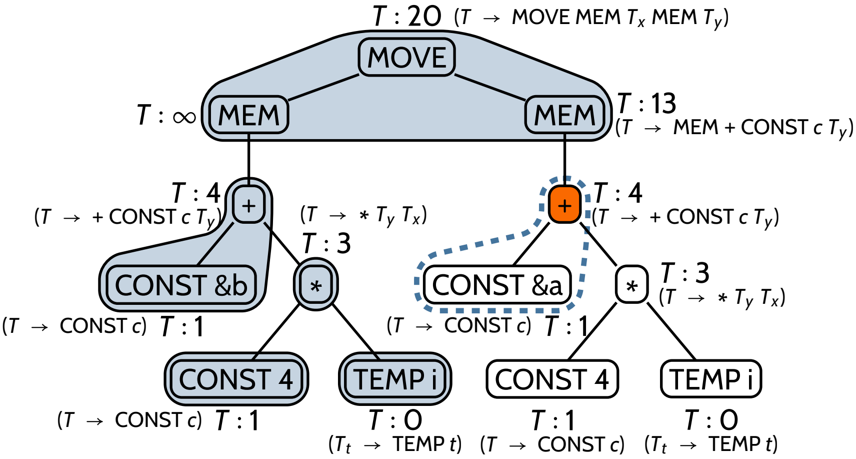
Action: select productions

Running dynamic programming on our IR tree


$T : 20$ $(T \rightarrow$ MOVE MEM T_x MEM $T_y)$
MOVE
$T : \infty$
MEM
MEM
$T : 13$
$(T \rightarrow$ MEM + CONST c $T_y)$
$T : 4$
$(T \rightarrow$ + CONST c $T_y)$
+
$(T \rightarrow * T_y T_x)$
$T : 3$
+
$T : 4$
$(T \rightarrow$ + CONST c $T_y)$
CONST &b
*
CONST &a
*
$T : 3$
$(T \rightarrow * T_y T_x)$
$(T \rightarrow$ CONST $c)$ $T : 1$
$(T \rightarrow$ CONST $c)$ $T : 1$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow$ CONST $c)$ $T : 1$
$T : 0$
$(T_t \rightarrow$ TEMP $t)$
$T : 1$
$(T \rightarrow$ CONST $c)$
$T : 0$
$(T_t \rightarrow$ TEMP $t)$

Action: select productions

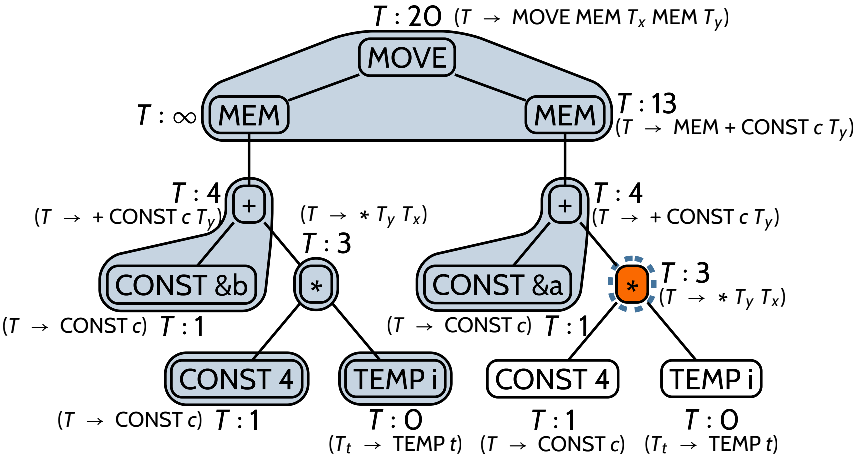
Running dynamic programming on our IR tree



$T : 20$ ($T \rightarrow$ MOVE MEM T_x MEM T_y)
MOVE
$T : \infty$
MEM
MEM
$T : 13$
($T \rightarrow$ MEM + CONST c T_y)
$T : 4$
($T \rightarrow$ + CONST c T_y)
+
($T \rightarrow * T_y T_x$)
$T : 3$
+
$T : 4$
($T \rightarrow$ + CONST c T_y)
CONST &b
*
CONST &a
*
$T : 3$
($T \rightarrow * T_y T_x$)
($T \rightarrow$ CONST c) $T : 1$
($T \rightarrow$ CONST c) $T : 1$
CONST 4
TEMP i
CONST 4
TEMP i
($T \rightarrow$ CONST c) $T : 1$
$T : 0$
($T_t \rightarrow$ TEMP t)
$T : 1$
($T \rightarrow$ CONST c)
$T : 0$
($T_t \rightarrow$ TEMP t)

Action: select productions

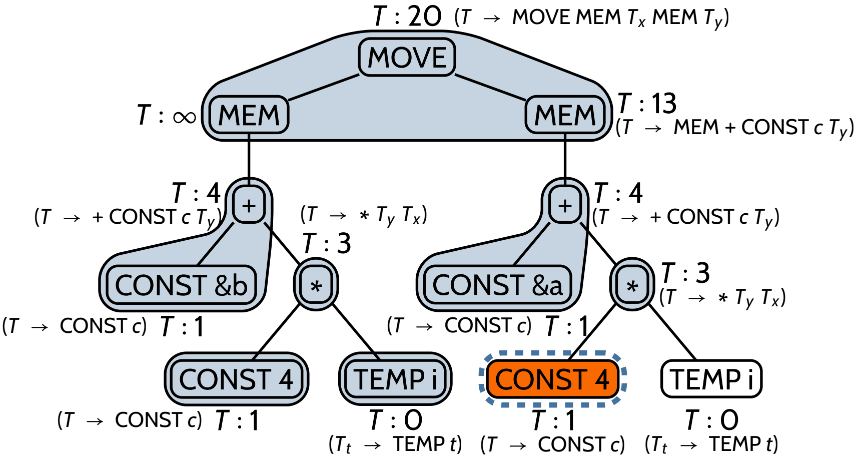
Running dynamic programming on our IR tree



$T : 20$ $(T \rightarrow \text{MOVE MEM } T_x \text{ MEM } T_y)$
MOVE
$T : \infty$
MEM
MEM
$T : 13$
$(T \rightarrow \text{MEM + CONST } c\ T_y)$
$T : 4$
$(T \rightarrow \text{+ CONST } c\ T_y)$
+
$(T \rightarrow *\ T_y\ T_x)$
$T : 3$
*
+
$T : 4$
$(T \rightarrow \text{+ CONST } c\ T_y)$
CONST &b
CONST &a
*
$T : 3$
$(T \rightarrow *\ T_y\ T_x)$
$(T \rightarrow \text{CONST } c)$ $T : 1$
$(T \rightarrow \text{CONST } c)$ $T : 1$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow \text{CONST } c)$ $T : 1$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$
$T : 1$
$(T \rightarrow \text{CONST } c)$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$

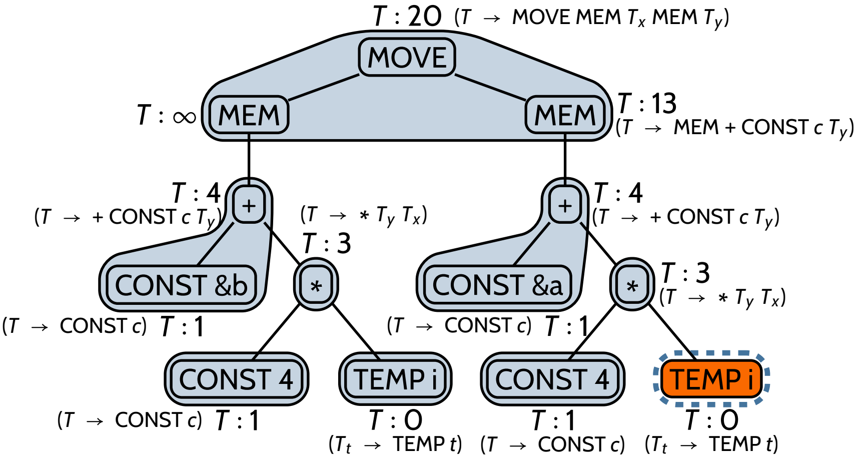
Action: select productions

Running dynamic programming on our IR tree


$T : 20$ $(T \rightarrow$ MOVE MEM T_x MEM $T_y)$
MOVE
$T : \infty$
MEM
MEM
$T : 13$
$(T \rightarrow$ MEM + CONST c $T_y)$
$T : 4$
$(T \rightarrow$ + CONST c $T_y)$
+
$(T \rightarrow * T_y T_x)$
$T : 3$
+
$T : 4$
$(T \rightarrow$ + CONST c $T_y)$
CONST &b
*
CONST &a
*
$T : 3$
$(T \rightarrow * T_y T_x)$
$(T \rightarrow$ CONST $c)$ $T : 1$
$(T \rightarrow$ CONST $c)$ $T : 1$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow$ CONST $c)$ $T : 1$
$T : 0$
$(T_t \rightarrow$ TEMP $t)$
$T : 1$
$(T \rightarrow$ CONST $c)$
$T : 0$
$(T_t \rightarrow$ TEMP $t)$

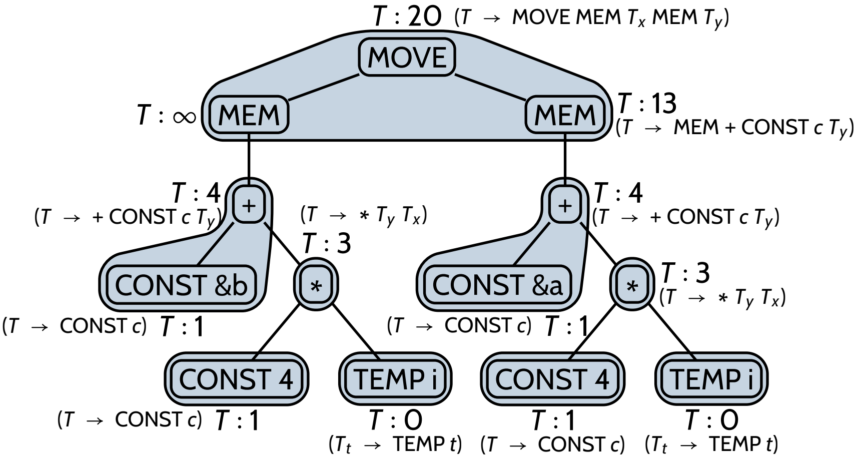
Action: select productions

Running dynamic programming on our IR tree


$T : 20$ $(T \rightarrow \text{MOVE MEM } T_x \text{ MEM } T_y)$
MOVE
$T : \infty$
MEM
MEM
$T : 13$
$(T \rightarrow \text{MEM + CONST } c\ T_y)$
$T : 4$
$(T \rightarrow +\text{ CONST } c\ T_y)$
+
$(T \rightarrow *\ T_y\ T_x)$
$T : 3$
+
$T : 4$
$(T \rightarrow +\text{ CONST } c\ T_y)$
CONST &b
*
CONST &a
*
$T : 3$
$(T \rightarrow *\ T_y\ T_x)$
$(T \rightarrow \text{CONST } c)$ $T : 1$
$(T \rightarrow \text{CONST } c)$ $T : 1$
CONST 4
TEMP i
CONST 4
TEMP i
$(T \rightarrow \text{CONST } c)$ $T : 1$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$
$T : 1$
$(T \rightarrow \text{CONST } c)$
$T : 0$
$(T_t \rightarrow \text{TEMP } t)$

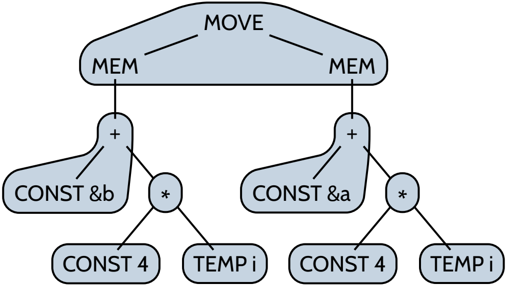
Action: select productions

Running dynamic programming on our IR tree


$T:20$ ($T \rightarrow$ MOVE MEM T_x MEM T_y)
MOVE
$T:\infty$
MEM
MEM
$T:13$
($T \rightarrow$ MEM + CONST c T_y)
$T:4$
($T \rightarrow$ + CONST c T_y)
+
($T \rightarrow * T_y T_x$)
$T:3$
+
$T:4$
($T \rightarrow$ + CONST c T_y)
CONST &b
*
CONST &a
*
$T:3$
($T \rightarrow * T_y T_x$)
($T \rightarrow$ CONST c) $T:1$
($T \rightarrow$ CONST c) $T:1$
CONST 4
TEMP i
CONST 4
TEMP i
($T \rightarrow$ CONST c) $T:1$
$T:0$
($T_t \rightarrow$ TEMP t)
$T:1$
($T \rightarrow$ CONST c)
$T:0$
($T_t \rightarrow$ TEMP t)

Action: done selecting productions

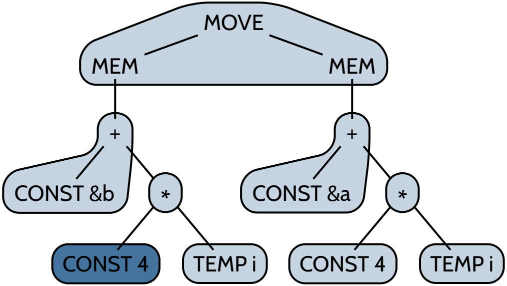
Running dynamic programming on our IR tree



Assembly code:

Action: emit assembly instructions

Running dynamic programming on our IR tree

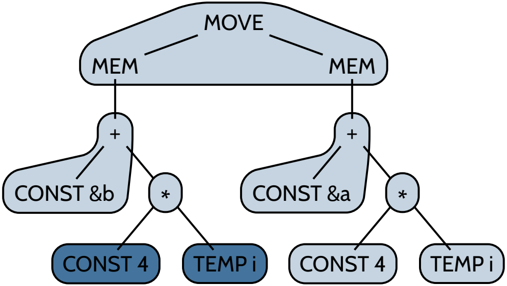


Assembly code:

ADDI $t_0 \leftarrow r_0 + \#4$

Action: emit assembly instructions

Running dynamic programming on our IR tree

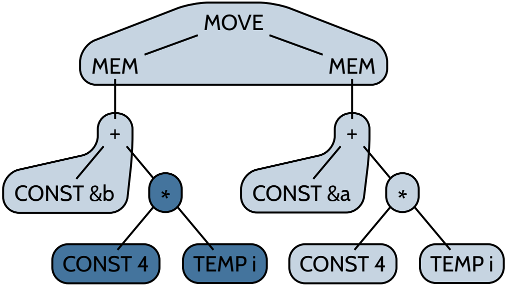


Assembly code:

ADDI $t_0 \leftarrow r_0 + \#4$

Action: emit assembly instructions

Running dynamic programming on our IR tree

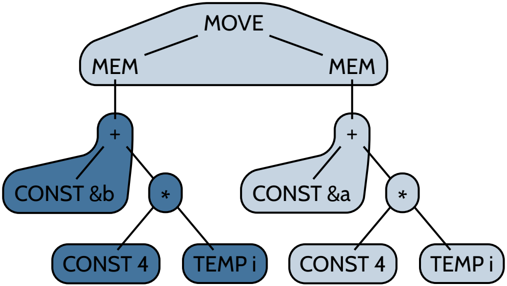


Assembly code:

```
ADDI    $t_0 \leftarrow r_0 + \#4$
MUL     $t_1 \leftarrow t_0 * t_i$
```

Action: emit assembly instructions

Running dynamic programming on our IR tree

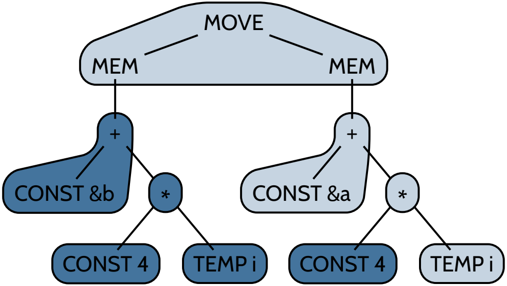


Assembly code:

```
ADDI    t0 ← r0 + #4
MUL     t1 ← t0 * ti
ADDI    t2 ← t1 + #4
```

Action: emit assembly instructions

Running dynamic programming on our IR tree

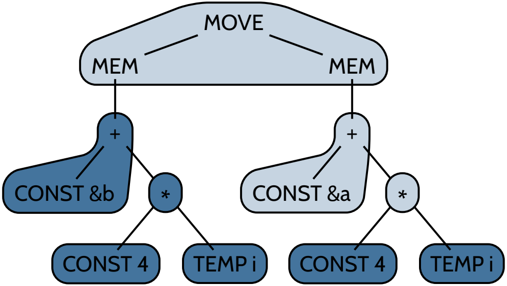


Assembly code:

ADDI	t_0	←	r_0	+	#4
MUL	t_1	←	t_0	$*$	t_i
ADDI	t_2	←	t_1	+	#4
ADDI	t_3	←	r_0	+	#4

Action: emit assembly instructions

Running dynamic programming on our IR tree

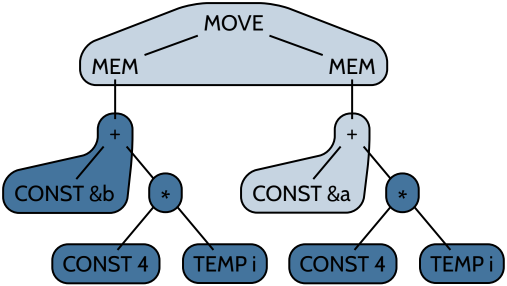


Assembly code:

ADDI	$t_0 \leftarrow r_0 + \#4$
MUL	$t_1 \leftarrow t_0 * t_i$
ADDI	$t_2 \leftarrow t_1 + \#4$
ADDI	$t_3 \leftarrow r_0 + \#4$

Action: emit assembly instructions

Running dynamic programming on our IR tree

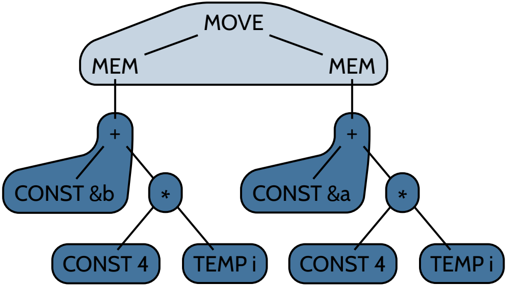


Assembly code:

ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$
ADDI $t_2 \leftarrow t_1 + \#4$
ADDI $t_3 \leftarrow r_0 + \#4$
MUL $t_4 \leftarrow t_3 * t_i$

Action: emit assembly instructions

Running dynamic programming on our IR tree

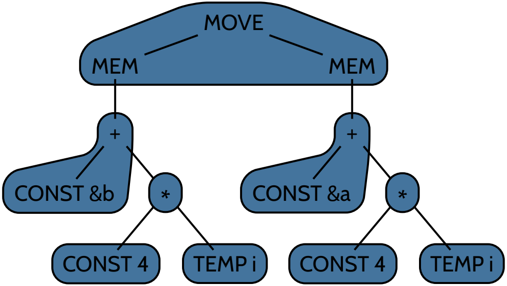


Assembly code:

ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$
ADDI $t_2 \leftarrow t_1 + \#4$
ADDI $t_3 \leftarrow r_0 + \#4$
MUL $t_4 \leftarrow t_3 * t_i$
ADDI $t_5 \leftarrow t_4 + \#4$

Action: emit assembly instructions

Running dynamic programming on our IR tree

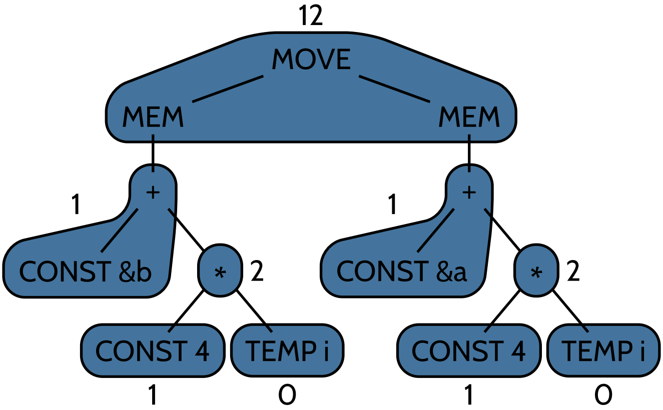


Assembly code:

ADDI	t_0	←	r_0	+	#4
MUL	t_1	←	t_0	$*$	t_i
ADDI	t_2	←	t_1	+	#4
ADDI	t_3	←	r_0	+	#4
MUL	t_4	←	t_3	$*$	t_i
ADDI	t_5	←	t_4	+	#4
MOVEM	$M[t_2]$	←	$M[t_5]$		

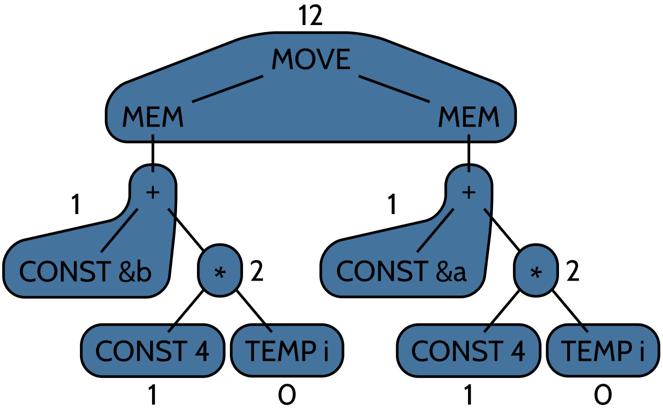
Action: done

Optimum tiling found with dynamic prog.



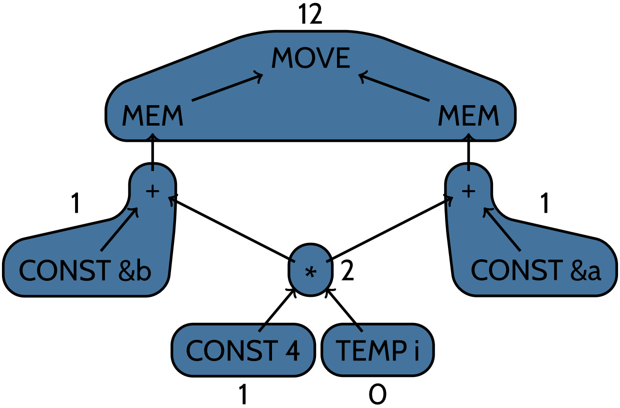
$\sum$ cost = 20

Can we do better?



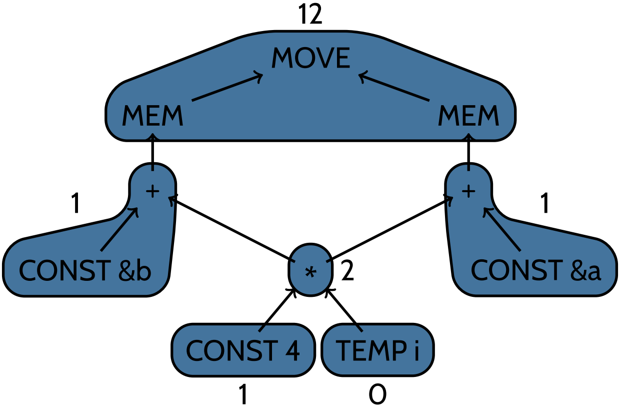
$\sum$ cost = 20

Yes, if our IR is instead represented as a **directed acyclic graph (DAG)** . . .



$\sum \text{cost} = 17$

. . . but finding optimum tilings for IR DAGs is an **NP-complete** problem



$\sum$ cost = 17

Summary

Macro expansion

- **Advantages:**
  - **Very simple** to implement
  - **Very fast** $\mathcal{O}(n)$
    - *n* is size of IR tree
    - Assuming tile set is fixed
- **Disadvantages:**
  - Only supports **single-node tiles**
  - May yield **suboptimal** tilings
- **Suitable for:**
  - Very simple (RISC) target architectures
    - 1-to-*n* mappings between IR operations and instructions
- **Modern implementations:**
  - Improved variant used in *GCC*

Maximum munch

- **Advantages over macro expansion:**
  - Supports **any-size** tree tiles
  - Always yields **optimal** tilings
  - **Very fast** $\mathcal{O}(n)$
    (provided single-node tiles exist for all IR operations)
- **Disadvantages:**
  - May yield **suboptimum** tilings
- **Suitable for:**
  - Target architectures where tile cost is **proportional** to tile size
- **Modern implementations:**
  - DAG-variant used in *LLVM*

Tree parsing

- **Advantages over maximum munch:**
  - Can **reuse** LR parsing techniques
  - **Very fast** $\mathcal{O}(n)$
- **Disadvantages:**
  - Can **fail** due to syntactic blocking
  - May still yield **suboptimum** tilings
- **Suitable for:**
  - Same as maximum munch
- **Modern implementations:**
  - None as far as I know

Dynamic programming

- **Advantages over tree parsing:**
  - Always yields **optimum** tilings
  - **Very fast** $\mathcal{O}(n)$
- **Disadvantages:**
  - More complicated compared to other approaches
  - Requires **IR trees** as input
- **Suitable for:**
  - Target architectures where all instructions are modeled as tree tiles
- **Modern implementations:**
  - *CoSy*
  - BURG $\xrightarrow{\text{“BURGer phenomenon”}}$ DBURG, GBURG, GPBURG, IBURG, JBURG, HBURG, LBURG, MBURG, OCAMLBURG, and WBURG

Further reading

- **Survey book:**
  - Gabriel Hjort Blindell – *Instruction Selection: Principles, Methods, and Applications* (2016) Springer.
    ISBN: 978-3-319-34019-7

    Free PDF:
    `http://kth.diva-portal.org/smash/get/diva2:951540/FULLTEXT01.pdf`