# ID2202 Lecture 07
# Instruction Selection

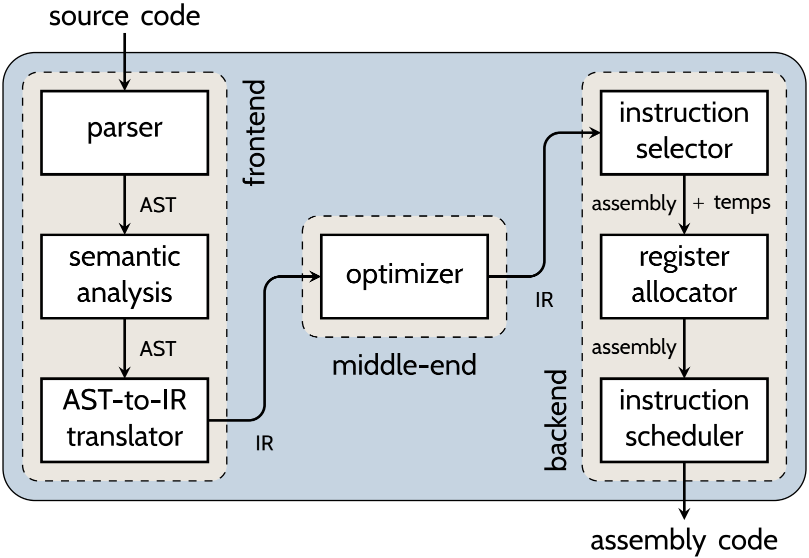
Gabriel Hjort Blindell
ghb@kth.se

SCALE, Software and Computer Systems (SCS)
KTH Royal Institute of Technology
SICS Swedish Institute of Computer Science
Stockholm, Sweden

# Say we want to compile the following program

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

# 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

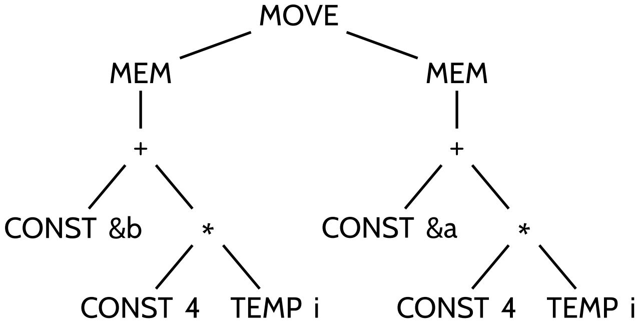
```
int a[ ];
int b[ ];
    ⋮
int num = . . .
for (int i = 0; i < num; i++)
{
  b[i] = a[i];
}
```

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

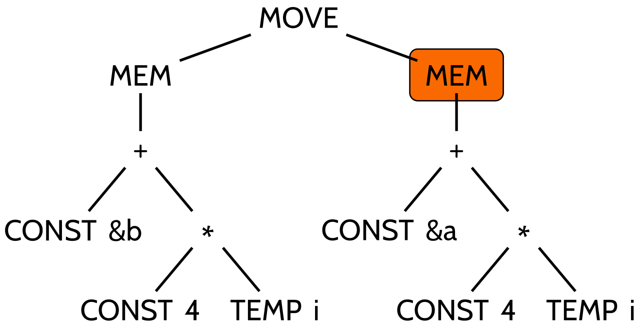
Assuming:

- 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];**

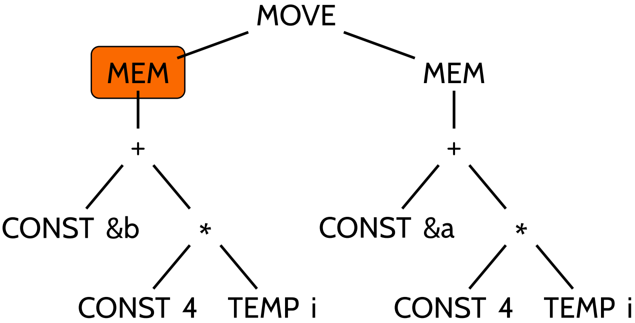


# IR tree of **b[i] = a[i];**



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:

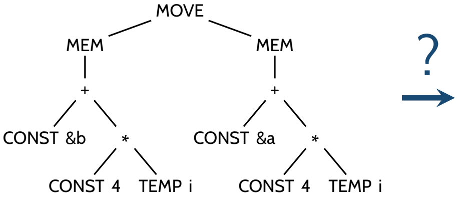
- $\mathrm{r}_i$ denotes "register *i*"
- $\mathrm{t}_i$ denotes "temporary *i*"
- *#c* denotes "integer constant *c*"
- *M*[*x*] denotes "memory value at address *x*"

# 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:**

- Every IR operation has predefined set of semantics
- Every assembly instruction has predefined set of semantics
- Data flow via temporaries
- For each IR operation:
  - Emit sequence of assembly instructions with equivalent semantics
  - 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

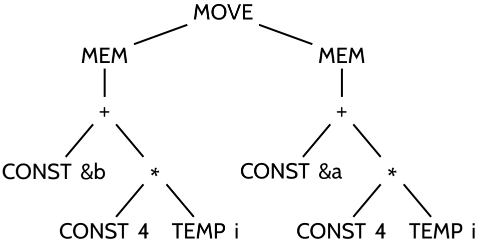
```
expand(MEM $E$) =
    if isRValue() then
        $t_x$ = getNewTemp()
        $t_y$ = getResultOf($E$)
        emit(“LOAD    $t_x \leftarrow M[t_y + \#0]$”)
        setResultIsIn($t_x$)
    else is L-value
        setResultIsIn(getResultOf($E$))
    endif
```

# Macros for MOVE

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

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

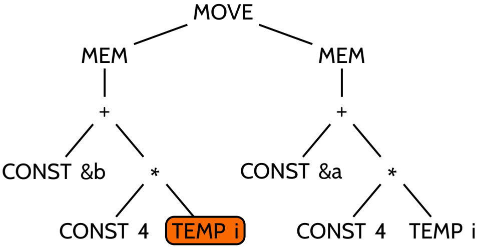
# Running macro expansion on our IR tree



**Assembly code:**

**Action:**

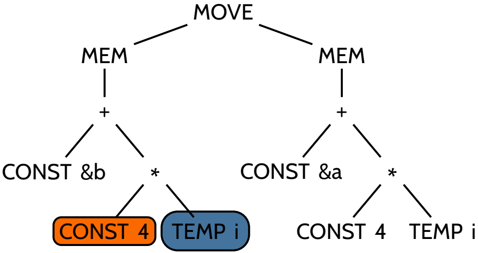
# Running macro expansion on our IR tree



**Assembly code:**

**Action:** execute corresponding macro on each node

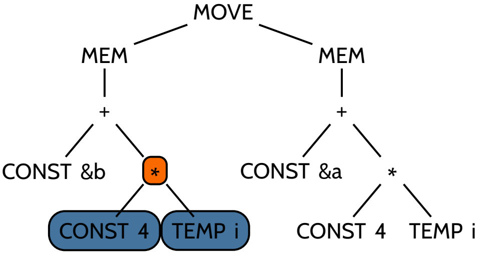
# Running macro expansion on our IR tree



**Assembly code:**

**Action:** execute corresponding macro on each node

# Running macro expansion on our IR tree

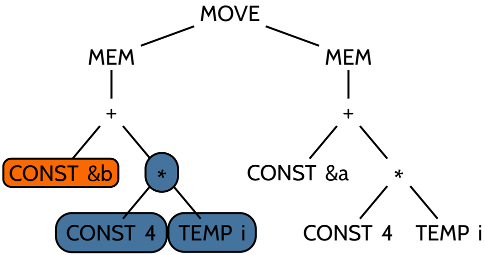


**Assembly code:**

ADDI $t_0 \leftarrow r_0 + \#4$

**Action:** execute corresponding macro on each node

# Running macro expansion on our IR tree

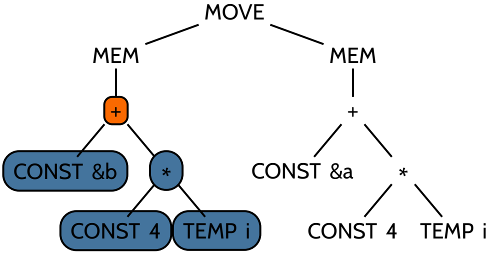


**Assembly code:**

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

**Action:** execute corresponding macro on each node

# Running macro expansion on our IR tree

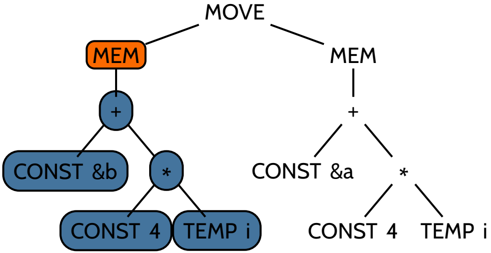


**Assembly code:**

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

**Action:** execute corresponding macro on each node

# Running macro expansion on our IR tree

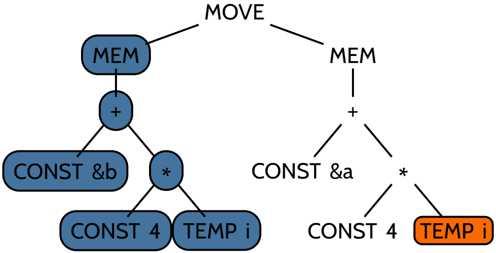


**Assembly code:**

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

**Action:** execute corresponding macro on each node

# Running macro expansion on our IR tree



**Assembly code:**

ADDI $t_0 \leftarrow r_0 + \#4$

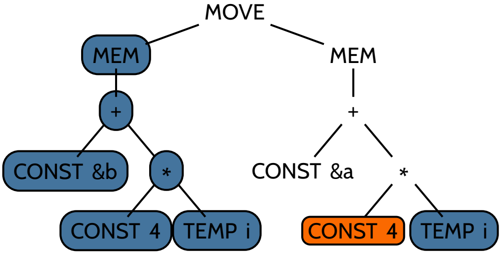
MUL $t_1 \leftarrow t_0 * t_i$

ADDI $t_2 \leftarrow r_0 + \#\&b$

ADD $t_3 \leftarrow t_2 + t_1$

**Action:** execute corresponding macro on each node

# Running macro expansion on our IR tree

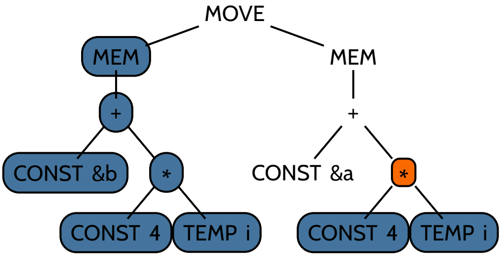


**Assembly code:**

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

**Action:** execute corresponding macro on each node

# Running macro expansion on our IR tree

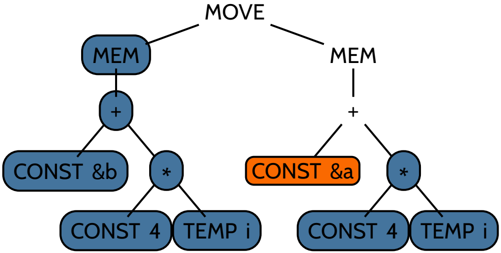


**Assembly code:**

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

**Action:** execute corresponding macro on each node

# Running macro expansion on our IR tree

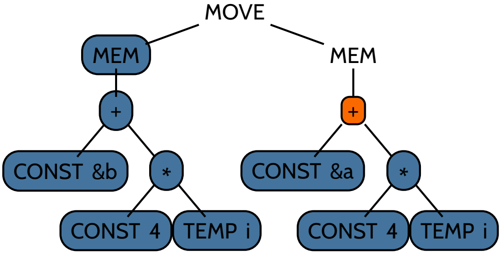


**Assembly code:**

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

**Action:** execute corresponding macro on each node

# Running macro expansion on our IR tree

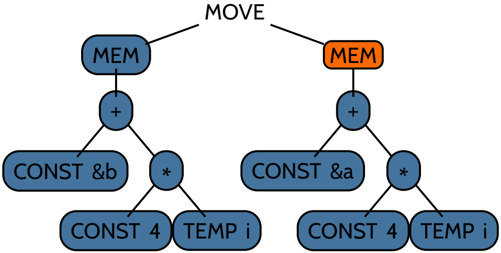


**Assembly code:**

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

**Action:** execute corresponding macro on each node

# Running macro expansion on our IR tree

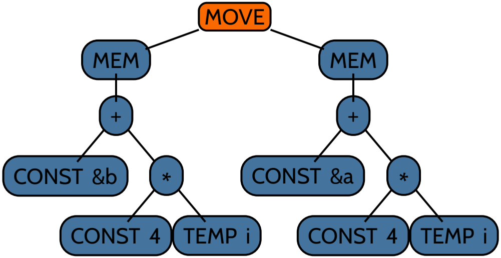


**Assembly code:**

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

**Action:** execute corresponding macro on each node

# Running macro expansion on our IR tree

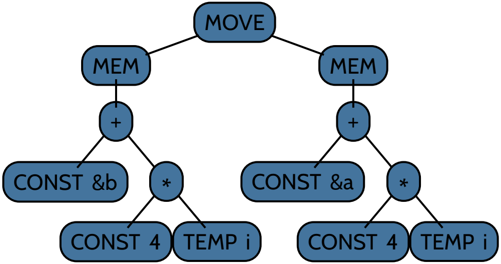


**Assembly code:**

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

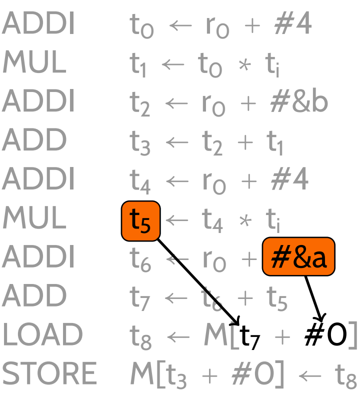
$$\sum \text{cost} = 30$$

# Can we do better?

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

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

# Result of improvement

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

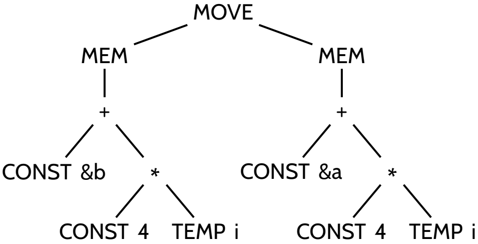
$$\sum \text{cost} = 28$$

# Why did not macro expansion emit this?

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

MUL $t_1 \leftarrow t_0 * t_i$

ADDI $t_2 \leftarrow r_0 + \#\&b$

ADD $t_3 \leftarrow t_2 + t_1$

ADDI $t_4 \leftarrow r_0 + \#4$

MUL $t_5 \leftarrow t_4 * t_i$

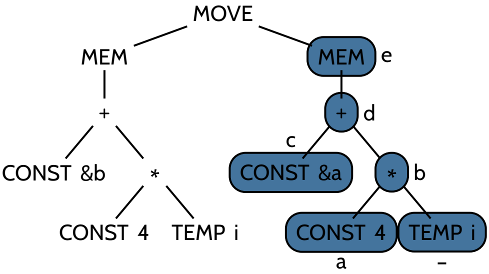
ADDI $t_6 \leftarrow r_0 + \#\&a$

ADD $t_7 \leftarrow t_6 + t_5$

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

STORE $M[t_3 + \#0] \leftarrow t_8$

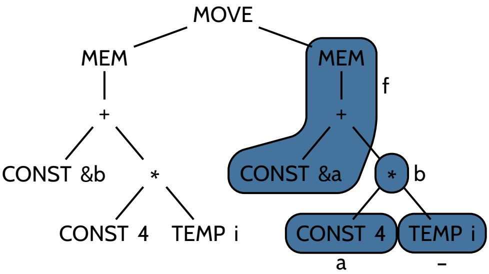
# **Limitation:** Macro expansion emits assembly code *one* IR operation at a time . . .



**Assembly code:**

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

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



**Assembly code:**

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


f  LOAD   t8 ← M[t5 + #&a]
   STORE  M[t3 + #0] ← t8
```

# 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 **tree tiles** (often also called **tree patterns**), such that:
  - every IR operation is covered by some 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 optimum/optimal tiling?

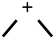
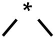
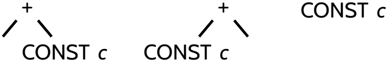
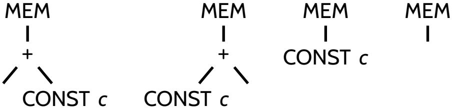
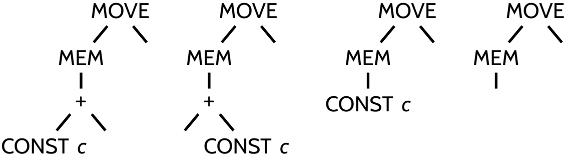
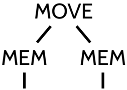
# Revisiting macro expansion

- 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

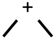
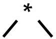
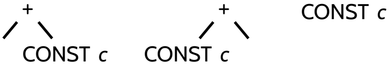
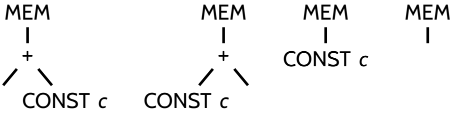
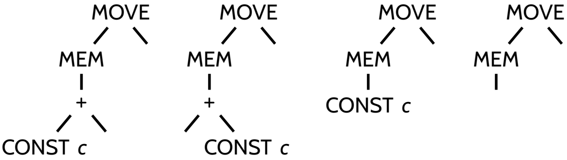
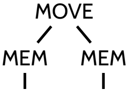
# Tiles set of macro expansion

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

# 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<br>+<br>CONST *c*<br>MEM<br>+<br>CONST *c*<br>MEM<br>CONST *c*<br>MEM |
| STORE | $M[t_x + \#c] \leftarrow t_y$ |  MOVE<br>MEM<br>+<br>CONST *c*<br>MOVE<br>MEM<br>+<br>CONST *c*<br>MOVE<br>MEM<br>CONST *c*<br>MOVE<br>MEM |
| MOVEM | $M[t_x] \leftarrow M[t_y]$ |  |

# **Problem:** How to use these efficiently?

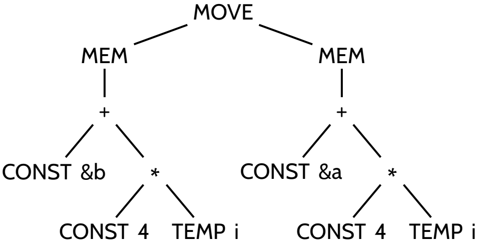
| Instruction | | Tiles |
|---|---|---|
| – | | TEMP *c* |
| ADD | $t_x \leftarrow t_y + t_z$ |  |
| MUL | $t_x \leftarrow t_y * t_z$ |  |
| ADDI | $t_x \leftarrow t_y + \#c$ |  |
| LOAD | $t_x \leftarrow M[t_y + \#c]$ |   MEM<br>+<br>CONST *c*<br>MEM<br>+<br>CONST *c*<br>MEM<br>CONST *c*<br>MEM |
|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

# $2^{nd}$ 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 in tiling:
    - Emit corresponding instruction

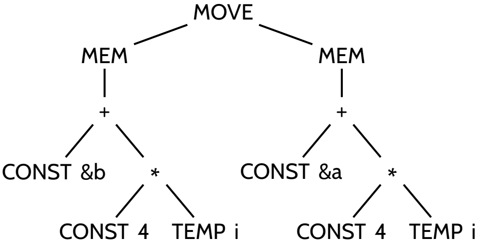
# Running maximum munch on our IR tree



**Assembly code:**

**Action:**

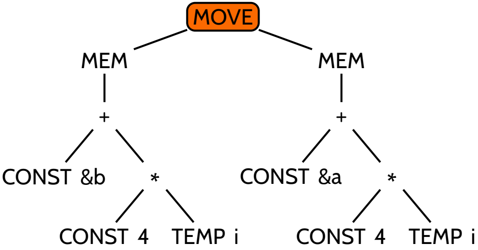
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

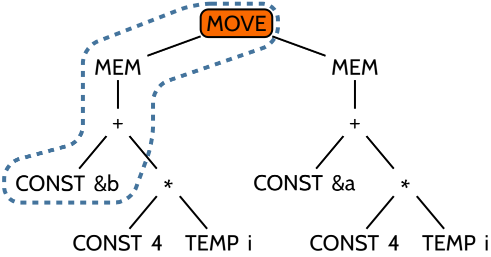
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

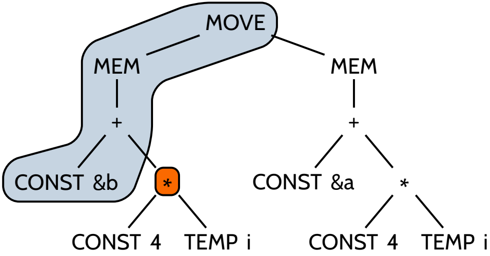
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

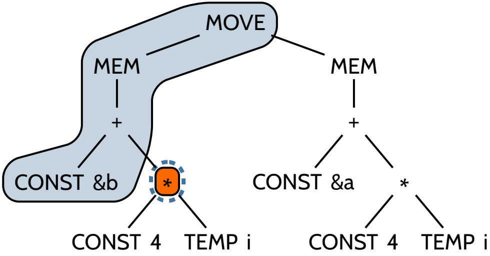
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

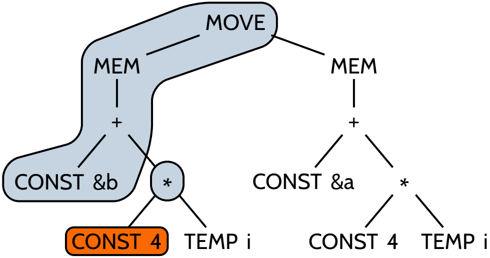
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

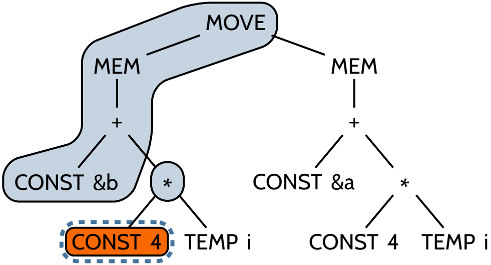
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

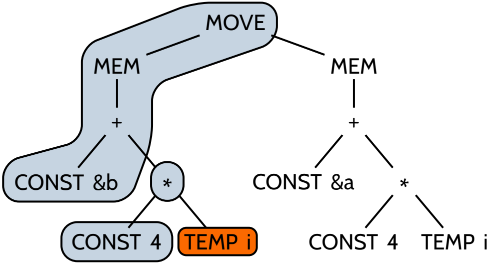
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

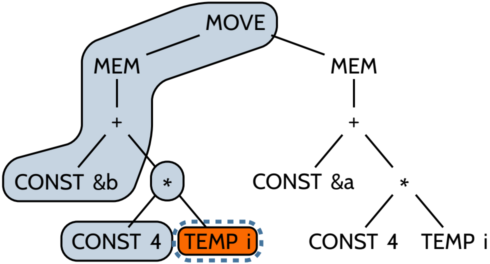
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

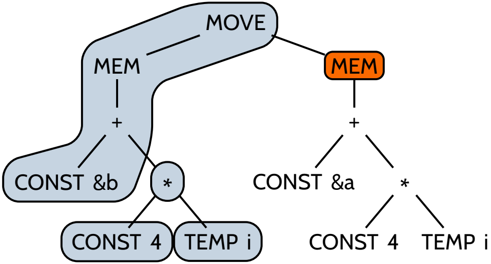
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

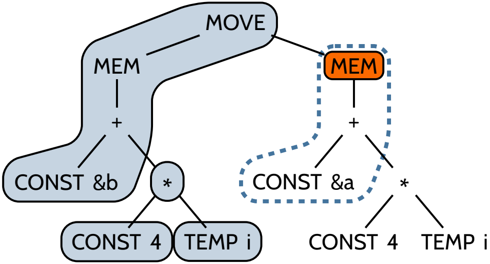
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

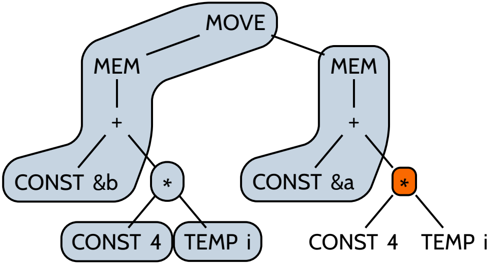
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

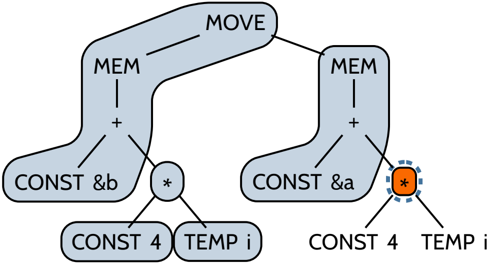
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

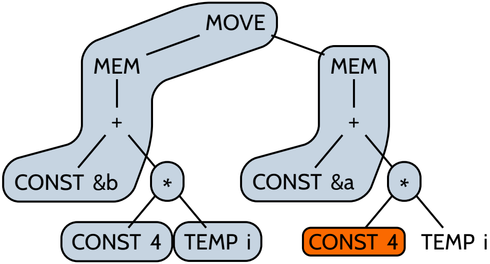
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

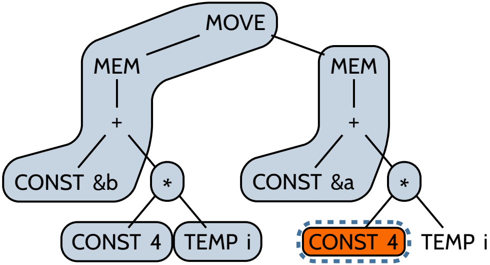
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

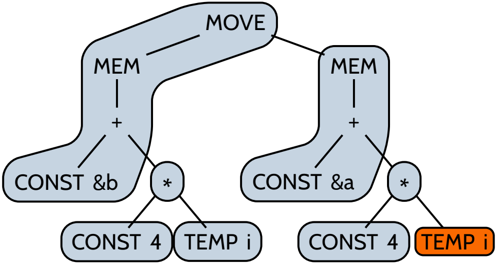
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

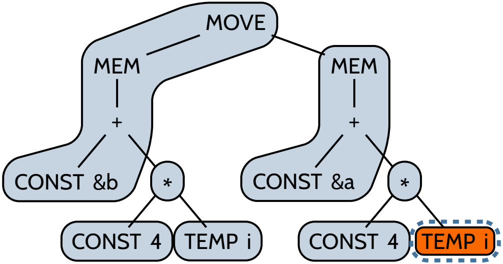
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

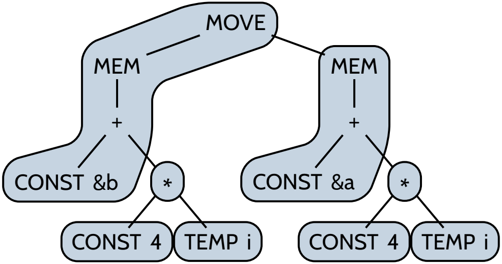
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** find largest matches

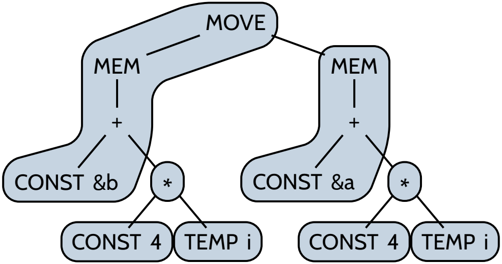
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** done finding matches

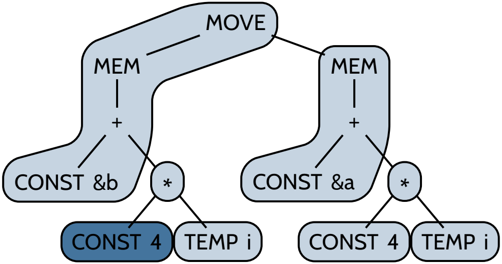
# Running maximum munch on our IR tree



**Assembly code:**

**Action:** emit assembly instructions

# Running maximum munch on our IR tree

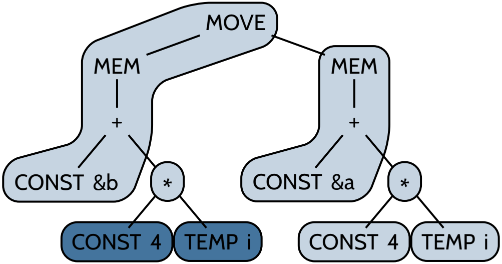


**Assembly code:**

ADDI $t_0 \leftarrow r_0 + \#4$

**Action:** emit assembly instructions

# Running maximum munch on our IR tree

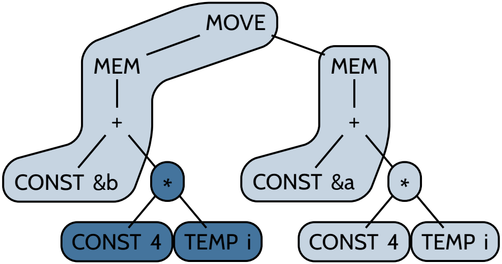


**Assembly code:**

ADDI $t_0 \leftarrow r_0 + \#4$

**Action:** emit assembly instructions

# Running maximum munch on our IR tree

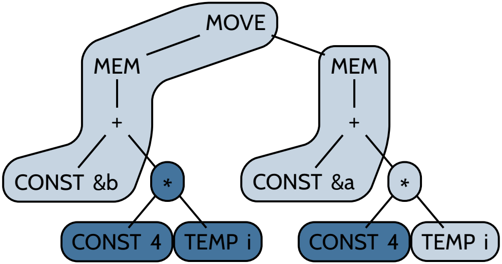


**Assembly code:**

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

**Action:** emit assembly instructions

# Running maximum munch on our IR tree

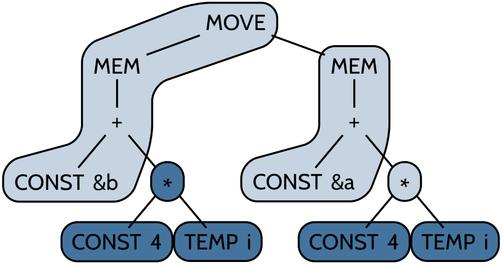


**Assembly code:**

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

**Action:** emit assembly instructions

# Running maximum munch on our IR tree

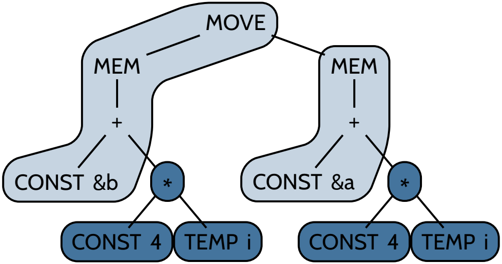


**Assembly code:**

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

**Action:** emit assembly instructions

# Running maximum munch on our IR tree

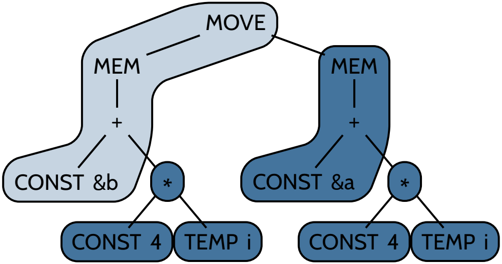


**Assembly code:**

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

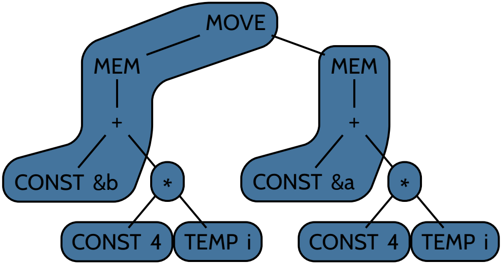


**Assembly code:**

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 $M[t_1 + \#\&b] \leftarrow t_4$

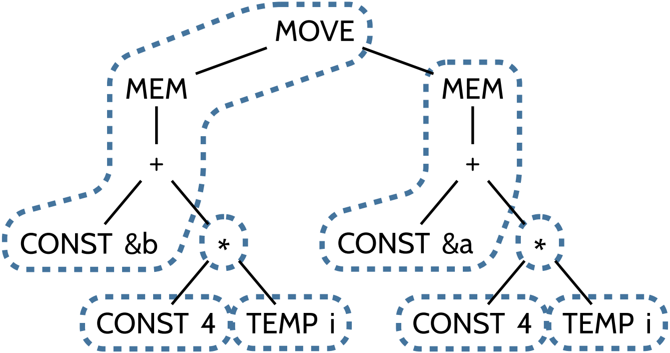
**Action:** done

# 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 |

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

# **Problem:** How to do tile matching?



- Can be implemented ad-hoc …
- … but how to retarget such an instruction selector?

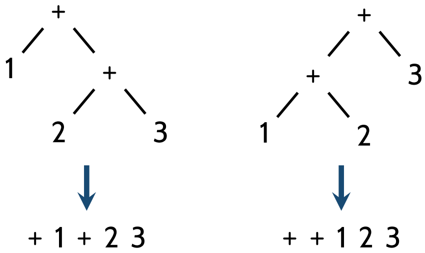
# 3$^{rd}$ approach: Tree parsing

**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 parser on IR string
- To emit assembly code:
  - When performing a reduction:
    - Emit corresponding instruction

# Transforming trees into strings

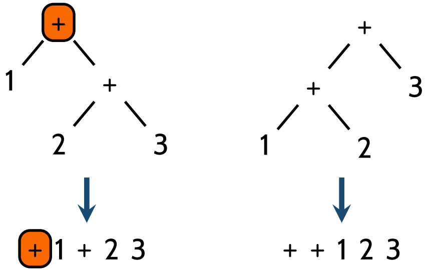
- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



- Achieved with top-down traversal of tree

# Transforming trees into strings

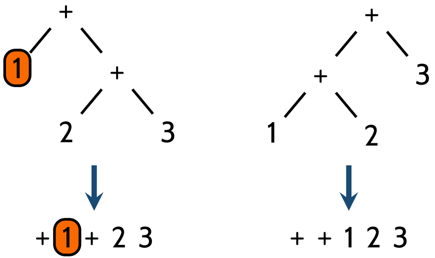
- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



- Achieved with top-down traversal of tree

# Transforming trees into strings

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



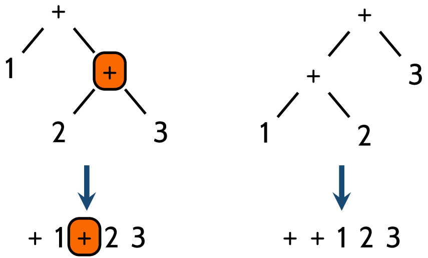
+ 1 + 2 3

+ + 1 2 3

- Achieved with top-down traversal of tree

# Transforming trees into strings

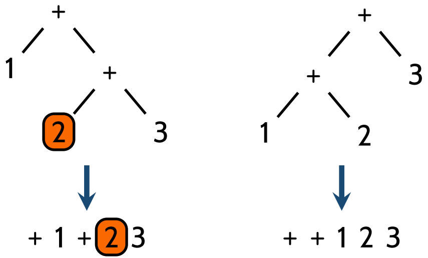
- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



- Achieved with top-down traversal of tree

# Transforming trees into strings

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



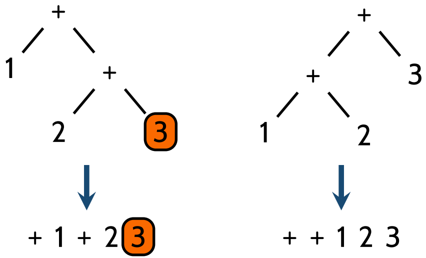
+ 1 + 2 3

+ + 1 2 3

- Achieved with top-down traversal of tree

# Transforming trees into strings

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



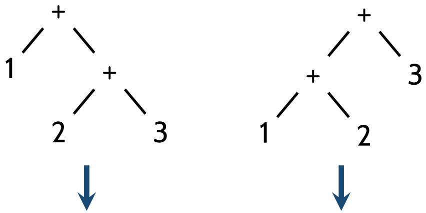
+ 1 + 2 3

+ + 1 2 3

- Achieved with top-down traversal of tree

# Transforming trees into strings

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



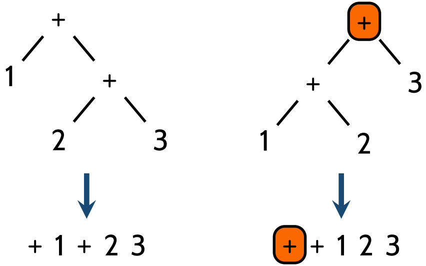
+ 1 + 2 3

+ + 1 2 3

- Achieved with top-down traversal of tree

# Transforming trees into strings

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



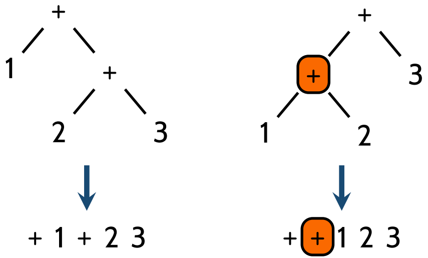
+ 1 + 2 3

+ + 1 2 3

- Achieved with top-down traversal of tree

# Transforming trees into strings

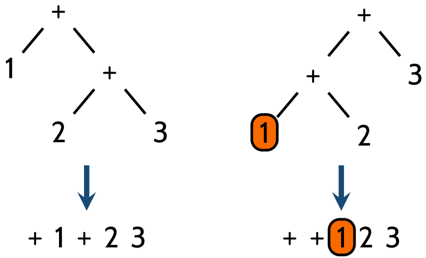
- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



- Achieved with top-down traversal of tree

# Transforming trees into strings

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



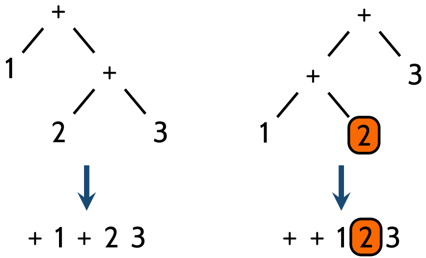
+ 1 + 2 3

+ + 1 2 3

- Achieved with top-down traversal of tree

# Transforming trees into strings

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



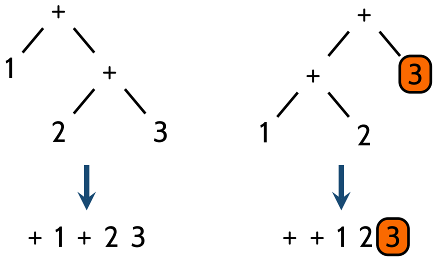
+ 1 + 2 3

+ + 1 2 3

- Achieved with top-down traversal of tree

# Transforming trees into strings

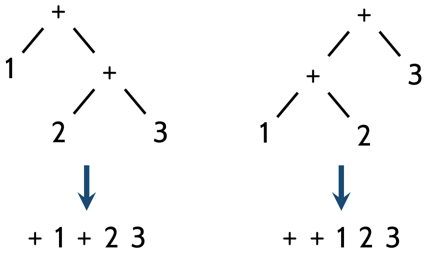
- **Polish notation:**
  - Operator is placed *in front* of arguments
  - Parentheses superfluous if all operators have fixed **arity** (number of arguments)



- Achieved with top-down traversal of tree

# Transforming trees into strings

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

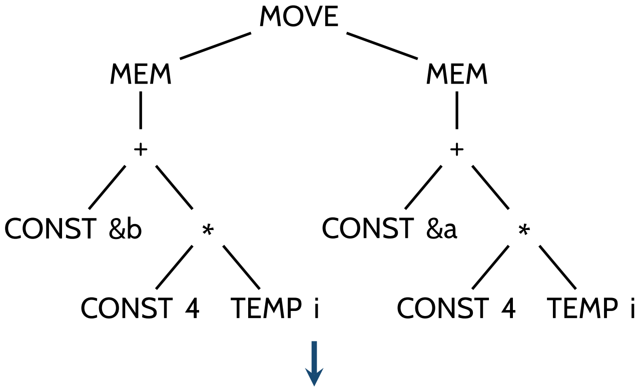


+ 1 + 2 3

+ + 1 2 3

- Achieved with top-down traversal of tree

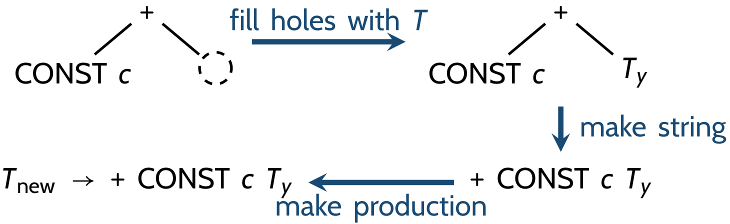
# Transforming our IR tree into an IR string



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

# Deriving the tree grammar

- Introduce nonterminal $T_t$ to represent temporaries
  - Subscript refers to some specific temporary $t$
- Introduce start symbol:
  - $S \rightarrow T\ \$$
- For each tile:



$+$

CONST $c$

fill holes with $T$

$+$

CONST $c$ $T_y$

make string

$T_{\text{new}} \rightarrow +\ \text{CONST}\ c\ T_y$

make production

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

- In relation to expansion macros:
  - $T_y$ corresponds to "$\text{t}_y$ = getResultOf($E_{\text{rhs}}$)"
  - $T_{\text{new}}$ corresponds to "$\text{t}_{\text{new}}$ = getNewTemp() . . . setResultIsIn($\text{t}_{\text{new}}$)"

# Tree grammar for *Jouette*

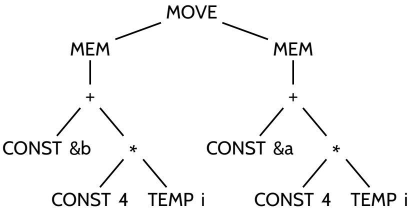
| Instruction | | Productions |
|---|---|---|
| – | | $T_t \rightarrow \mathrm{TEMP}\ t$ |
| ADD | $\mathrm{t_{new}} \leftarrow \mathrm{t_y} + \mathrm{t_z}$ | $T_{\mathrm{new}} \rightarrow +\ T_y\ T_z$ |
| MUL | $\mathrm{t_{new}} \leftarrow \mathrm{t_y} * \mathrm{t_z}$ | $T_{\mathrm{new}} \rightarrow *\ T_y\ T_z$ |
| ADDI | $\mathrm{t_{new}} \leftarrow \mathrm{t_y} + \#c$ | $T_{\mathrm{new}} \rightarrow +\ T_y\ \mathrm{CONST}\ c$<br>$T_{\mathrm{new}} \rightarrow +\ \mathrm{CONST}\ c\ T_y$<br>$T_{\mathrm{new}} \rightarrow \mathrm{CONST}\ c$ |
| LOAD | $\mathrm{t_{new}} \leftarrow \mathrm{M}[\mathrm{t_y} + \#c]$ | $T_{\mathrm{new}} \rightarrow \mathrm{MEM}\ +\ T_y\ \mathrm{CONST}\ c$<br>$T_{\mathrm{new}} \rightarrow \mathrm{MEM}\ +\ \mathrm{CONST}\ c\ T_y$<br>$T_{\mathrm{new}} \rightarrow \mathrm{MEM}\ \mathrm{CONST}\ c$<br>$T_{\mathrm{new}} \rightarrow \mathrm{MEM}\ T_y$ |
| STORE | $\mathrm{M}[\mathrm{t_x} + \#c] \leftarrow \mathrm{t_y}$ | $T \rightarrow \mathrm{MOVE}\ \mathrm{MEM}\ +\ \mathrm{CONST}\ c\ T_x\ T_y$<br>$T \rightarrow \mathrm{MOVE}\ \mathrm{MEM}\ +\ T_x\ \mathrm{CONST}\ c\ T_y$<br>$T \rightarrow \mathrm{MOVE}\ \mathrm{MEM}\ \mathrm{CONST}\ c\ T_y$<br>$T \rightarrow \mathrm{MOVE}\ \mathrm{MEM}\ T_x\ T_y$ |
| MOVEM | $\mathrm{M}[\mathrm{t_x}] \leftarrow \mathrm{M}[\mathrm{t_y}]$ | $T \rightarrow \mathrm{MOVE}\ \mathrm{MEM}\ T_x\ \mathrm{MEM}\ T_y$ |

# Tree grammars derived from assembly instructions often highly ambiguous

- 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



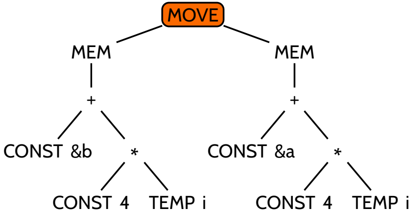
**Assembly code:**

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

**Action:**

**Stack:**

# Running tree parsing on our IR tree



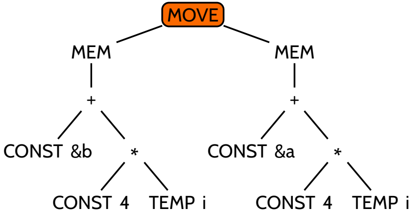
**Assembly code:**

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



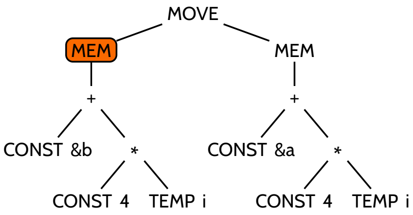
**Assembly code:**

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



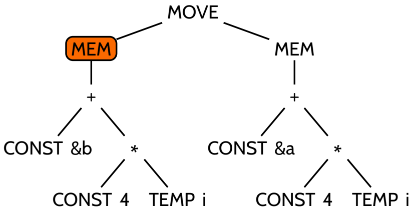
**Assembly code:**

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



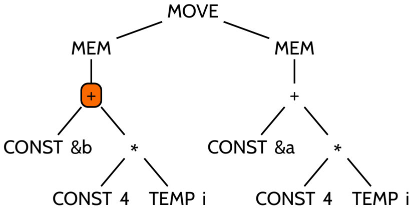
**Assembly code:**

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



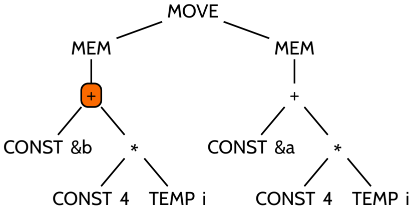
**Assembly code:**

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



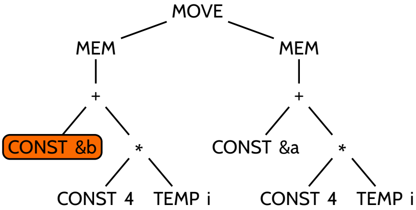
**Assembly code:**

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



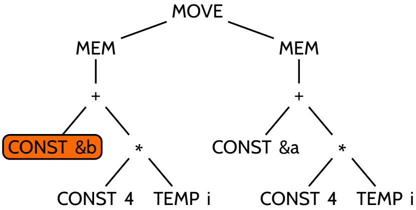
**Assembly code:**

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



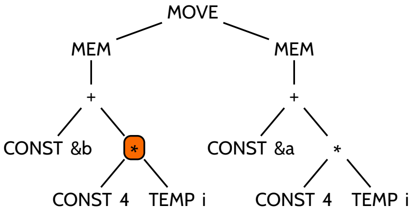
**Assembly code:**

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



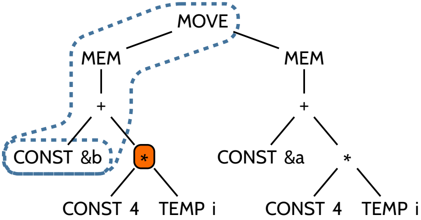
**Assembly code:**

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 $

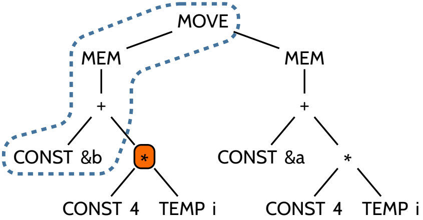
**Action:** shift-reduce conflict!

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

$T_{\text{new}} \rightarrow$ CONST $c$ *reducible now*

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

# Running tree parsing on our IR tree



**Assembly code:**

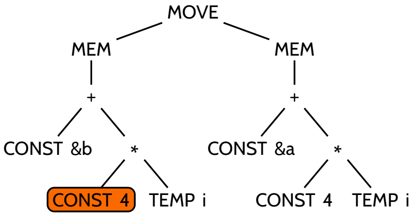
MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

**Action:** always shift

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

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

# Running tree parsing on our IR tree



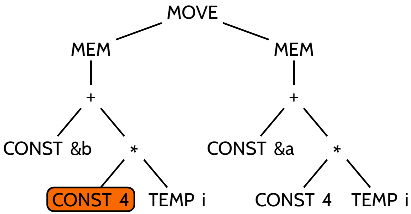
**Assembly code:**

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



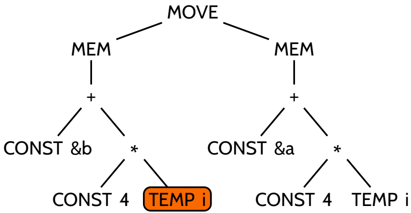
**Assembly code:**

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



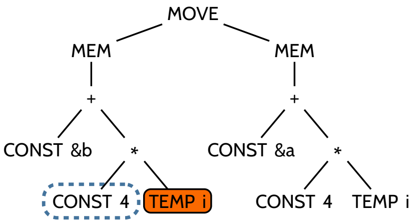
**Assembly code:**

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 $

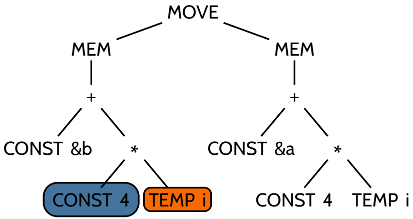
**Action:** reduce

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

$T_{\text{new}} \rightarrow \text{CONST } c$

*due to* $T_{\text{new}} \rightarrow * \; T_y \; T_z$

# Running tree parsing on our IR tree



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

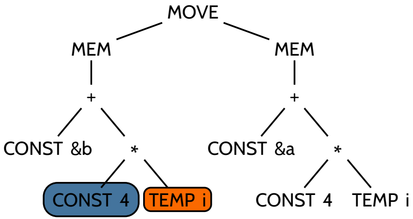
**Action:**

**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:**

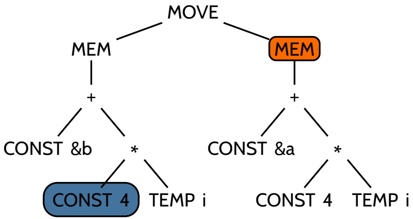
ADDI $t_0 \leftarrow r_0 + \#4$

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:**

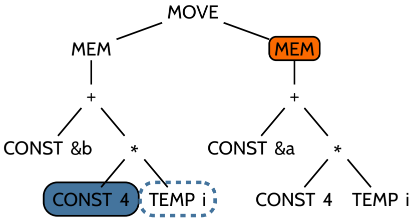
ADDI $t_0 \leftarrow r_0 + \#4$

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 $

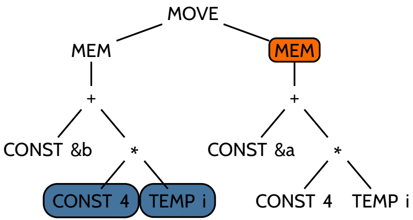
**Action:** reduce

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

$T_t \rightarrow$ TEMP $t$

*due to* $T_{new} \rightarrow * \ T_y \ T_z$

# Running tree parsing on our IR tree



**Assembly code:**

ADDI $t_0 \leftarrow r_0 + \#4$

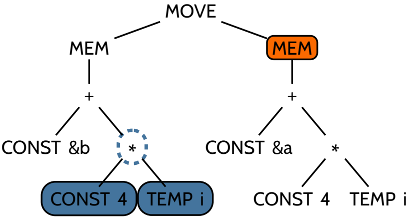
MOVE MEM + CONST &b * CONST 4 TEMP i

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

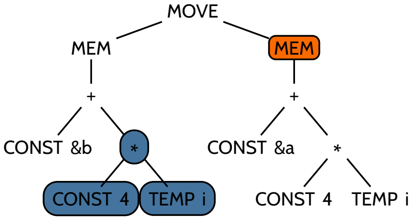
MEM + CONST &a * CONST 4 TEMP i \$

**Action:** reduce

**Stack:** MOVE MEM + CONST &b $* \; T_0 \; T_i$

$T_{new} \rightarrow * \; T_y \; T_z$

# Running tree parsing on our IR tree



**Assembly code:**

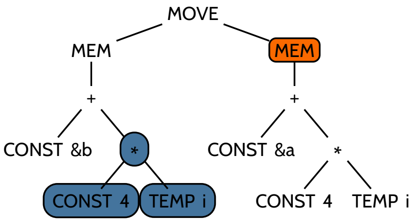
| | |
|---|---|
| ADDI | $t_0 \leftarrow r_0 + \#4$ |
| MUL | $t_1 \leftarrow t_0 * t_i$ |

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:**

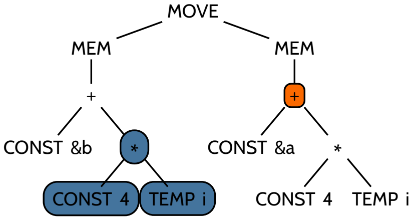
| | |
|---|---|
| ADDI | $t_0 \leftarrow r_0 + \#4$ |
| MUL | $t_1 \leftarrow t_0 * t_i$ |

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:**

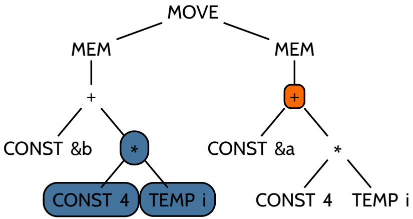
| | |
|---|---|
| ADDI | $t_0 \leftarrow r_0 + \#4$ |
| MUL | $t_1 \leftarrow t_0 * t_i$ |

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:**

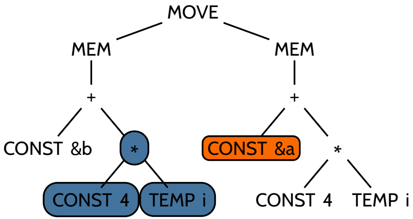
ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$

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:**

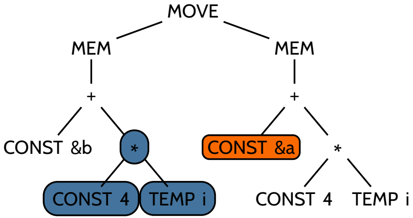
```
ADDI    t0 ← r0 + #4
MUL     t1 ← t0 * ti
```

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:**

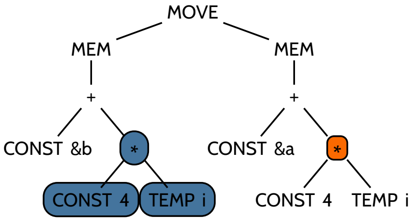
| | |
|---|---|
| ADDI | $t_0 \leftarrow r_0 + \#4$ |
| MUL | $t_1 \leftarrow t_0 * t_i$ |

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:**

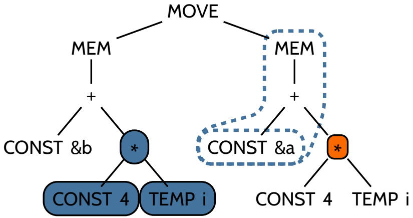
```
ADDI   t0 ← r0 + #4
MUL    t1 ← t0 * ti
```

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 + 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 $

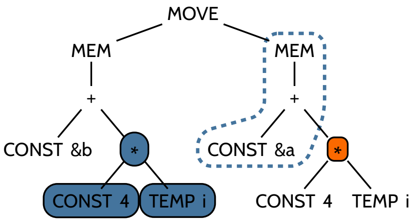
**Action:** shift-reduce conflict!

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

$T_{new} \rightarrow$ 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 | $t_1 \leftarrow t_0 * t_i$ |

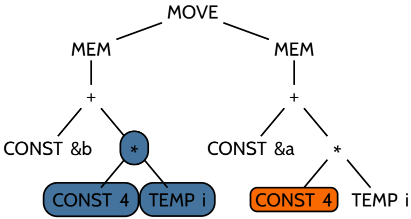
MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

**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:**

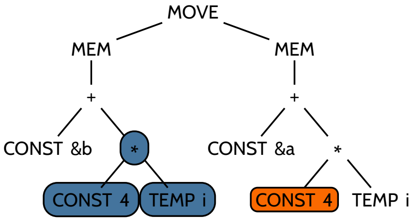
ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$

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:**

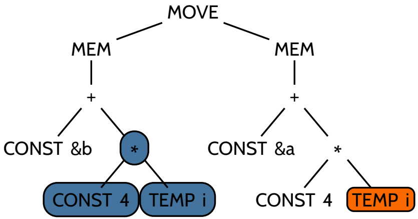
ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$

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:**

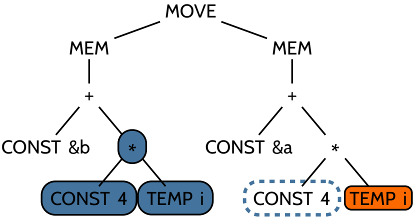
| | |
|---|---|
| ADDI | $t_0 \leftarrow r_0 + \#4$ |
| MUL | $t_1 \leftarrow t_0 * t_i$ |

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 $

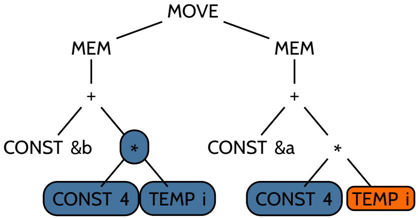
**Action:** reduce

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

$T_{new} \rightarrow$ CONST $c$

*due to* $T_{new} \rightarrow * \; T_y \; T_z$

# Running tree parsing on our IR tree



**Assembly code:**

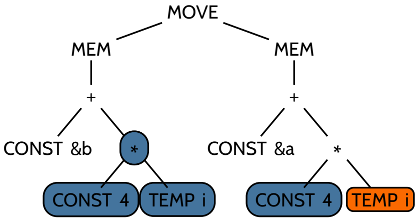
| | |
|---|---|
| ADDI | $t_0 \leftarrow r_0 + \#4$ |
| MUL | $t_1 \leftarrow t_0 * t_i$ |
| ADDI | $t_2 \leftarrow r_0 + \#4$ |

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:**

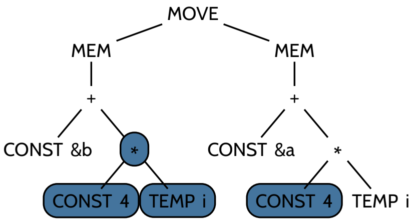
ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$
ADDI $t_2 \leftarrow r_0 + \#4$

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

**Action:** shift

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

# Running tree parsing on our IR tree



**Assembly code:**

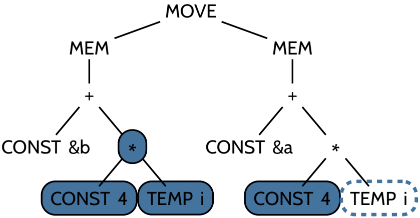
```
ADDI    t0 ← r0 + #4
MUL     t1 ← t0 * ti
ADDI    t2 ← r0 + #4
```

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 $

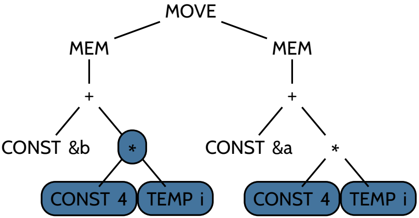
**Action:** reduce

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

$T_t \longrightarrow$ TEMP $t$

*due to* $T_{new} \longrightarrow * \; T_y \; T_z$

# Running tree parsing on our IR tree



**Assembly code:**

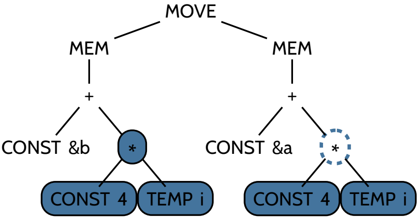
| | |
|---|---|
| ADDI | $t_0 \leftarrow r_0 + \#4$ |
| MUL | $t_1 \leftarrow t_0 * t_i$ |
| ADDI | $t_2 \leftarrow r_0 + \#4$ |

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 | $t_2 \leftarrow r_0 + \#4$ |

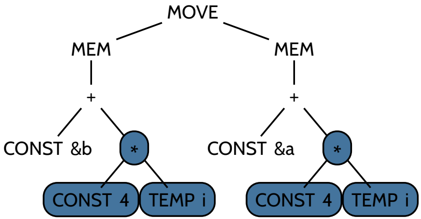
MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

**Action:** reduce

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

$T_{new} \longrightarrow * \; T_y \; T_z$

# Running tree parsing on our IR tree



## Assembly code:

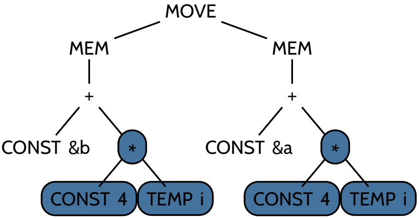
ADDI $t_0 \leftarrow r_0 + \#4$
MUL $t_1 \leftarrow t_0 * t_i$
ADDI $t_2 \leftarrow r_0 + \#4$
MUL $t_3 \leftarrow t_2 * t_i$

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 $

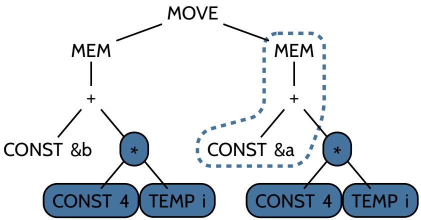
**Action:** reduce-reduce conflict!

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

$T_{new}$ → + CONST $c$ $T_y$

$T_{new}$ → 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
```

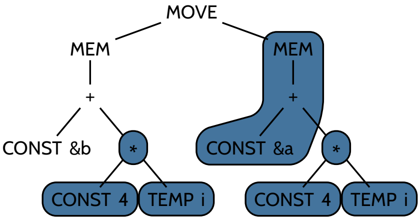
MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

**Action:** reduce longest production

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

$T_{new}$ → MEM + CONST $c$ $T_y$

# Running tree parsing on our IR tree



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

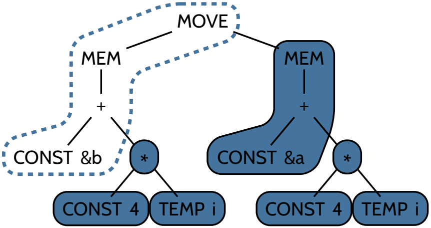
## Assembly code:

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:**

**Stack:** MOVE MEM + CONST &b $T_1$ $T_4$

# Running tree parsing on our IR tree



**Assembly code:**

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

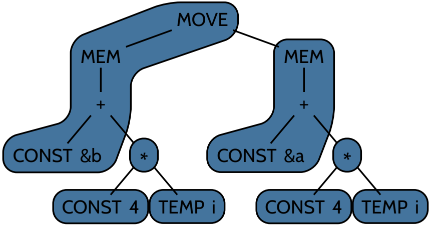
MOVE MEM + CONST &b * CONST 4 TEMP i
MEM + CONST &a * CONST 4 TEMP i $

**Action:** reduce

**Stack:** MOVE MEM + CONST &b $T_1$ $T_4$

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

# Running tree parsing on our IR tree



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

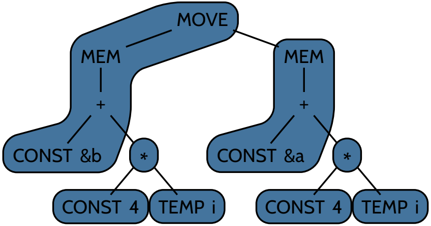
**Assembly code:**

| | |
|---|---|
| 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:**

**Stack:** $T$

# Running tree parsing on our IR tree



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

**Action:** accept

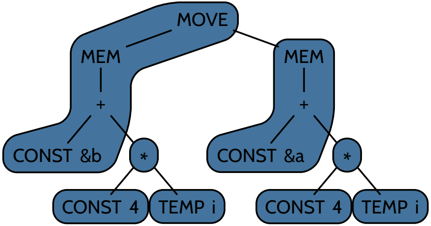
**Stack:** *T*

*S* → *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



## 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 | $M[t_1 + \#\&b] \leftarrow t_4$ |

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

**Action:** done

**Stack:**

# **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 *

$T_{\text{new}} \rightarrow \text{CONST } c$ *reducible now*

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

- **Stack some time after shifting:**

**Stack:** . . . MEM + CONST 4 $T_5$ $X$

$T \rightarrow \text{MEM} + \text{CONST } c \ \ T_x \ \ T_y$ *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 |

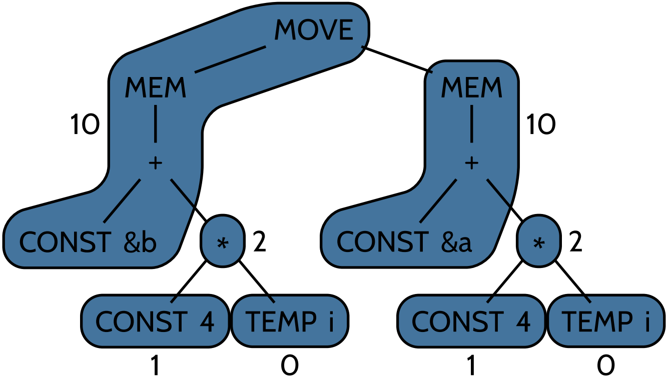
$$\sum \text{cost} = 26$$

# 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 |

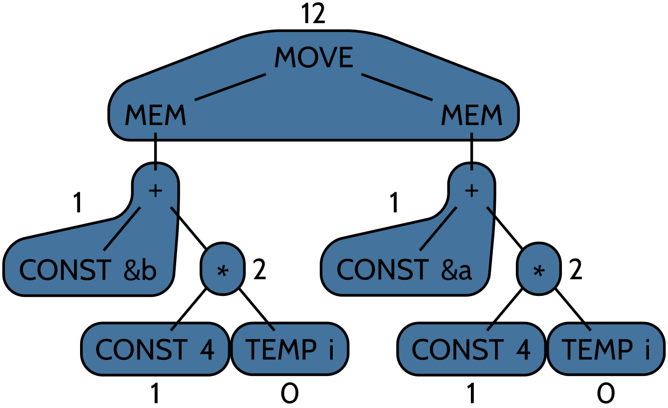
$$\sum \text{cost} = 26$$

# Optimal tiling found with tree parsing



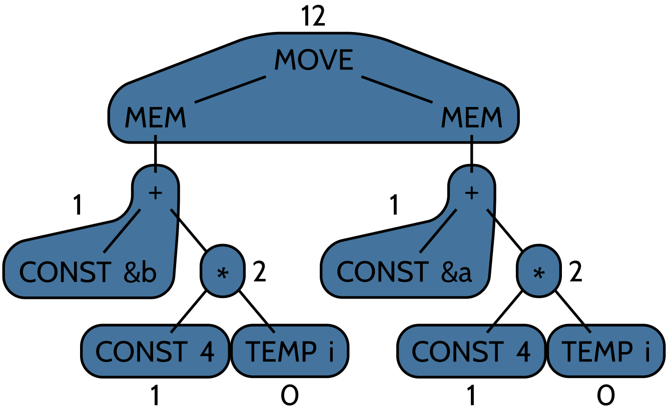
$$\sum \text{cost} = 26$$

# Optimum tiling



$$\sum \text{cost} = 20$$

# Need non-greedy approaches to find this tiling



$$\sum \text{cost} = 20$$

# $4^{th}$ 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 in tiling:
    - Emit corresponding instruction

# 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 $\mathcal{O}(n)$

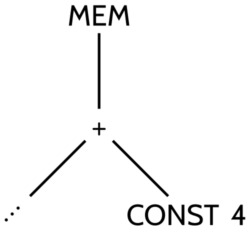
# Cost of reduction

- Cost of reducing nonterminal *nt* using production *P*:

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

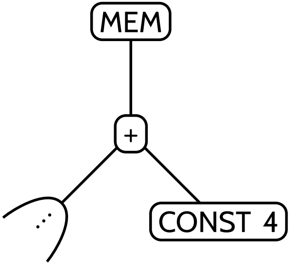
$c_P$ = cost of *P* itself
$c_i$ = cost of *i*th nonterminal appearing in *P*

# Computing costs on example



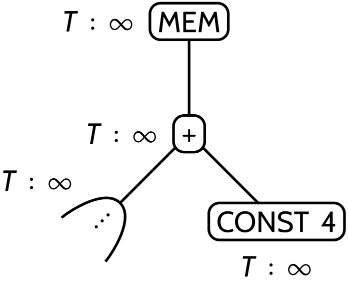
**Action:**

# Computing costs on example



**Action:** added boxes for better readability

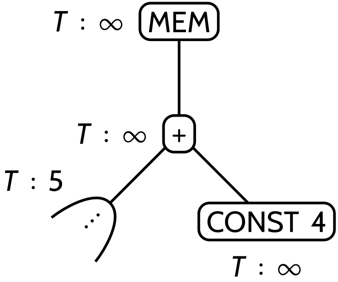
# Computing costs on example



**Action:** initialize all costs to $\infty$

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

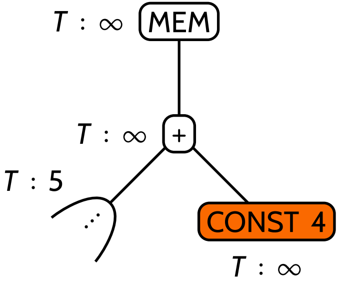
# Computing costs on example



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

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

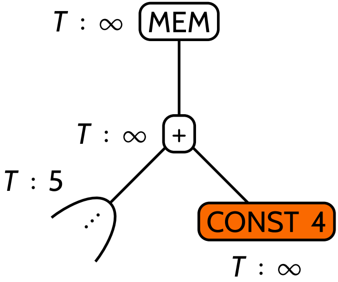
# Computing costs on example



**Action:** start at CONST 4 node

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

# Computing costs on example

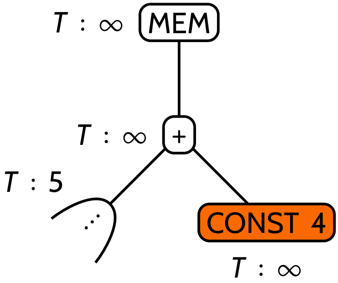


| **Productions:** | **Costs:** |
|---|---|
| $(T_{\text{new}} \rightarrow \text{CONST } c)$ | 1 |

**Action:** get productions of tiles that match

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

# Computing costs on example

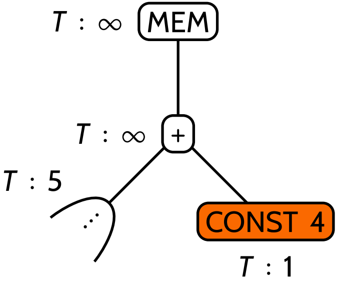


| **Productions:** | **Costs:** |
|---|---|
| $(T_{\text{new}} \rightarrow \text{CONST } c)$ | $1 + 0$ |

**Action:** compute costs of reduction

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

# Computing costs on example

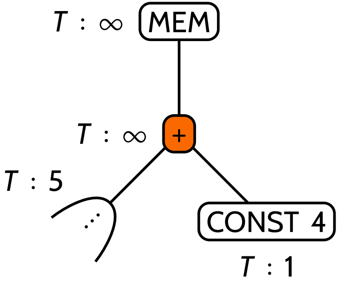


| **Productions:** | **Costs:** |
|---|---|
| $(T_{\text{new}} \rightarrow \text{CONST } c)$ | 1 |

**Action:** select production with least cost

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

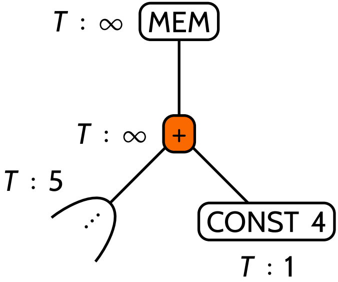
# Computing costs on example



**Action:** continue to + node

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

# Computing costs on example

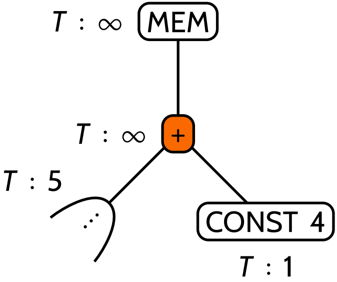


| **Productions:** | **Costs:** |
|---|---|
| $(T_{\text{new}} \rightarrow + \; T_y \; T_z)$ | 1 |
| $(T_{\text{new}} \rightarrow + \; T_y \; \text{CONST} \; c)$ | 1 |

**Action:** get productions of tiles that match

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

# Computing costs on example

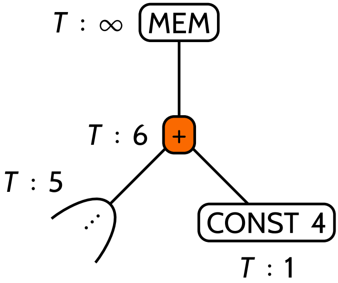


| **Productions:** | **Costs:** |
|---|---|
| $(T_{\text{new}} \rightarrow + \ T_y \ T_z)$ | $1 + 5 + 1$ |
| $(T_{\text{new}} \rightarrow + \ T_y \ \text{CONST} \ c)$ | $1 + 5$ |

**Action:** compute costs of reduction

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

# Computing costs on example

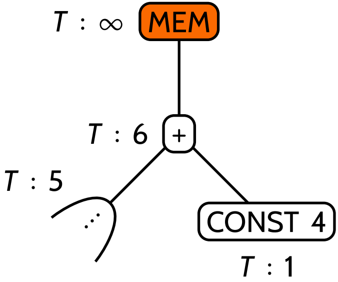


| **Productions:** | **Costs:** |
|---|---|
| $(T_{\text{new}} \rightarrow + \ T_y \ T_z)$ | 7 |
| $(T_{\text{new}} \rightarrow + \ T_y$ CONST $c)$ | 6 |

**Action:** select production with least cost

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

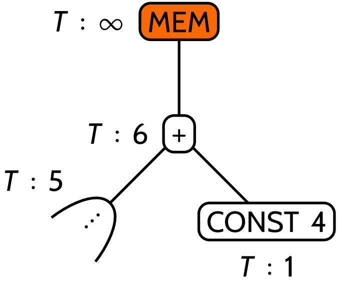
# Computing costs on example



**Action:** continue to *MEM* node

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

# Computing costs on example

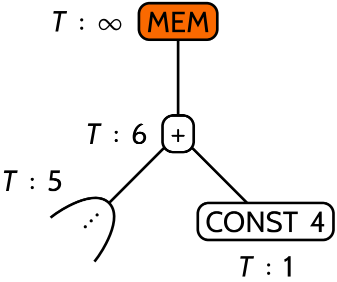


| **Productions:** | **Costs:** |
|---|---|
| $(T_{\text{new}} \rightarrow \text{MEM}\ T_y)$ | 10 |
| $(T_{\text{new}} \rightarrow \text{MEM} + T_y\ \text{CONST}\ c)$ | 10 |

**Action:** get productions of tiles that match

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

# Computing costs on example

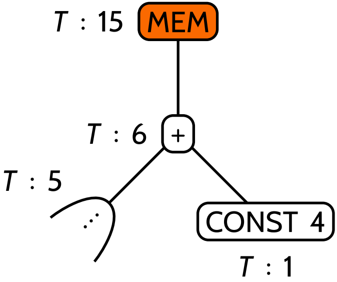


| **Productions:** | **Costs:** |
|---|---|
| $(T_{\text{new}} \rightarrow \text{MEM } T_y)$ | $10 + 6$ |
| $(T_{\text{new}} \rightarrow \text{MEM } + T_y \text{ CONST } c)$ | $10 + 5$ |

**Action:** compute costs of reduction

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

# Computing costs on example

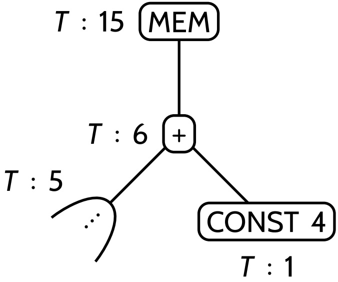


| **Productions:** | **Costs:** |
|---|---|
| $(T_{\text{new}} \rightarrow \text{MEM } T_y)$ | 16 |
| $(T_{\text{new}} \rightarrow \text{MEM} + T_y \text{ CONST } c)$ | 15 |

**Action:** select production with least cost

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

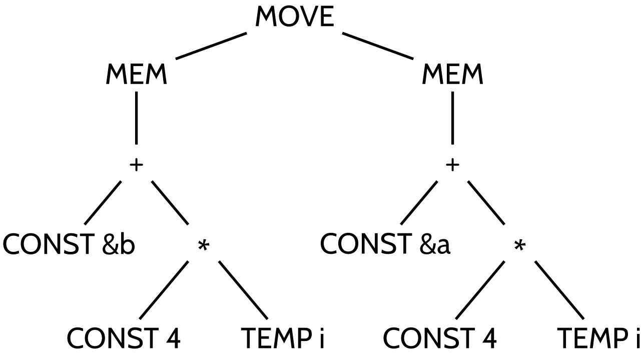
# Computing costs on example



**Action:** done

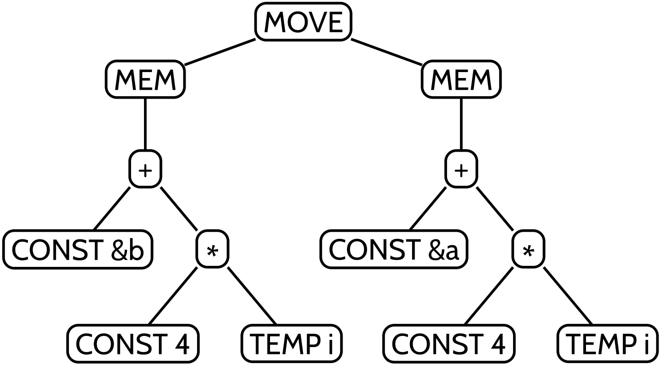
- $T : c$ denotes “reducible to nonterminal $T$at cost $c$”

# Running dynamic programming on our IR tree



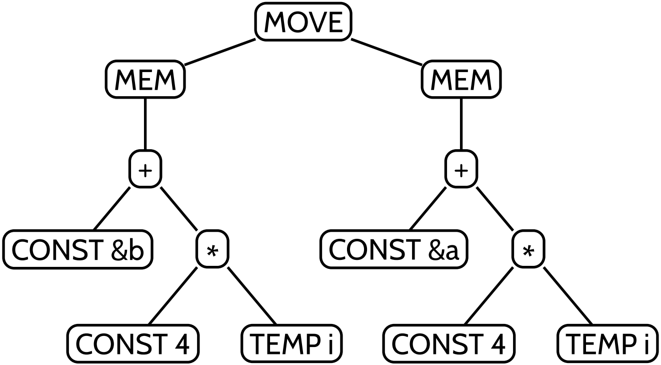
**Action:**

# Running dynamic programming on our IR tree



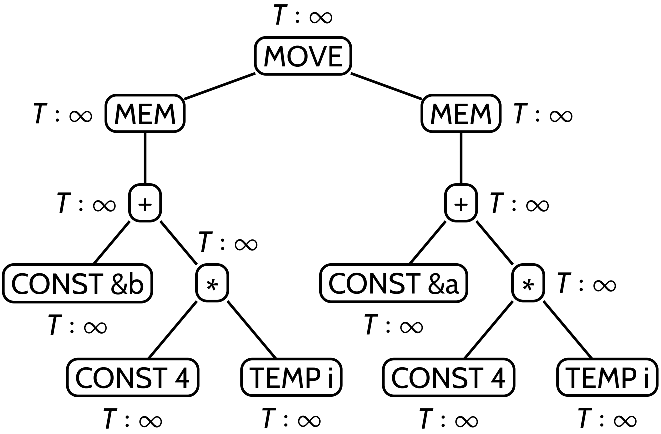
**Action:** added boxes for better readability

# Running dynamic programming on our IR tree



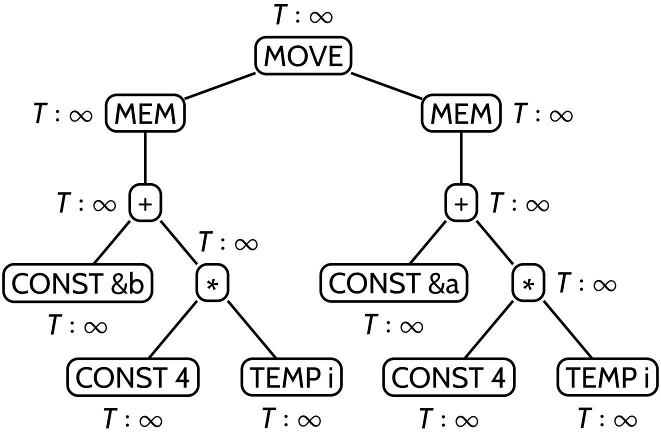
**Action:** assume tile matches already found

# Running dynamic programming on our IR tree



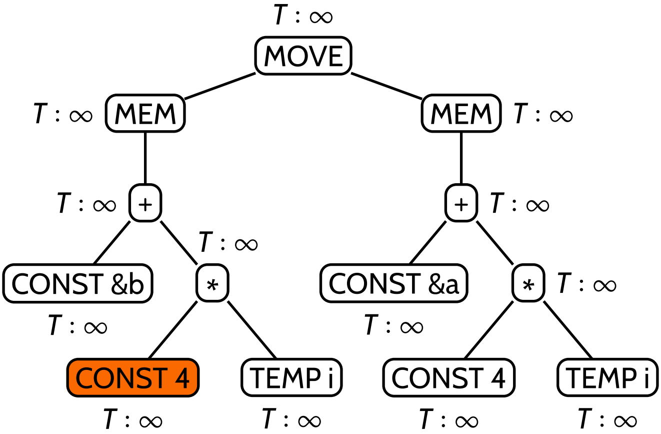
**Action:** initialize reduction costs

# Running dynamic programming on our IR tree



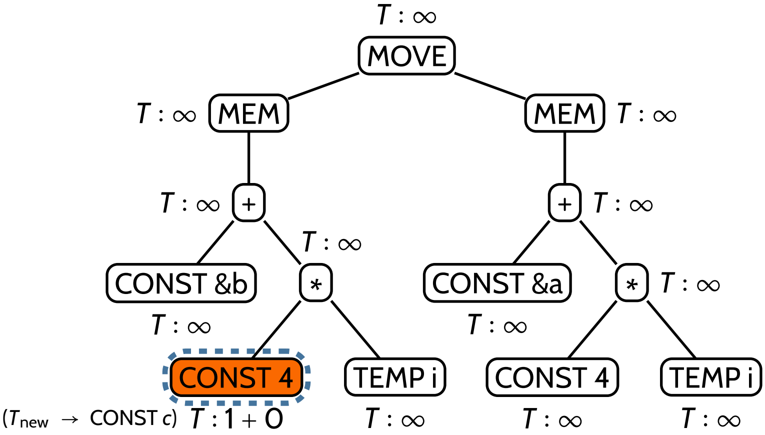
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree



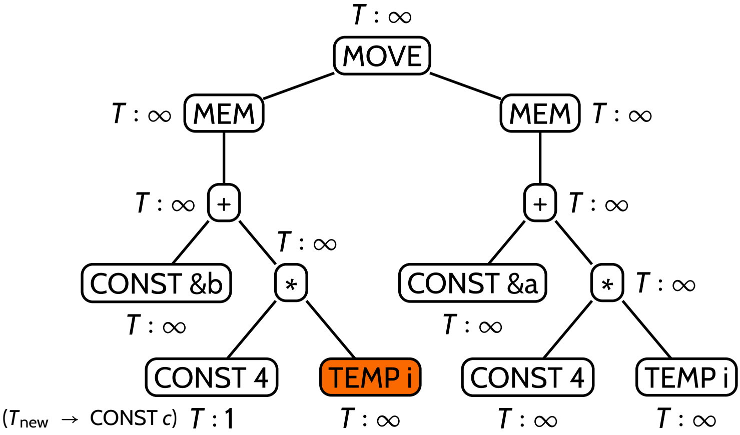
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree



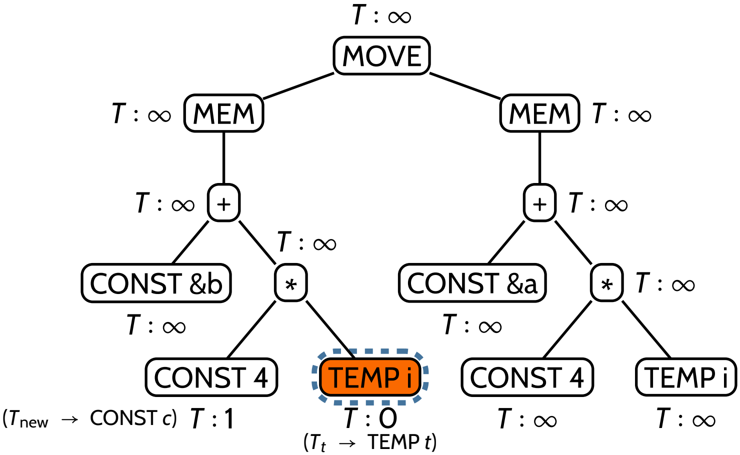
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree



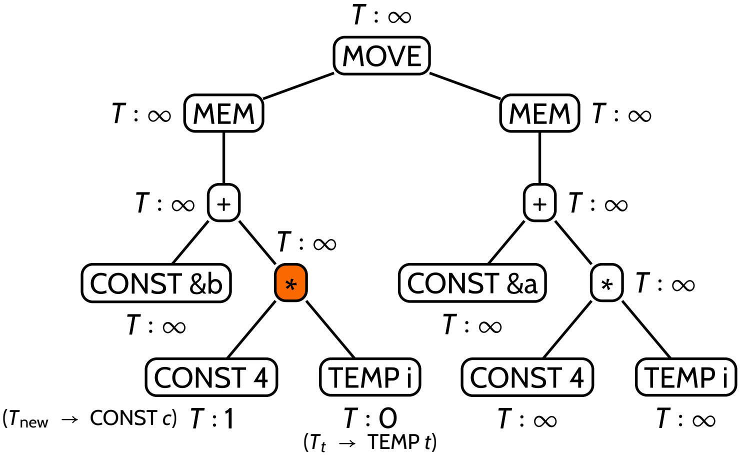
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree



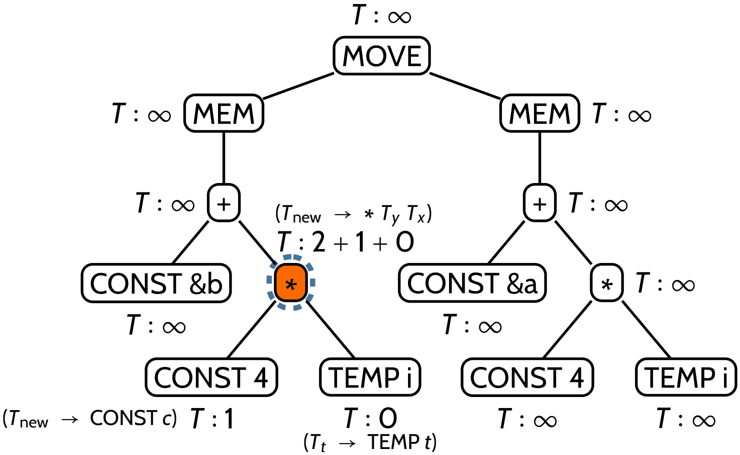
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree



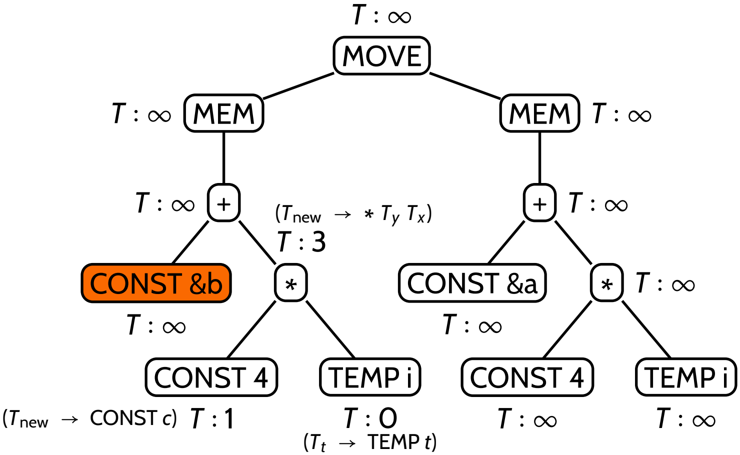
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree



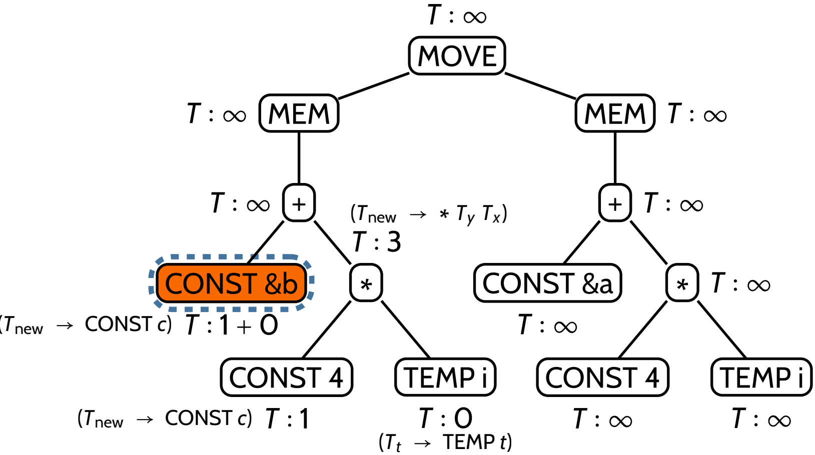
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree



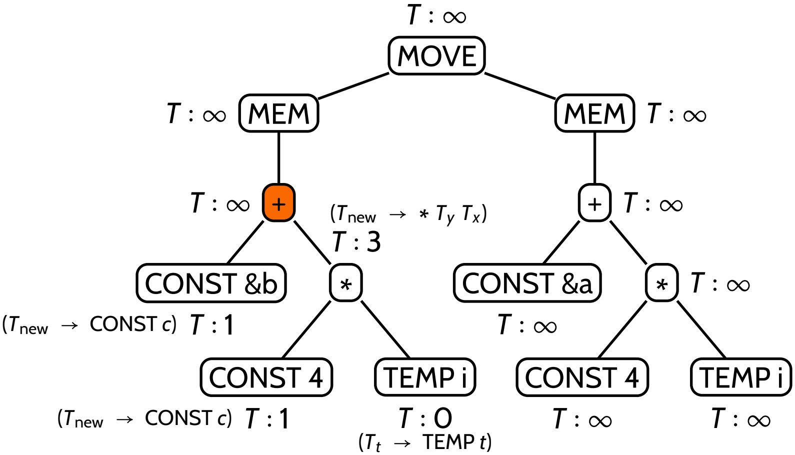
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree



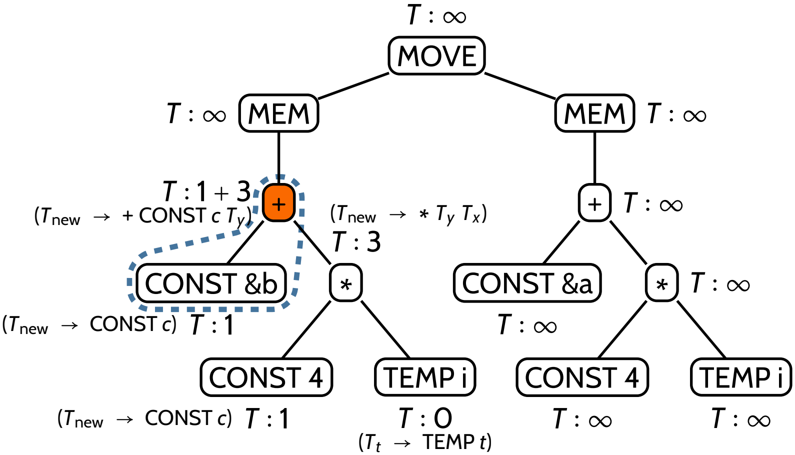
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree


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

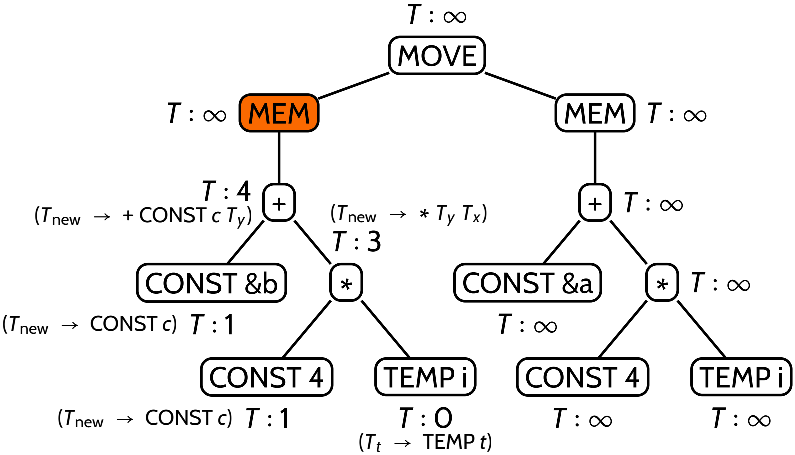
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree


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

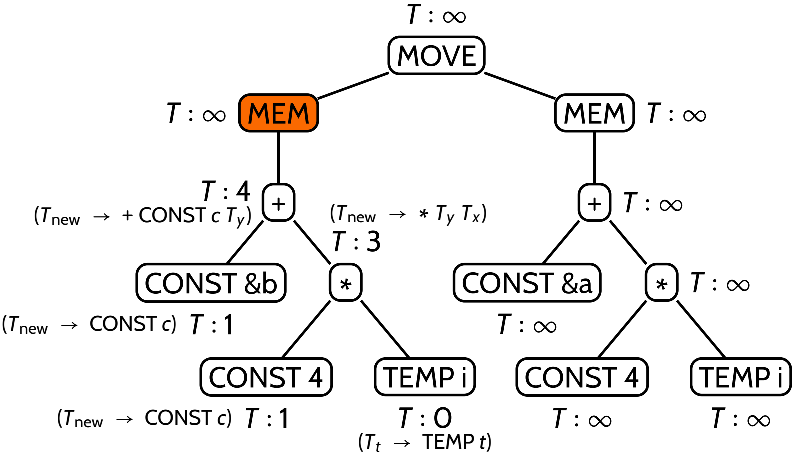
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree


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

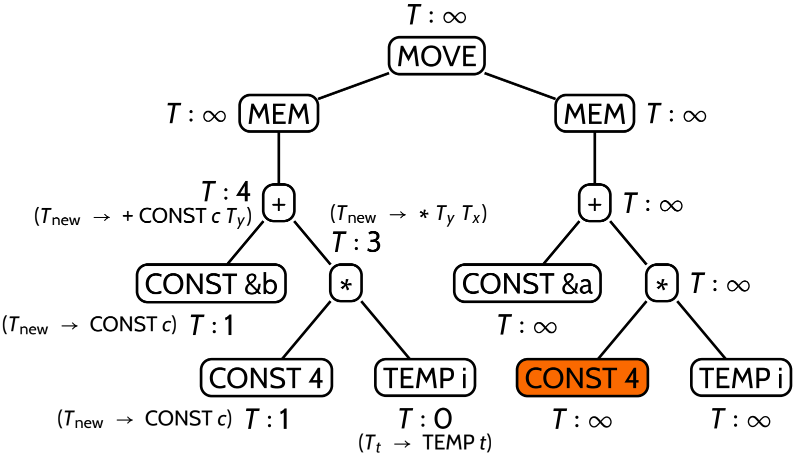
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree


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

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

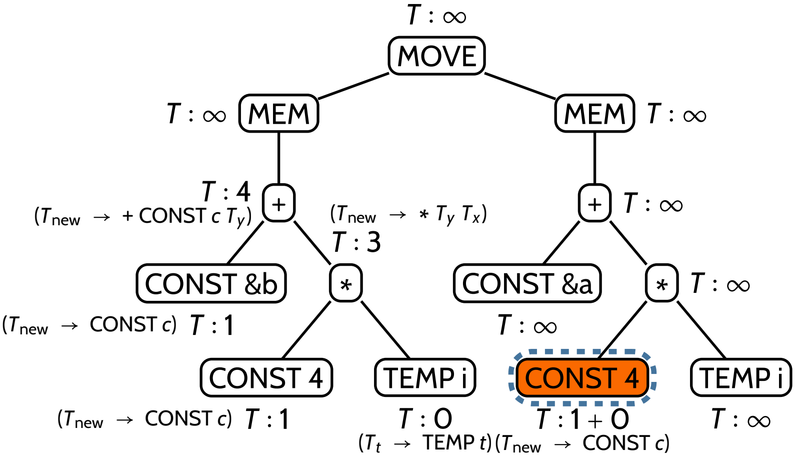
# Running dynamic programming on our IR tree



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

**Action:** compute least reduction costs

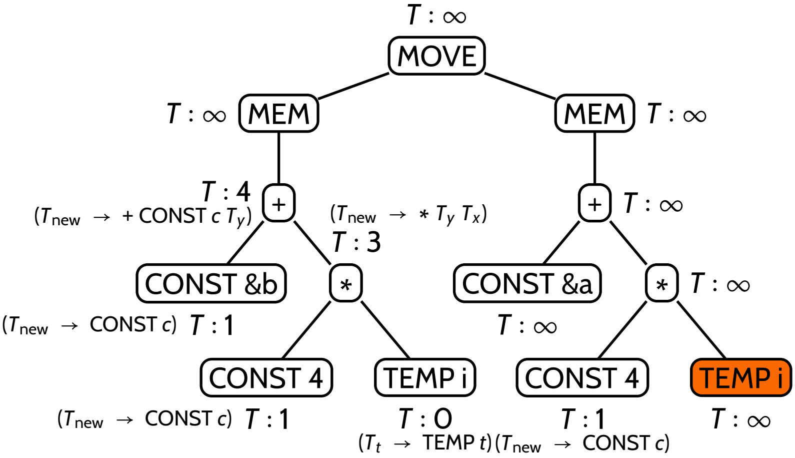
# Running dynamic programming on our IR tree



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

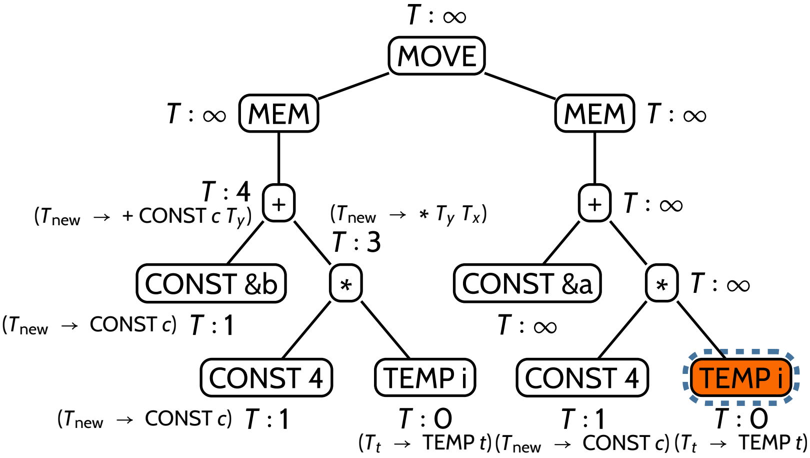
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree


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

**Action:** compute least reduction costs

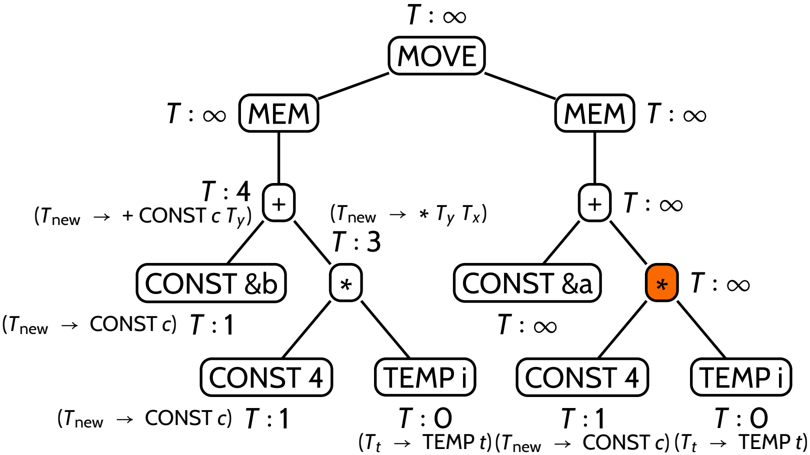
# Running dynamic programming on our IR tree



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

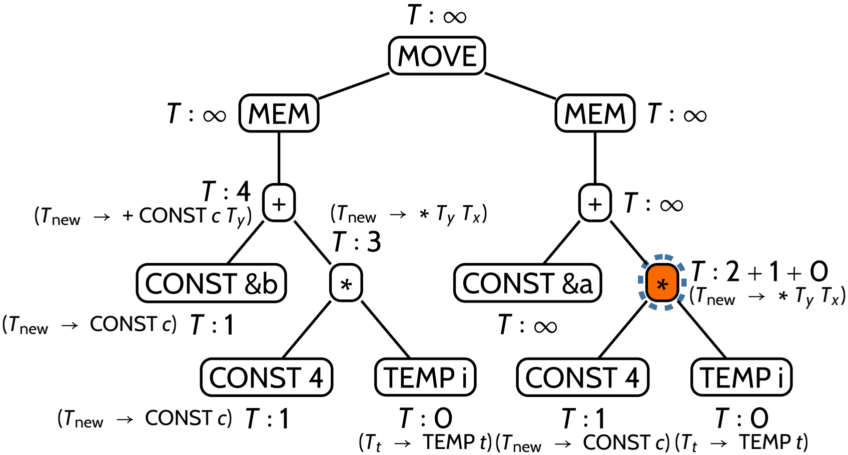
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree


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

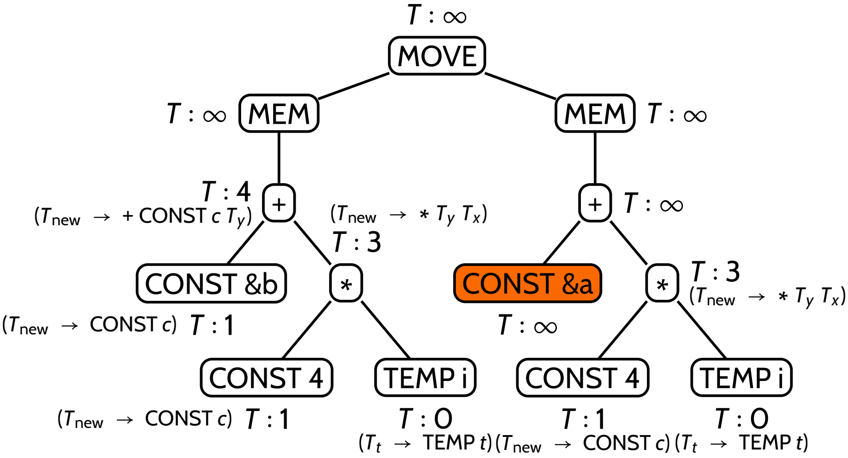
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree


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

**Action:** compute least reduction costs

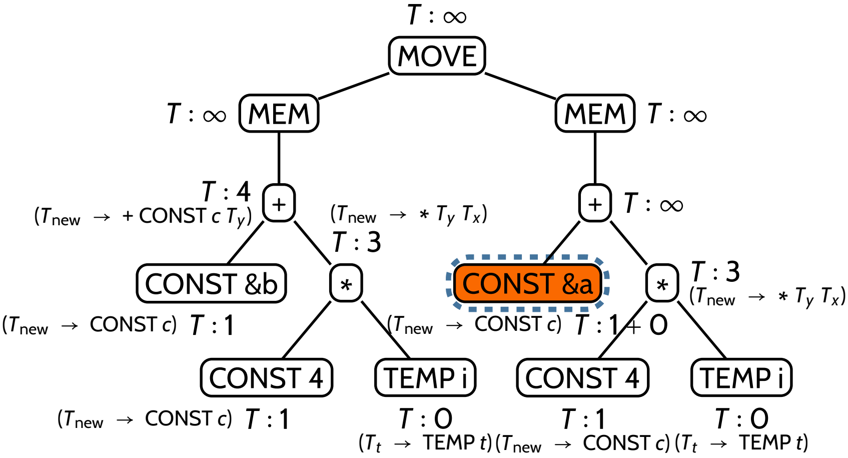
# Running dynamic programming on our IR tree



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

**Action:** compute least reduction costs

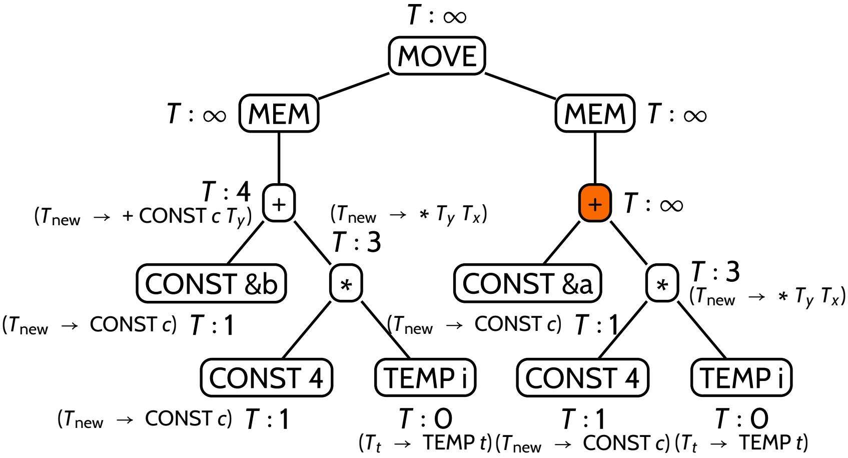
# Running dynamic programming on our IR tree



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

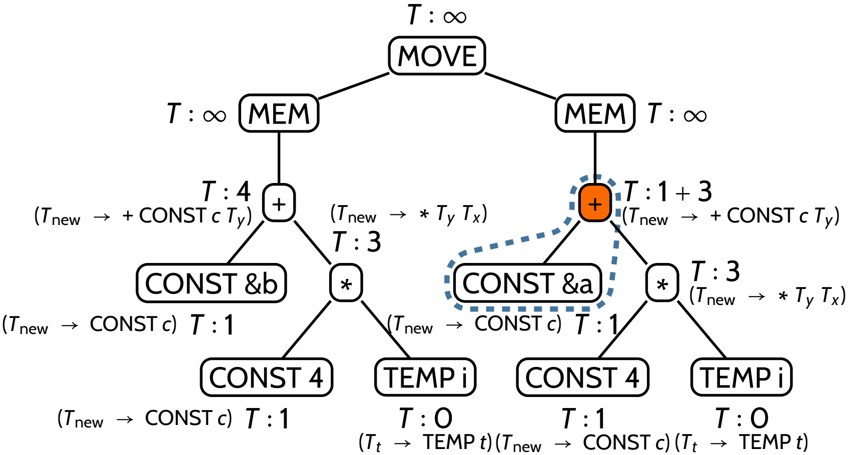
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree


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

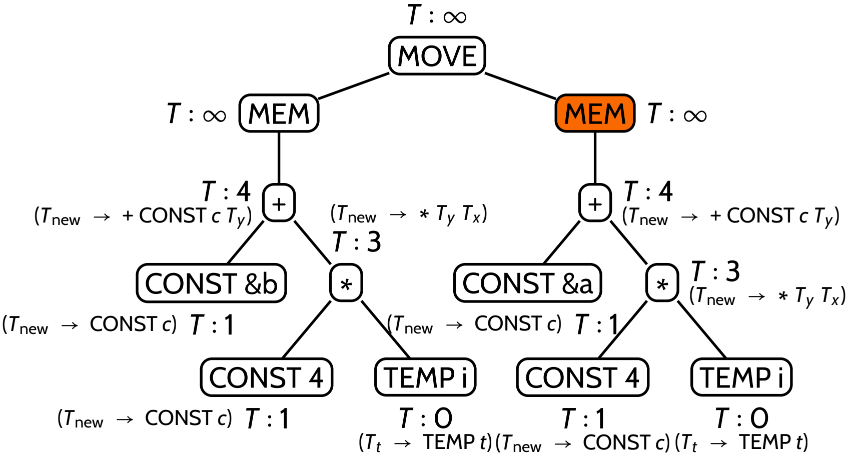
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree


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

**Action:** compute least reduction costs

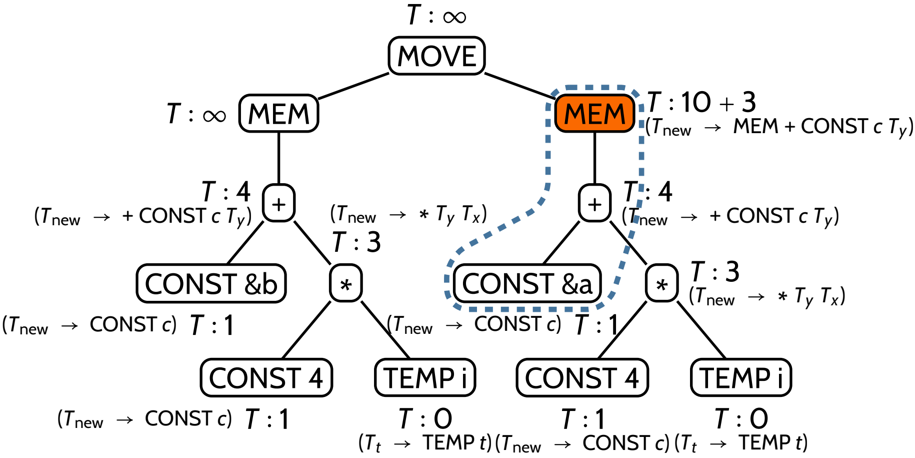
# Running dynamic programming on our IR tree



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

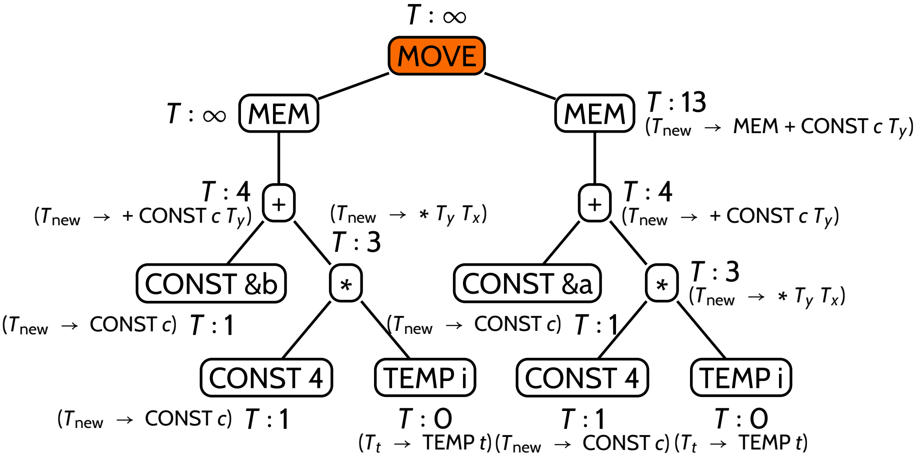
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree


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

**Action:** compute least reduction costs

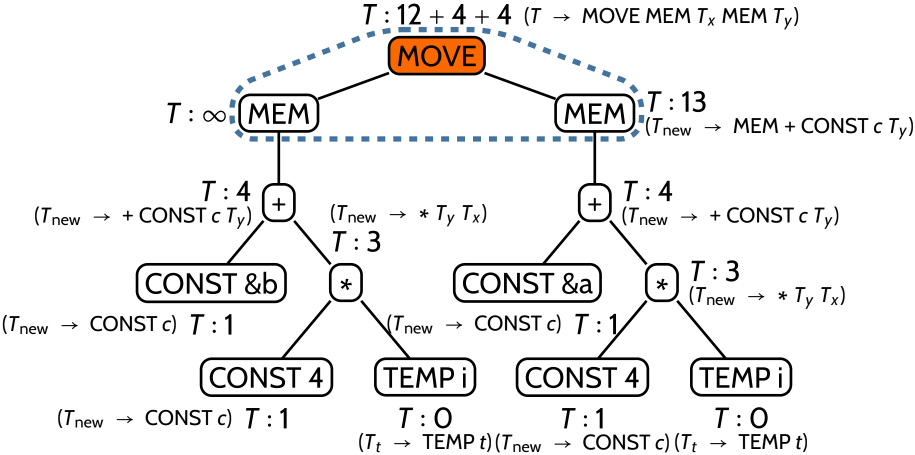
# Running dynamic programming on our IR tree



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

**Action:** compute least reduction costs

# Running dynamic programming on our IR tree



$T : 12 + 4 + 4$ $(T \rightarrow \text{MOVE MEM } T_x \text{ MEM } T_y)$

MOVE

$T : \infty$ MEM

MEM $T : 13$ $(T_{\text{new}} \rightarrow \text{MEM + CONST } c\ T_y)$

$T : 4$ $(T_{\text{new}} \rightarrow \text{+ CONST } c\ T_y)$ +

$(T_{\text{new}} \rightarrow * \ T_y\ T_x)$ $T : 3$

+ $T : 4$ $(T_{\text{new}} \rightarrow \text{+ CONST } c\ T_y)$

CONST &b

$*$

CONST &a

$*$ $T : 3$ $(T_{\text{new}} \rightarrow * \ T_y\ T_x)$

$(T_{\text{new}} \rightarrow \text{CONST } c)$ $T : 1$

$(T_{\text{new}} \rightarrow \text{CONST } c)$ $T : 1$

CONST 4

TEMP i

CONST 4

TEMP i

$(T_{\text{new}} \rightarrow \text{CONST } c)$ $T : 1$

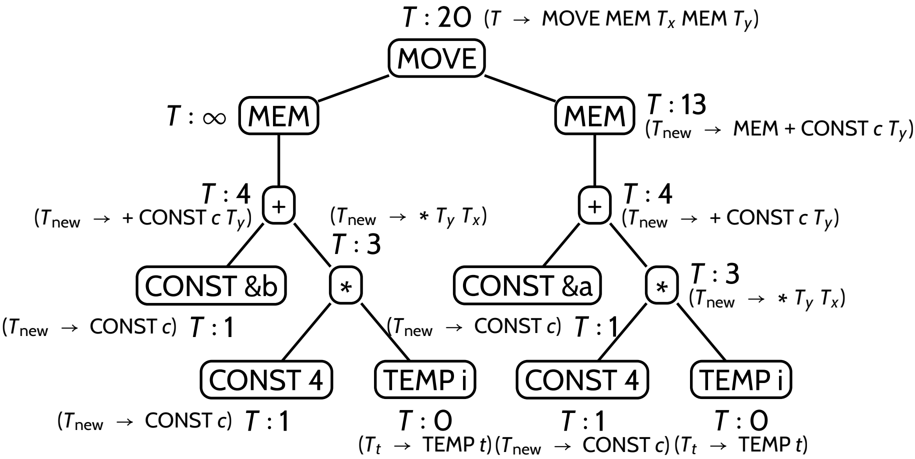
$T : 0$ $(T_t \rightarrow \text{TEMP } t)$

$T : 1$ $(T_{\text{new}} \rightarrow \text{CONST } c)$

$T : 0$ $(T_t \rightarrow \text{TEMP } t)$

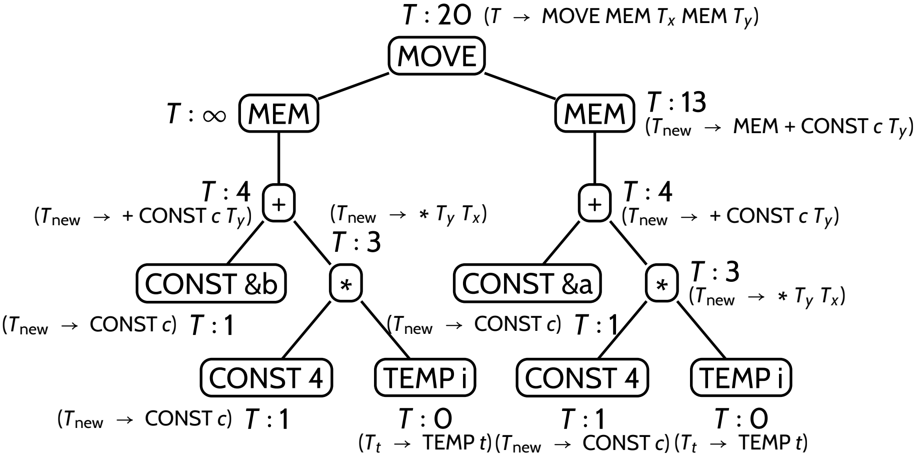
**Action:** compute least reduction costs

# Running dynamic programming on our IR tree


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

**Action:** done computing costs

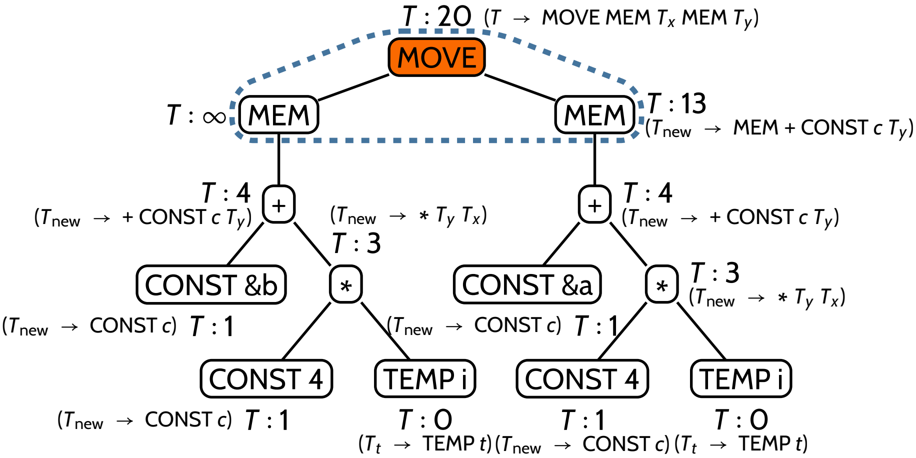
# Running dynamic programming on our IR tree



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

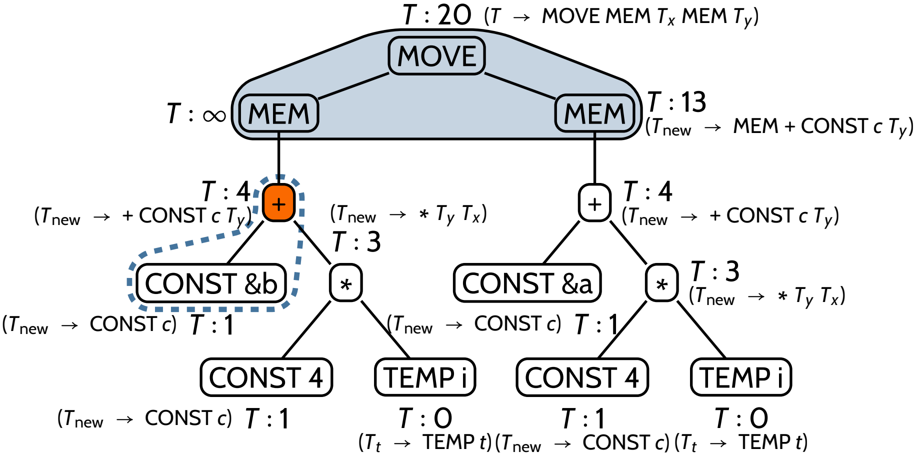
**Action:** select productions

# Running dynamic programming on our IR tree


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

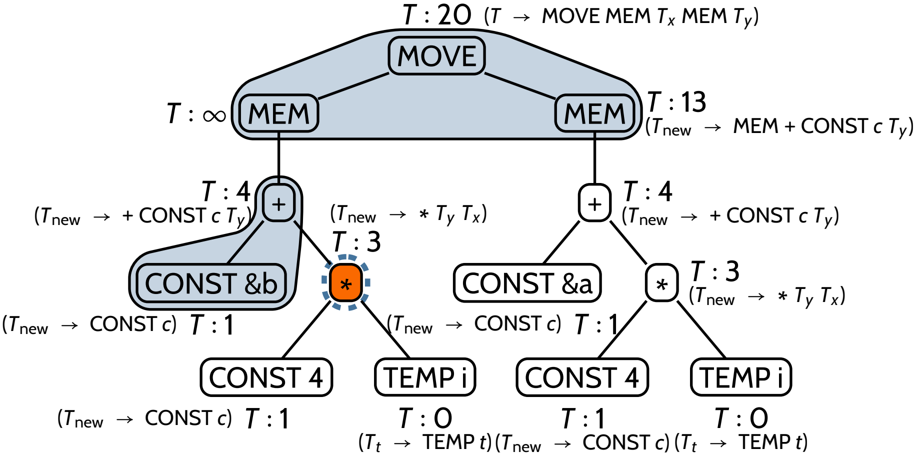
**Action:** select productions

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

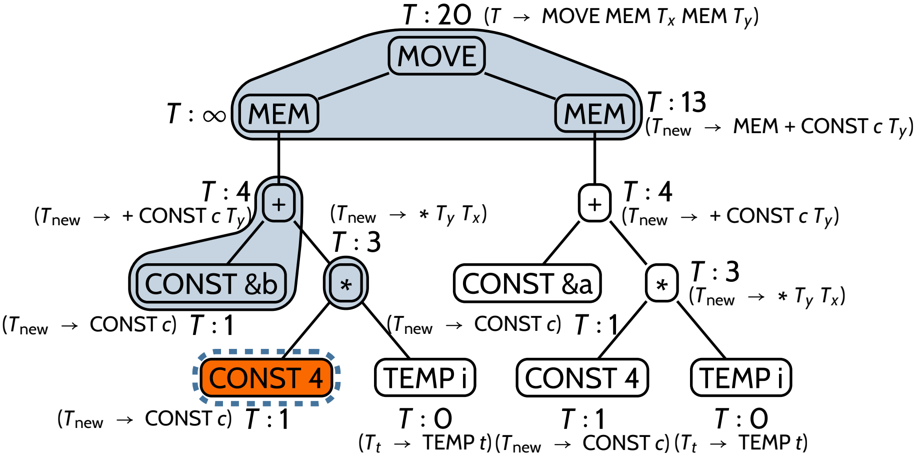
**Action:** select productions

# Running dynamic programming on our IR tree


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

**Action:** select productions

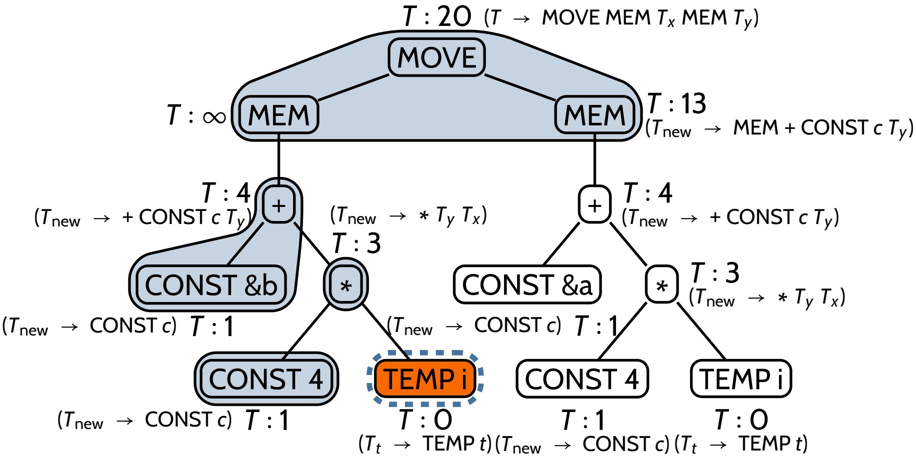
# Running dynamic programming on our IR tree



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

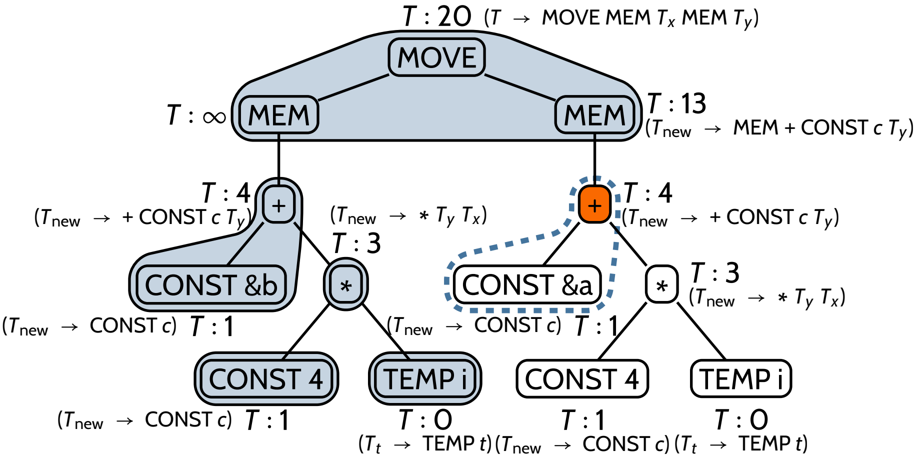
**Action:** select productions

# Running dynamic programming on our IR tree


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

**Action:** select productions

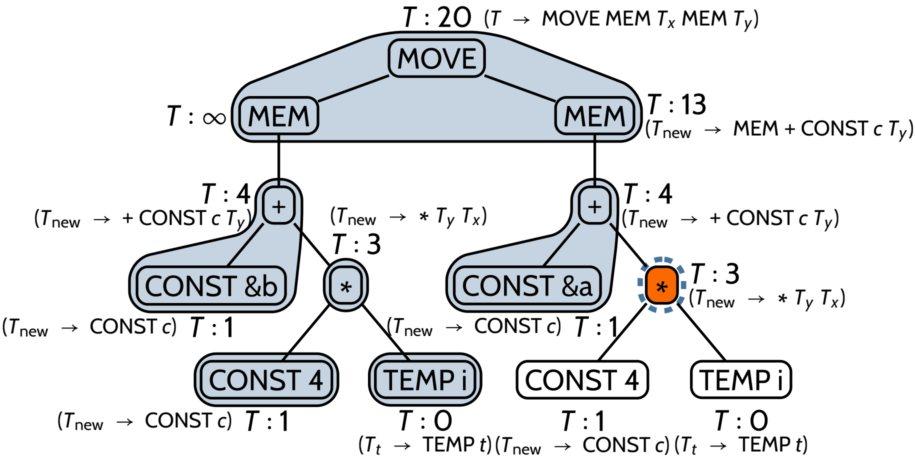
# Running dynamic programming on our IR tree



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

**Action:** select productions

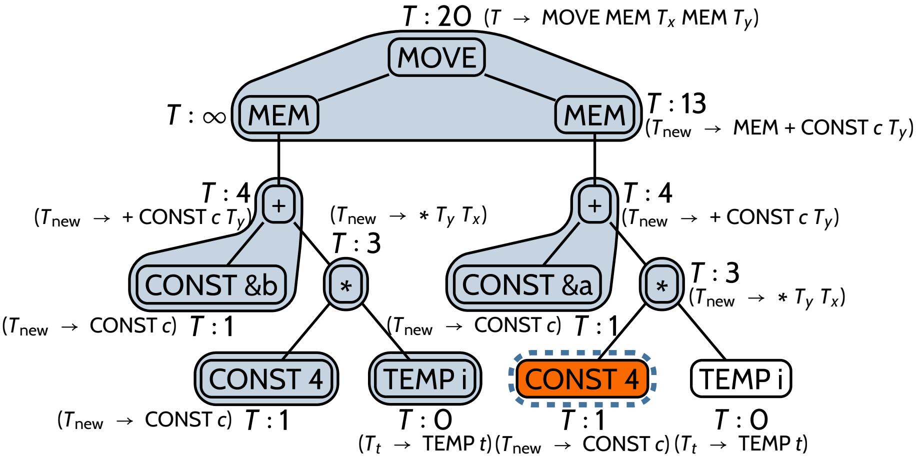
# Running dynamic programming on our IR tree



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

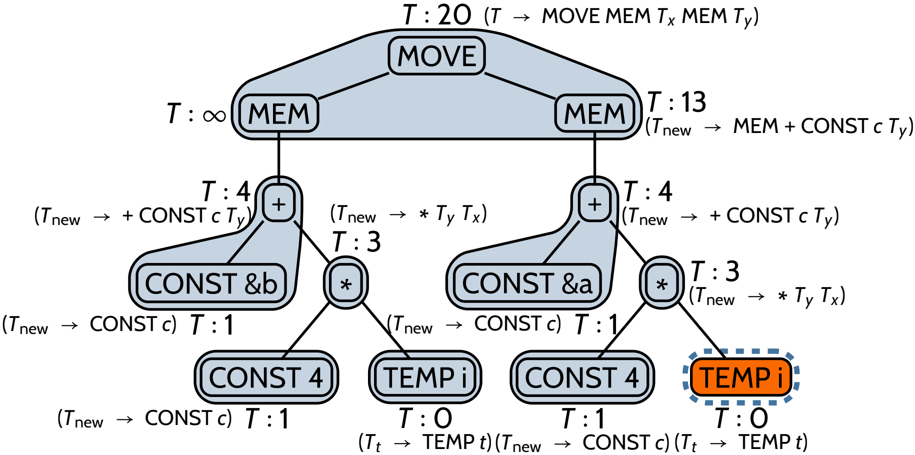
**Action:** select productions

# Running dynamic programming on our IR tree


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

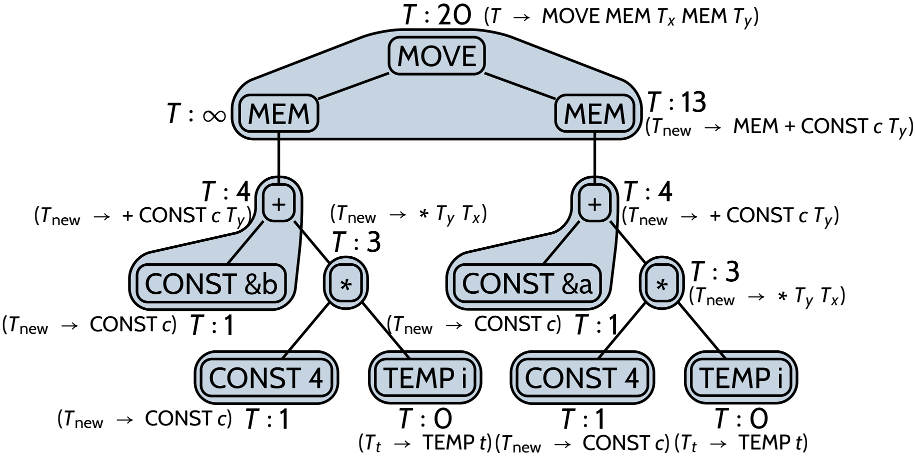
**Action:** select productions

# Running dynamic programming on our IR tree


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

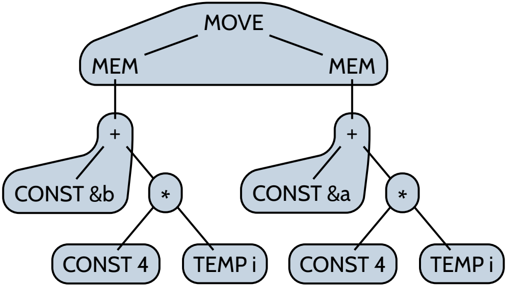
**Action:** select productions

# Running dynamic programming on our IR tree


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

**Action:** done selecting productions

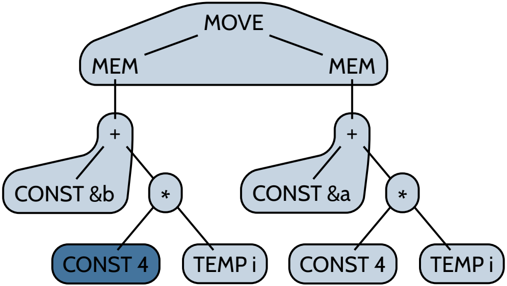
# Running dynamic programming on our IR tree



**Assembly code:**

**Action:** emit assembly instructions

# Running dynamic programming on our IR tree

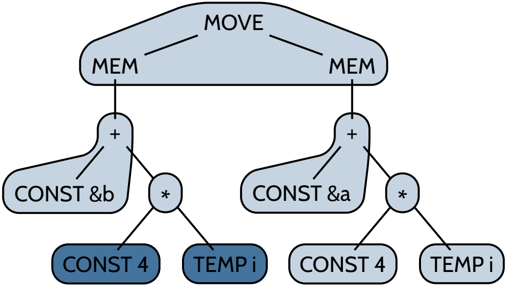


**Assembly code:**

ADDI $t_0 \leftarrow r_0 + \#4$

**Action:** emit assembly instructions

# Running dynamic programming on our IR tree

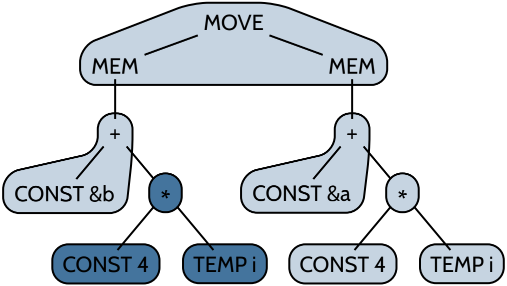


**Assembly code:**

ADDI $t_0 \leftarrow r_0 + \#4$

**Action:** emit assembly instructions

# Running dynamic programming on our IR tree

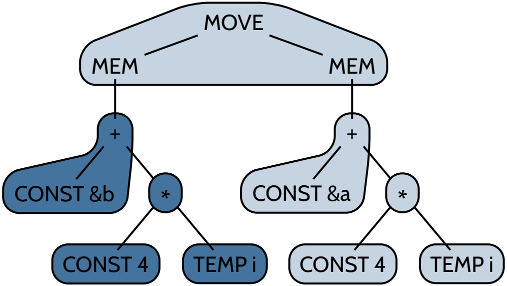


**Assembly code:**

| | |
|---|---|
| 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

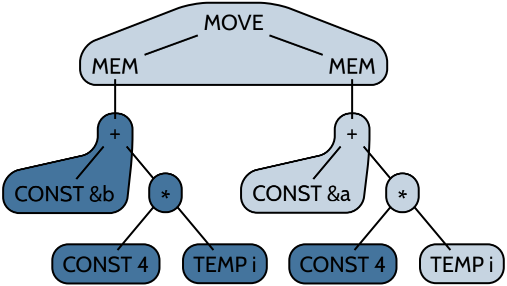


**Assembly code:**

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

**Action:** emit assembly instructions

# Running dynamic programming on our IR tree

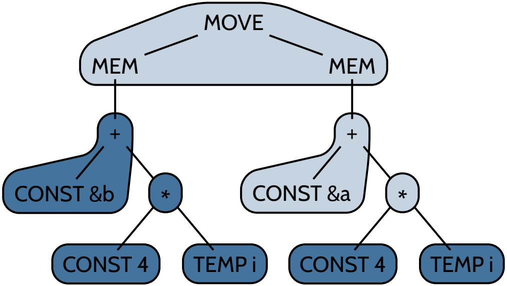


**Assembly code:**

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

**Action:** emit assembly instructions

# Running dynamic programming on our IR tree

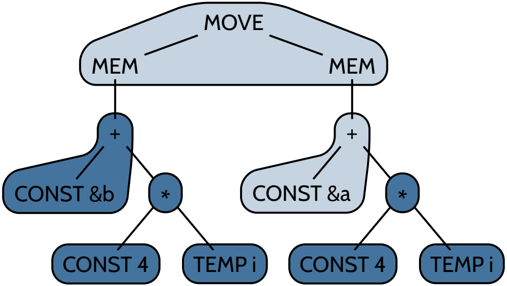


**Assembly code:**

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

**Action:** emit assembly instructions

# Running dynamic programming on our IR tree

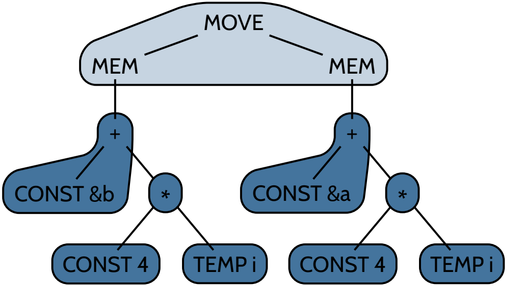


**Assembly code:**

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

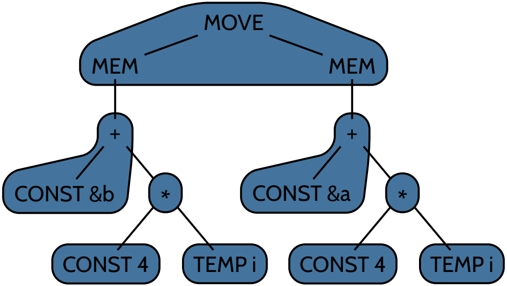


**Assembly code:**

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

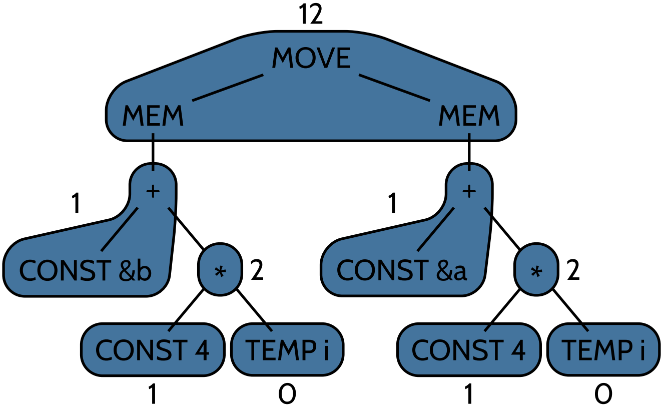


**Assembly code:**

ADDI $t_0 \leftarrow r_0 + \#4$<br>
MUL $t_1 \leftarrow t_0 * t_i$<br>
ADDI $t_2 \leftarrow t_1 + \#4$<br>
ADDI $t_3 \leftarrow r_0 + \#4$<br>
MUL $t_4 \leftarrow t_3 * t_i$<br>
ADDI $t_5 \leftarrow t_4 + \#4$<br>
MOVEM $M[t_2] \leftarrow M[t_5]$

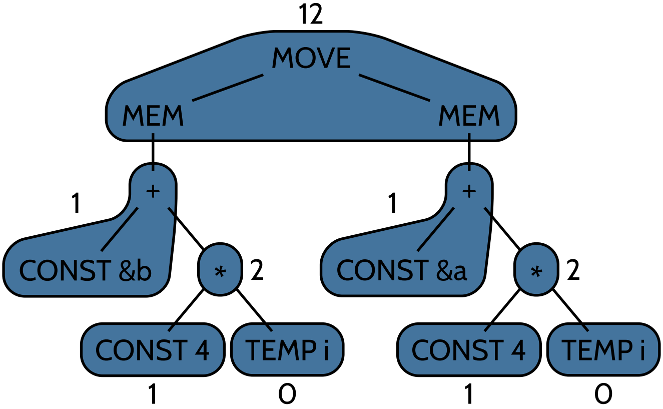
**Action:** done

# Optimum tiling found with dynamic prog.



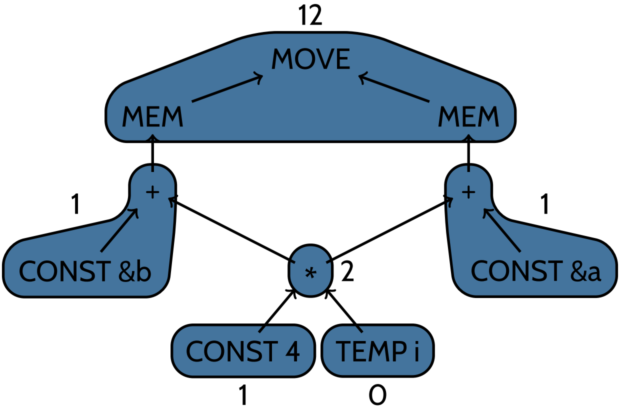
$$\sum \text{cost} = 20$$

# Can we do better?



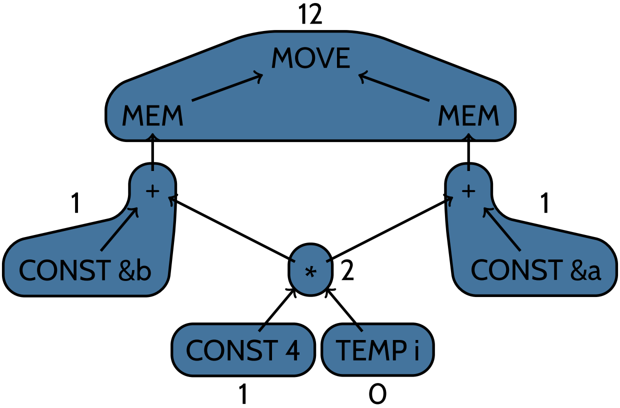
$\sum$ cost = 20

## **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*

# **Summary:** Maximum munch

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

# **Summary:** Tree parsing

- **Advantages over maximum munch:**
  - Instructions modeled as **tree grammar**
    - **Retargetable** (instruction selector is generated)
    - Supports more complicated target architectures (via additional nonterminals)
    - Can be automatically verified
  - **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

# **Summary:** Dynamic programming

- **Advantages over tree parsing:**
  - Always yields **optimum** tilings
  - **Very fast** $\mathcal{O}(n)$
- **Disadvantages:**
  - Requires **IR trees** as input
- **Suitable for:**
  - Target architectures where all instructions are modeled as tree tiles
- **Modern implementations:**
  - *CoSy*
  - Burg $\xrightarrow{\textbf{“Burger phenomenon”}}$ Dburg, Gburg, GPburg, Iburg, Jburg, Hburg, Lburg, Mburg, OCamlBurg, and Wburg

# Further reading

- **Technical report:**
  - Gabriel Hjort Blindell – *Survey on Instruction Selection: An Extensive and Modern Literature Review* (2013) `http://www.diva-portal.org/smash/record.jsf?pid=diva2:653943`
  - PDF downloadable for free (Yaay!)
  - Also soon available as book