Target Code Generation

Exp : a = b*-c + b*-c

- KNOW YOUR LANGAUGE's Opcode : e.g. ADD, SUB, MULT, DIV, MOV
- **St Type** : Opcode Source, Destination
- Example : **MOV a, R0** (Move value in a to Register R0)
- For a TAC arithmetic expression **y op z** , move the left operand (**y**) to an available GP register and apply right operand (**z**) on it. i.e.

  MOV y, Rn

  Op z, Rn
- The Assembly language can have various General Purpose Registers (R0 – Rn), along with some special registers like Accumulator, Flag etc.
- When there are no available registers then we have to use the memory for completing the assembly operation.

TAC	Assembly (R0-R4)
t1 = -c (R0)	MOV c,R0 UMINUS R0
t2 = b*t1 (R1)	MOV b, R1 MULT R0, R1
t3 = -c (R0)	MOV c,R0 UMINUS R0
t4 = b*t3 (R2)	MOV b, R2 MULT R0, R2
t5 = t2+t4 (R2)	ADD R1, R2
a = t5	MOV R2,a

- Instruction Cost
- Register : 0
- Memory : 1
- Opcode : 1

TAC	Optimal Assembly (R0-R4)	Instr. Cost
t1 = -c (R0)	MOV c,R0 UMINUS R0	1+1+0 = 2 1 + 0 = 1 **(3)**
t2 = b*t1 (R1)	MOV b, R1 MULT R0, R1	1+1+0 = 2 1+0+0 = 1 **(3)**
t3 = -c (R0)	MOV c,R0 UMINUS R0	1+1+0 = 2 1 + 0 = 1 **(3)**
t4 = b*t3 (R2)	MOV b, R2 MULT R0, R2	1+1+0 = 2 1+0+0 = 1 **(3)**
t5 = t2+t4 (R2)	ADD R1, R2	1+0+0 = 1 **(1)**
a = t5	MOV R2,a	1+0+1 = 2 **(2)**
	(10) Net Instr Cost	**15**

Sub-optimal Assembly (Used R0 Only even when we had R0-R4)	Instr. Cost

MOV c,R0
UMINUS R0
MOV R0,t1
	1+1+0 = 2
	1 + 0 = 1
	1+0+1 = 2 **(5)**

MOV b, R0
MULT t1, R0
MOV R0,t2
	1+1+0 = 2
	1+1+0 = 2
	1+0+1 = 2 **(6)**

MOV c,R0
UMINUS R0
MOV R0,t3
	1+1+0 = 2
	1 + 0 = 1
	1+0+1 = 2 **(5)**

MOV b, R0
MULT t3, R0
MOV R0,t4
	1+1+0 = 2
	1+1+0 = 2
	1+0+1 = 2 **(6)**

MOV t2, R0
ADD t4, R0
MOV R0,t5
	1+1+0 = 2
	1+1+0 = 2
	1+1+0 = 2 **(6)**

MOV R2,a	1+0+1 = 2 **(2)**
(16) Net Instr Cost	**30**

Code Generation

- The target machine
- Runtime environment
- Basic blocks and flow graphs
- Instruction selection
- Instruction selector generator
- Register allocation
- Peephole optimization

The Target Machine

- A byte addressable machine with four bytes to a word and *n* general purpose registers
- Two address instructions
  - *op source, destination*
- Six addressing modes

– absolute	M	M	1
– register	R	R	0
– indexed	c(R)	c+content(R)	1
– ind register	*R	content(R)	0
– ind indexed	*c(R)	content(c+content(R))	1
– literal	#c	c	1

Examples

```
MOV     R0,   M
MOV     4 (R0),  M
MOV     *R0,  M
MOV     *4 (R0),  M
MOV     #1,  R0
```

Instruction Costs

- Cost of an instruction = 1 + costs of source and destination addressing modes
- This cost corresponds to the length (in words) of the instruction
- Minimize instruction length also tend to minimize the instruction execution time

Examples

MOV	R0, R1	1
MOV	R0, M	2
MOV	#1, R0	2
MOV	4 (R0), *12 (R1)	3

An Example

Consider a := b + c

1. MOV b, R0
   ADD c, R0
   MOV R0, a

2. MOV b, a
   ADD c, a

3. R0, R1, R2 contains the addresses of a, b, c
   MOV *R1, *R0
   ADD *R2, *R0

4. R1, R2 contains the values of b, c
   ADD R2, R1
   MOV R1, a

Instruction Selection

- Code skeleton

```
x := y + z          a := b + c          d := a + e
MOV   y, R0         MOV   b, R0         MOV   a, R0
ADD   z, R0         ADD   c, R0         ADD   e, R0
MOV   R0, x         MOV   R0, a         MOV   R0, d
```

- Multiple choices

```
a := a + 1          MOV   a, R0         INC   a
                    ADD   #1, R0
                    MOV   R0, a
```

Register Allocation

- Register allocation: select the set of variables that will reside in registers
- Register assignment: pick the specific register that a variable will reside in
- The problem is NP-complete

An Example

```
t := a + b
t := t * c
t := t / d
```

```
MOV  a, R0
ADD  b, R0
MUL  c, R0
DIV  d, R0
MOV  R1, t
```

```
t := a + b
t := t + c
t := t / d
```

```
MOV  a, R0
ADD  b, R0
ADD  c, R0
SRDA R0, 32
DIV  d, R0
MOV  R1, t
```

Basic Blocks

- A *basic block* is a sequence of consecutive statements in which control enters at the beginning and leaves at the end without halt or possibility of branching except at the end

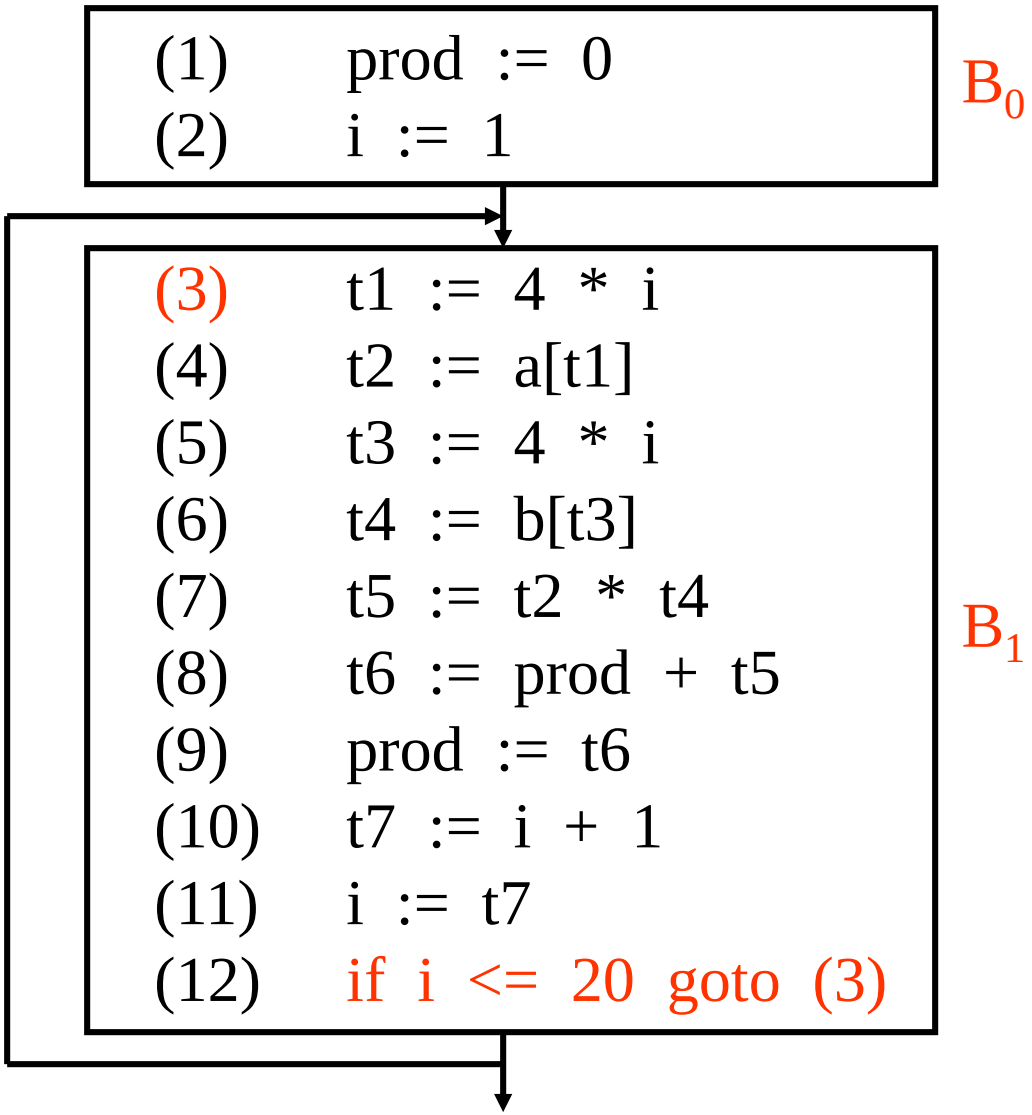
An Example

```
(1)     prod := 0
(2)     i := 1
(3)     t1 := 4 * i
(4)     t2 := a[t1]
(5)     t3 := 4 * i
(6)     t4 := b[t3]
(7)     t5 := t2 * t4
(8)     t6 := prod + t5
(9)     prod := t6
(10)    t7 := i + 1
(11)    i := t7
(12)    if i <= 20 goto (3)
```

Control Flow Graphs

- A (*control*) *flow graph* is a directed graph
- The *nodes* in the graph are *basic blocks*
- There is an *edge* from B_1 to B_2 iff B_2 immediately follows B_1 in some execution sequence
  - there is a jump from B_1 to B_2
  - B_2 immediately follows B_1 in program text
- B_1 is a *predecessor* of B_2, B_2 is a *successor* of B_1

An Example


(1) prod := 0
(2) i := 1
B0
(3) t1 := 4 * i
(4) t2 := a[t1]
(5) t3 := 4 * i
(6) t4 := b[t3]
(7) t5 := t2 * t4
(8) t6 := prod + t5
(9) prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i <= 20 goto (3)
B1

Construction of Basic Blocks

- Determine the set of *leaders*
  - For ethe first statement is a leader
  - the target of a jump is a leader
  - any statement immediately following a jump is a leader
- ach leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program

Representation of Basic Blocks

- Each basic block is represented by a record consisting of
  - a count of the number of statements
  - a pointer to the leader
  - a list of predecessors
  - a list of successors

DAG Representation of Blocks

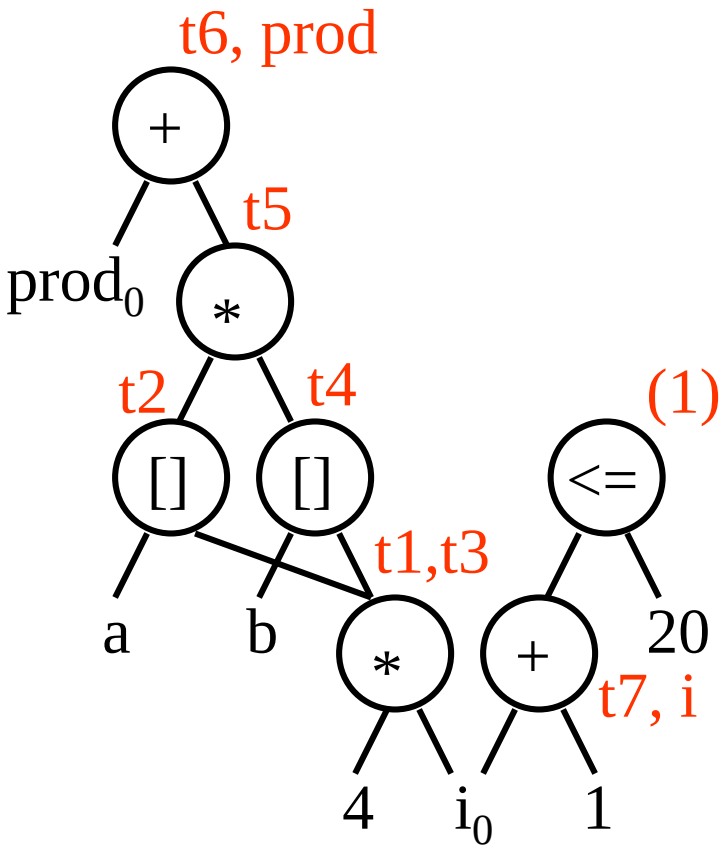
- Easy to determine:
- *common subexpressions*
- *names* used in the block but *evaluated outside* the block
- *names* whose values could be *used outside* the block

DAG Representation of Blocks

- Leaves labeled by *unique identifiers*
- Interior nodes labeled by *operator symbols*
- Interior nodes optionally given a sequence of *identifiers*, having the value represented by the nodes

An Example

(1) t1 := 4 * i
(2) t2 := a[t1]
(3) t3 := 4 * i
(4) t4 := b[t3]
(5) t5 := t2 * t4
(6) t6 := prod + t5
(7) prod := t6
(8) t7 := i + 1
(9) i := t7
(10) if i <= 20 goto (1)



Constructing a DAG

- Consider $x := y$ op z. Other statements can be handled similarly
- If *node*(y) is undefined, create a leaf labeled y and let *node*(y) be this leaf. If *node*(z) is undefined, create a leaf labeled z and let *node*(z) be that leaf

Constructing a DAG

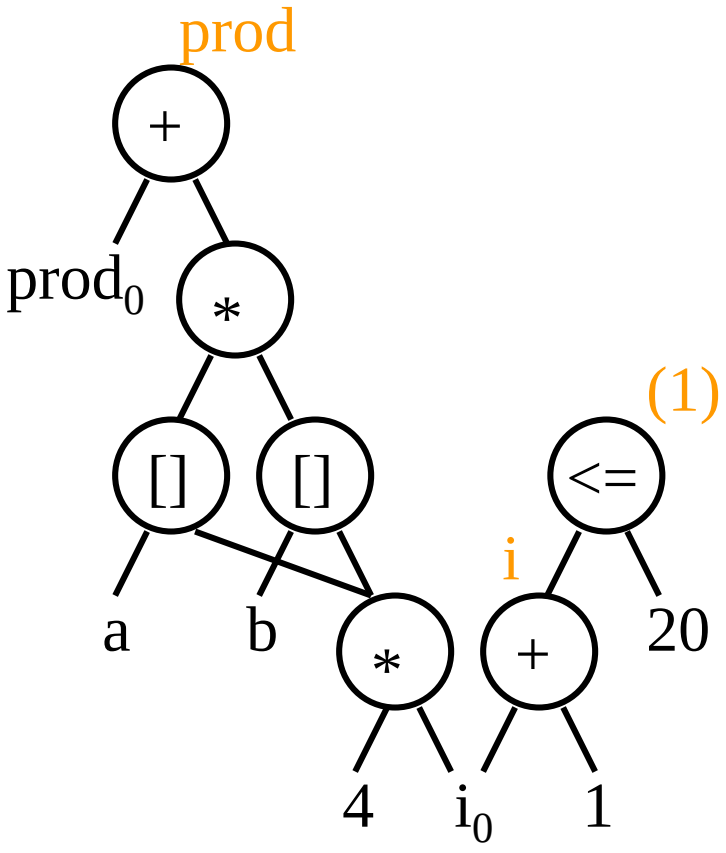
- Determine if there is a node labeled op, whose left child is *node*(*y*) and its right child is *node*(*z*). If not, create such a node. Let *n* be the node found or created.
- Delete *x* from the list of attached identifiers for *node*(*x*). Append *x* to the list of attached identifiers for the node *n* and set *node*(*x*) to *n*

Reconstructing Quadruples

- Evaluate the interior nodes in *topological order*
- Assign the evaluated value to one of its attached identifier *x*, preferring one whose value is needed outside the block
- If there is no attached identifier, create a *new temp* to hold the value
- If there are additional attached identifiers *y*1, *y*2, …, *yk* whose values are also needed outside the block, add

  $y1 := x,\ y2 := x, \ldots,\ yk := x$

An Example

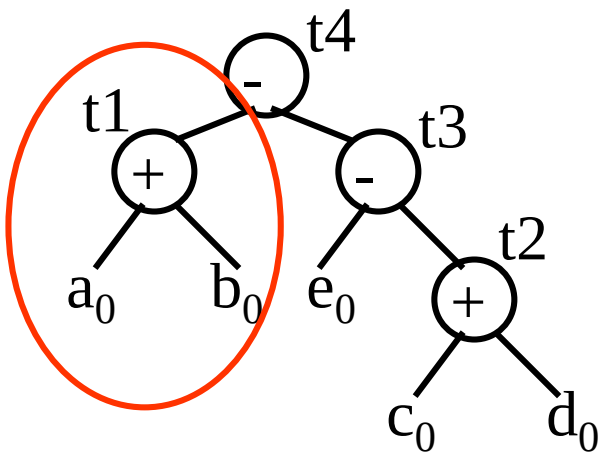


(1) t1 := 4 * i
(2) t2 := a[t1]
(3) t3 := b[t1]
(4) t4 := t2 * t3
(5) prod := prod + t4
(6) i := i + 1
(7) if i <= 20 goto (1)

Generating Code From Tree

(a+b)-(e-(c+d))

t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3

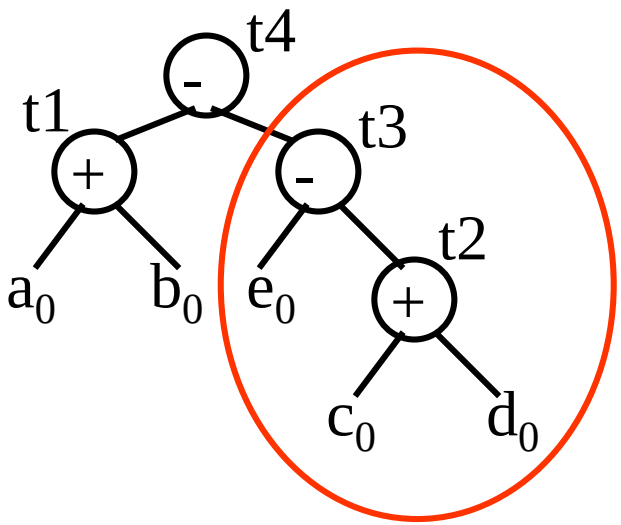


(1)	MOV	a, R0
(2)	ADD	b, R0
(3)	MOV	c, R1
(4)	ADD	d, R1
(5)	MOV	R0, t1
(6)	MOV	e, R0
(7)	SUB	R1, R0
(8)	MOV	t1, R1
(9)	SUB	R0, R1
(10)	MOV	R1, t4

Only R0 and R1 available

Rearranging the Order

t2 := c + d
t3 := e - t2
t1 := a + b
t4 := t1 - t3



(1)	MOV	c, R0
(2)	ADD	d, R0
(3)	MOV	e, R1
(4)	SUB	R0, R1
(5)	MOV	a, R0
(6)	ADD	b, R0
(7)	SUB	R1, R0
(8)	MOV	R0, t4

Code Generation

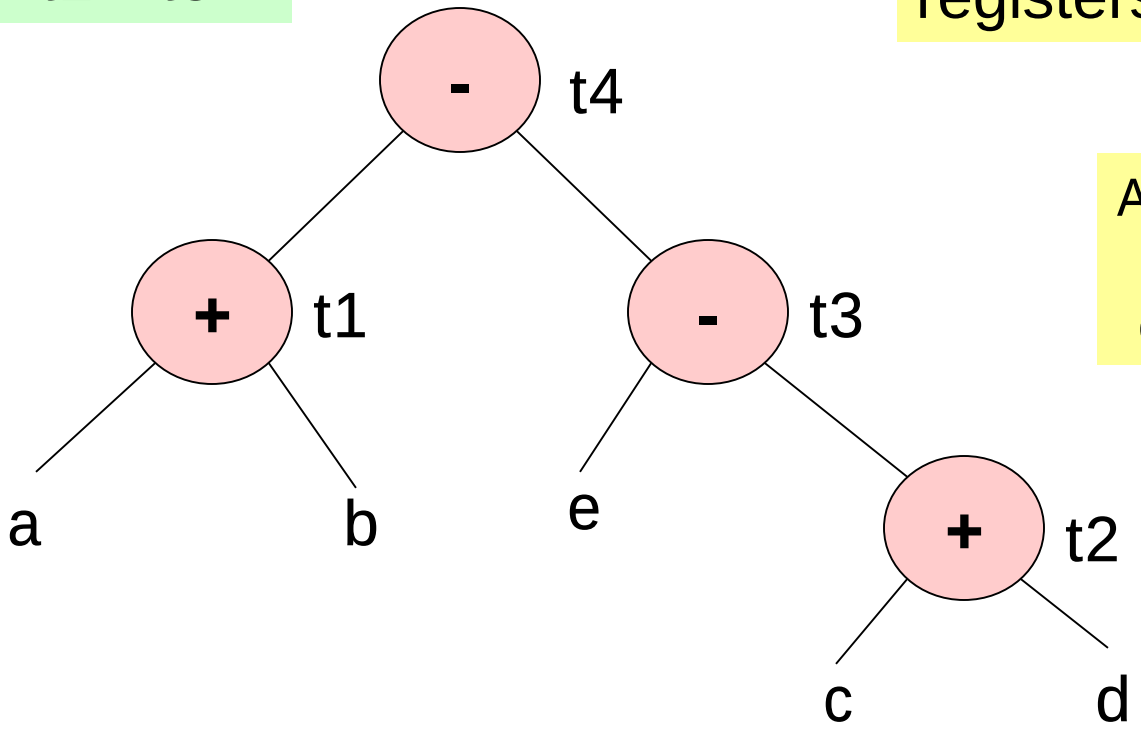
Problem: How to generate optimal code for a basic block specified by its DAG representation?

If the DAG is a tree, we can use Sethi-Ullman algorithm to generate code that is optimal in terms of program length or number of registers used.

Example

```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```

Generate code for a machine with two registers.

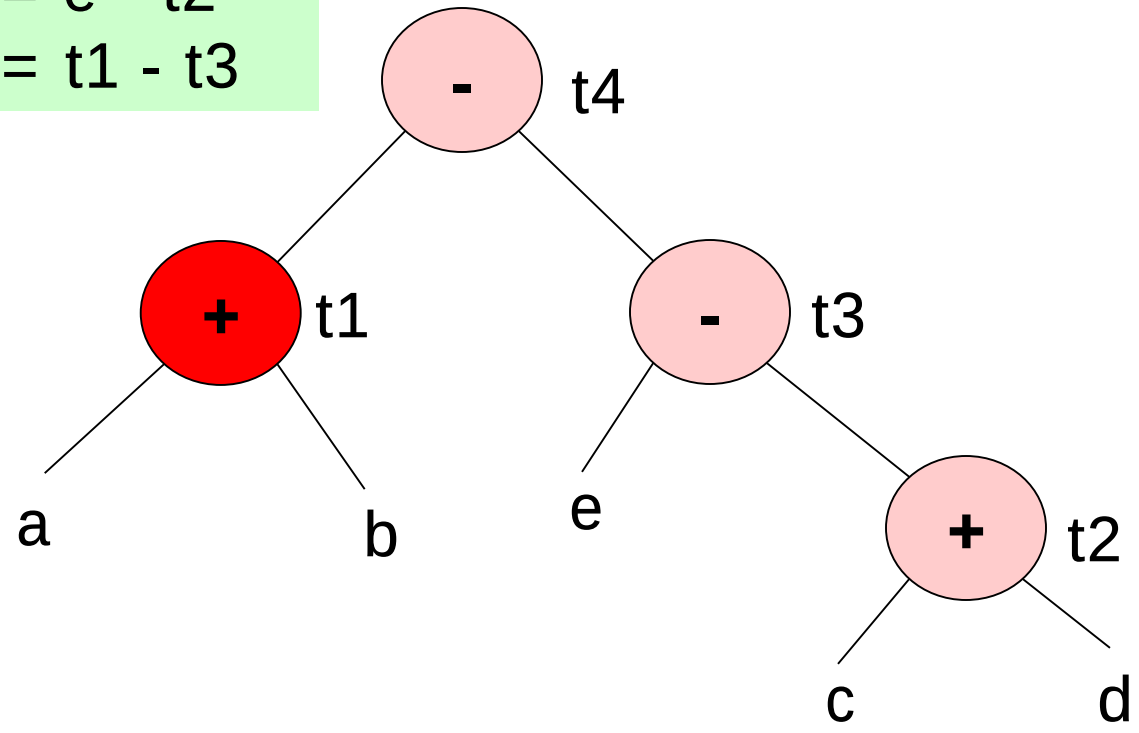


Assume that only t4 is alive at the exit of the basic block.

Example

```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```

```
LOAD   a, R0
ADD    b, R0
```



Example

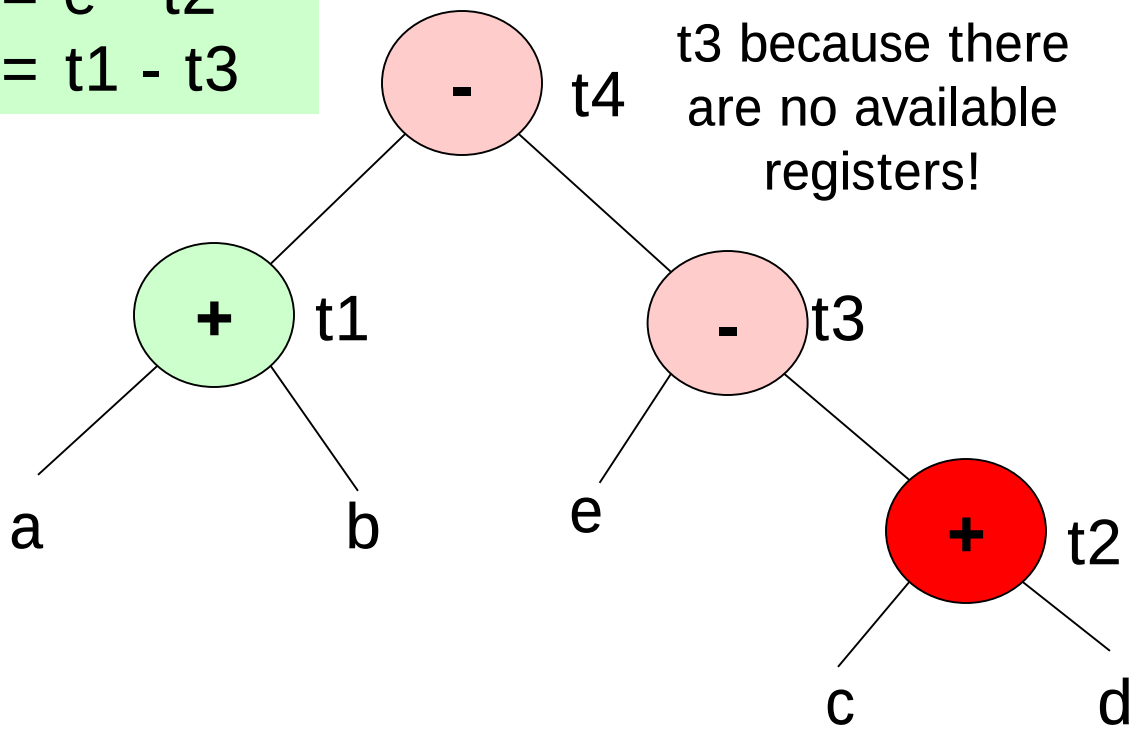
```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```

Assume that SUB only works with registers.

→

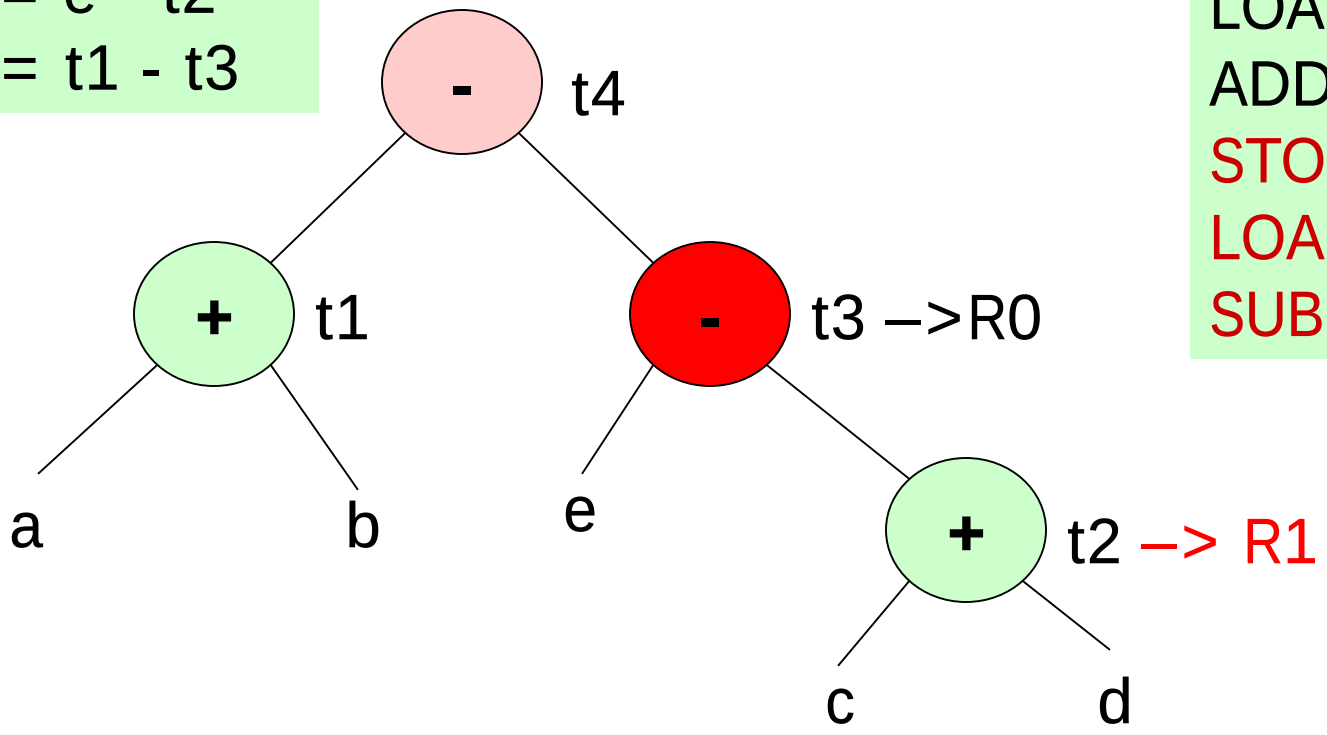
Can't evaluate t3 because there are no available registers!

```
LOAD    a, R0
ADD     b, R0
LOAD    c, R1
ADD     d, R1
```



Example

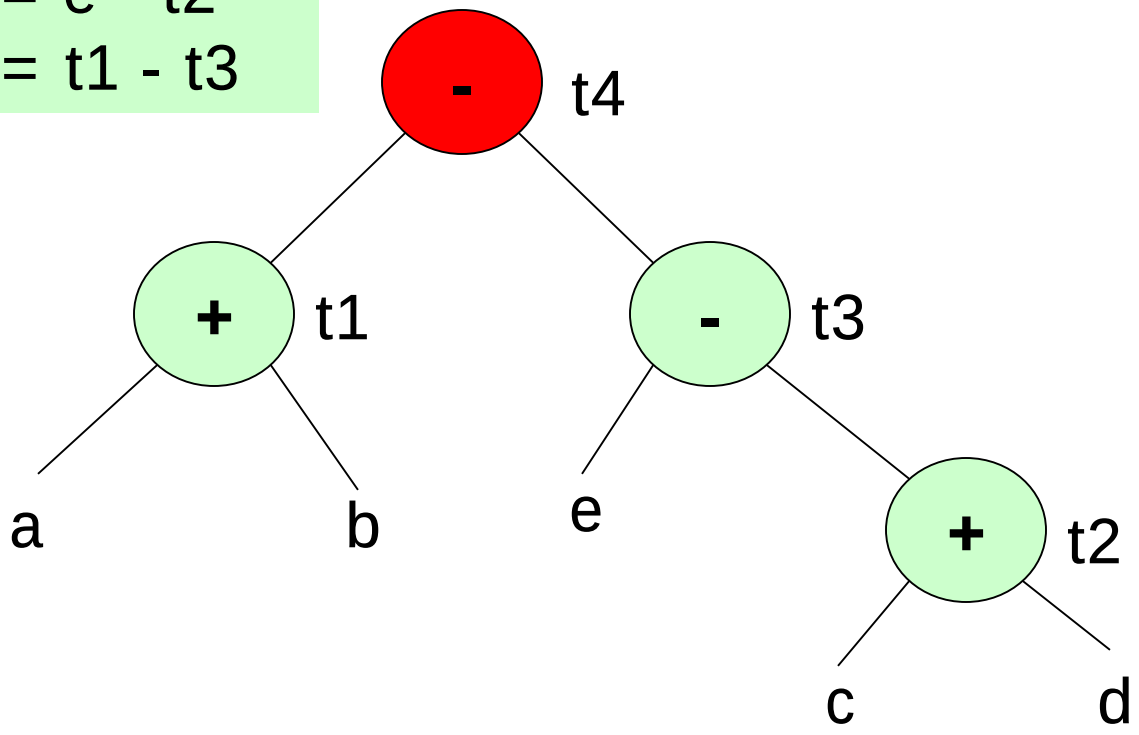
```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```




```
LOAD    a, R0
ADD     b, R0
LOAD    c, R1
ADD     d, R1
STORE   R0, t1
LOAD    e, R0
SUB     R1, R0
```

Example

```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```

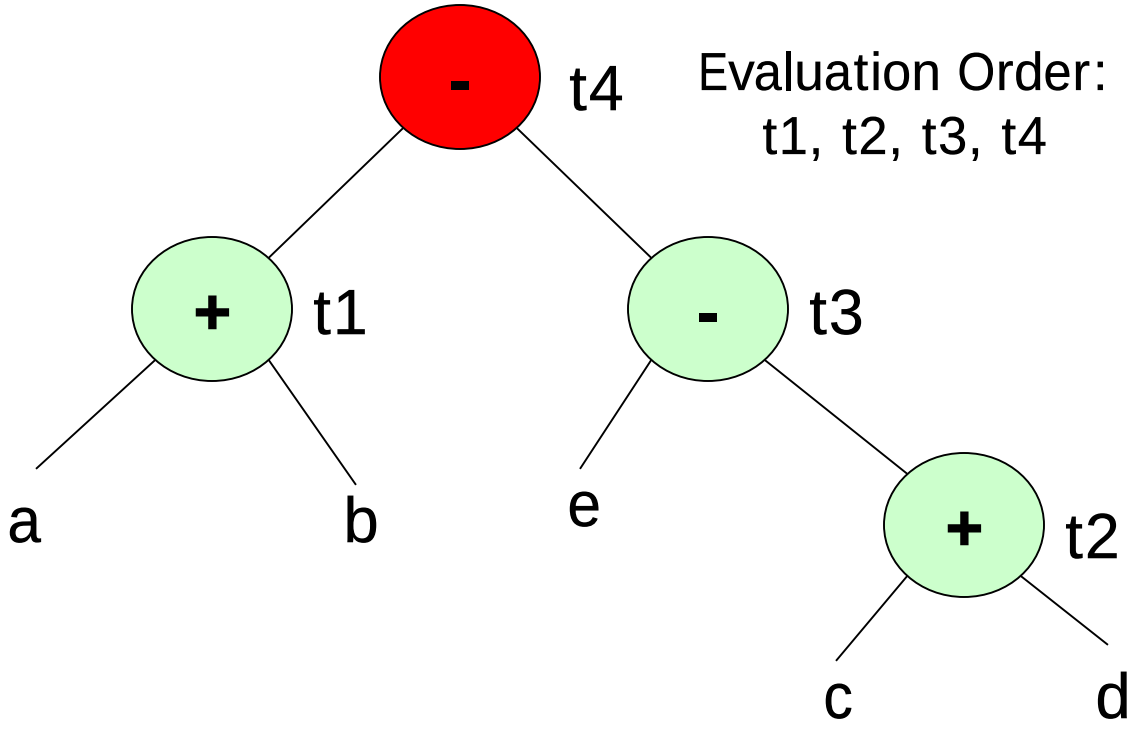



```
LOAD    a, R0
ADD     b, R0
LOAD    c, R1
ADD     d, R1
STORE   R0, t1
LOAD    e, R0
SUB     R1, R0
LOAD    t1, R1
SUB     R0, R1
STORE   R1, t4
```

Example

```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```

1 spill



Evaluation Order:
t1, t2, t3, t4

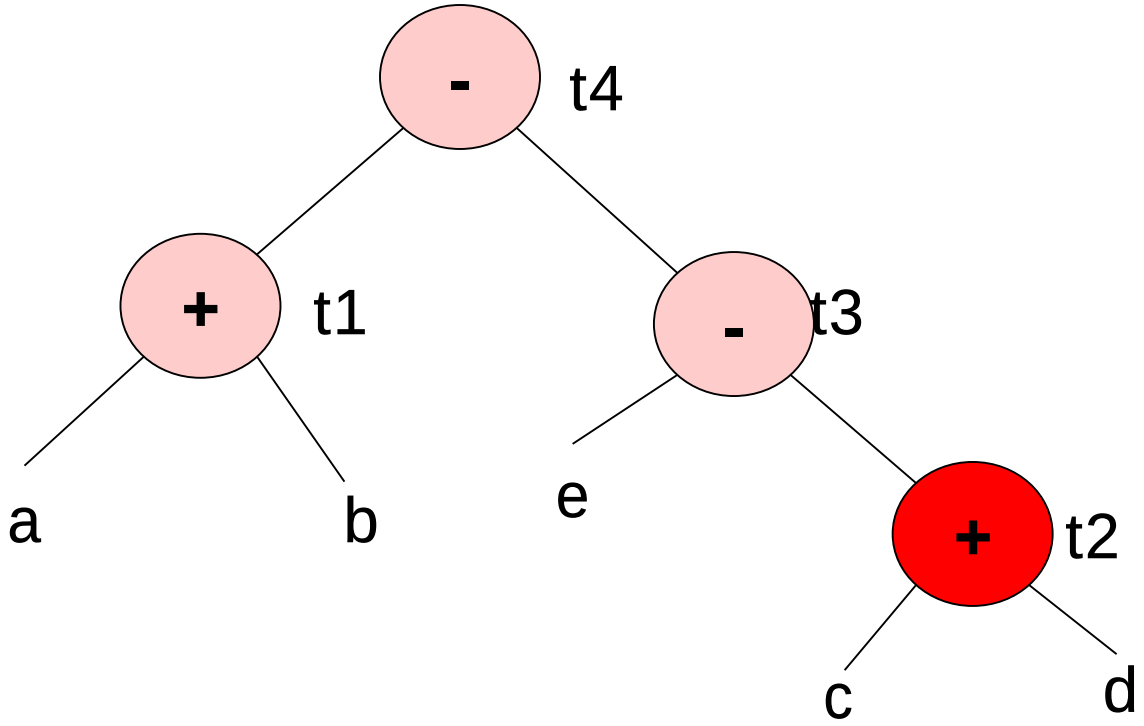
```
LOAD   a, R0
ADD    b, R0
LOAD   c, R1
ADD    d, R1
STORE  R0, t1
LOAD   e, R0
SUB    R1, R0
LOAD   t1, R1
SUB    R0, R1
STORE  R1, t4
```

10 instructions

Example
(can we do better?)

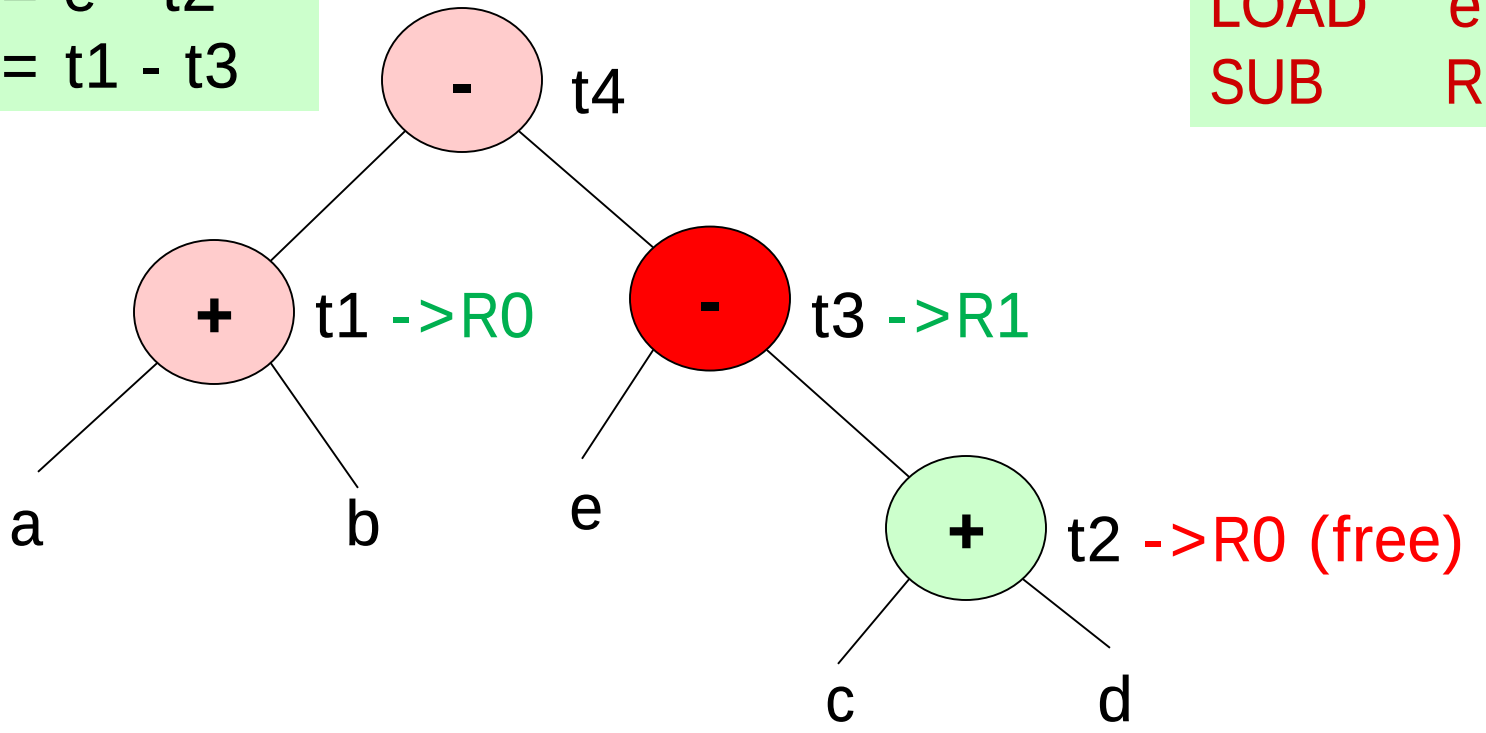
```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```

```
LOAD   c, R0
ADD    d, R0
```



Example
(can we do better?)

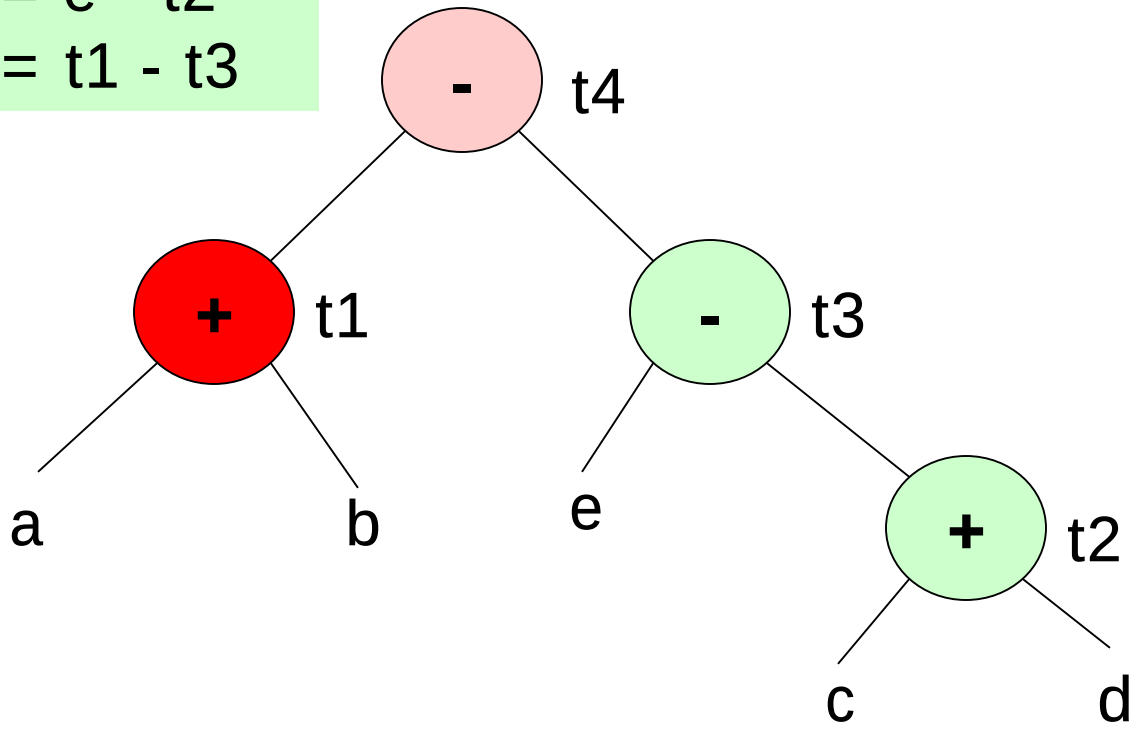
```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```




```
LOAD    c, R0
ADD     d, R0
LOAD    e, R1
SUB     R0,R1
```

Example
(can we do better?)

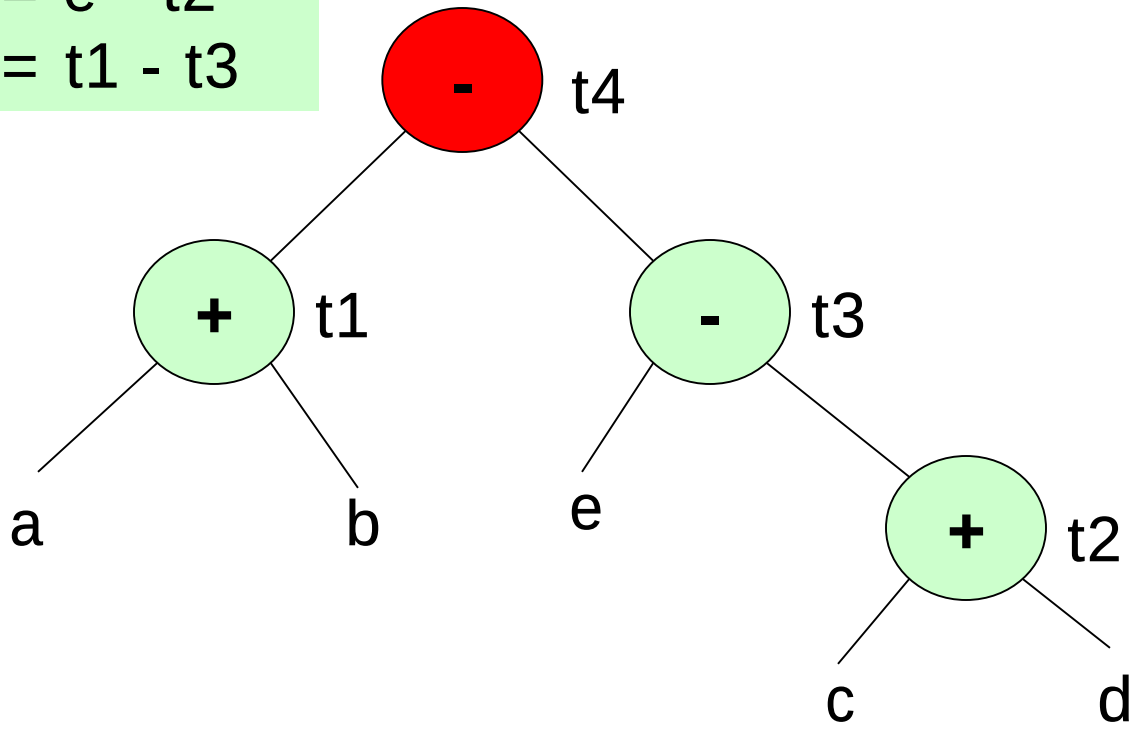
```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```




```
LOAD    c, R0
ADD     d, R0
LOAD    e, R1
SUB     R0, R1
LOAD    a, R0
ADD     b, R0
```

Example
(can we do better?)

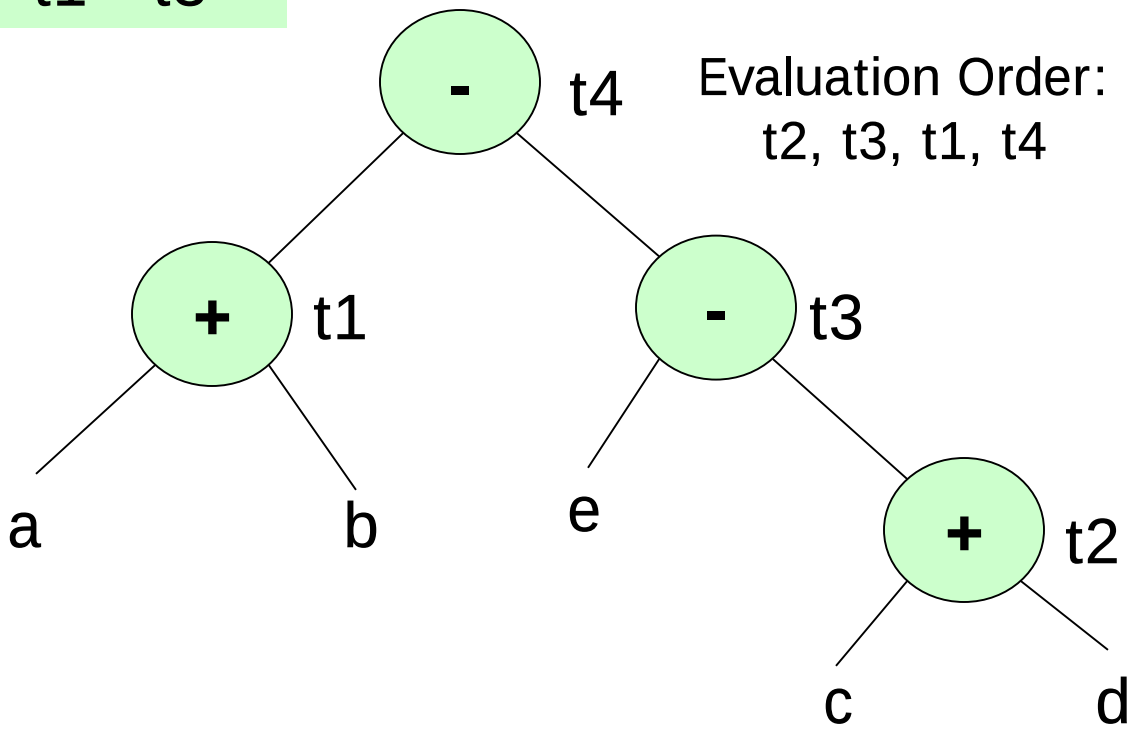
```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```




```
LOAD    c, R0
ADD     d, R0
LOAD    e, R1
SUB     R0, R1
LOAD    a, R0
ADD     b, R0
SUB     R1, R0
STORE   R0, t4
```

Example
(can we do better? Yes!!!)

```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```



no spills!!!

Evaluation Order:
t2, t3, t1, t4

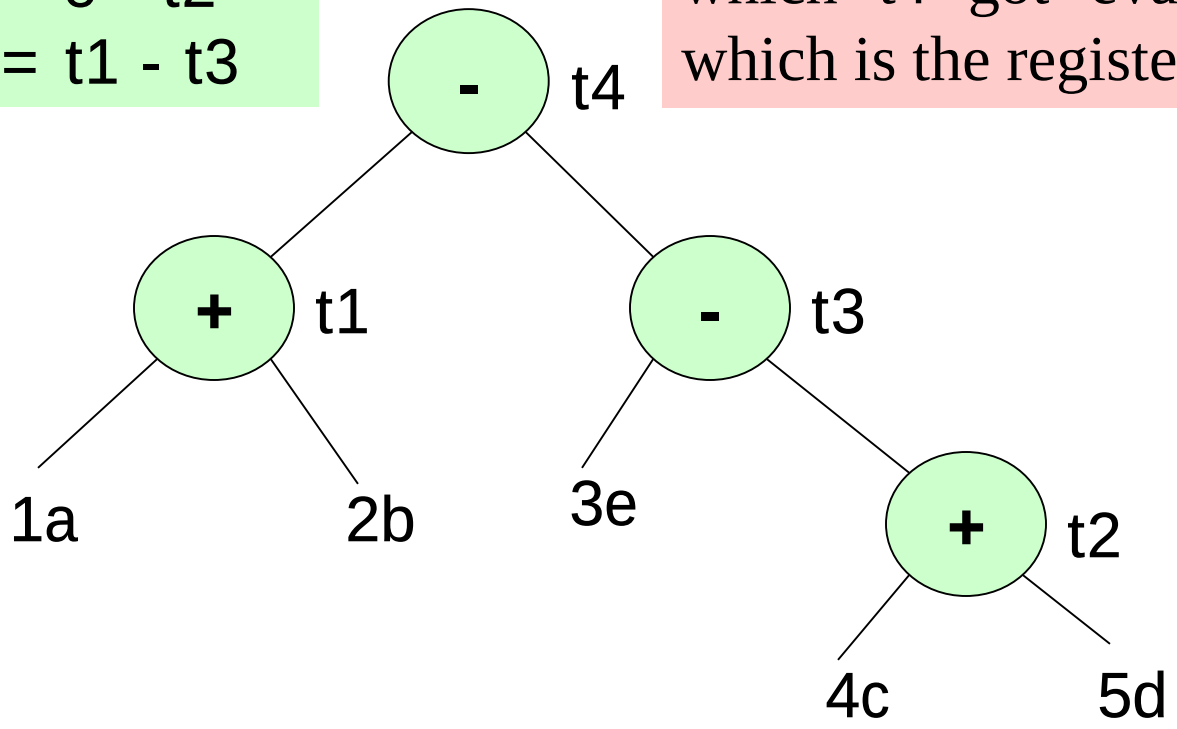
```
LOAD    c, R0
ADD     d, R0
LOAD    e, R1
SUB     R0, R1
LOAD    a, R0
ADD     b, R0
SUB     R1, R0
STORE   R0, t4
```

8 instructions

Example:
Why the improvement?

```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```

We evaluated t4 immediately after t1 (its leftmost argument). Due to which t4 got evaluated in R0 itself which is the register of the left child t1.

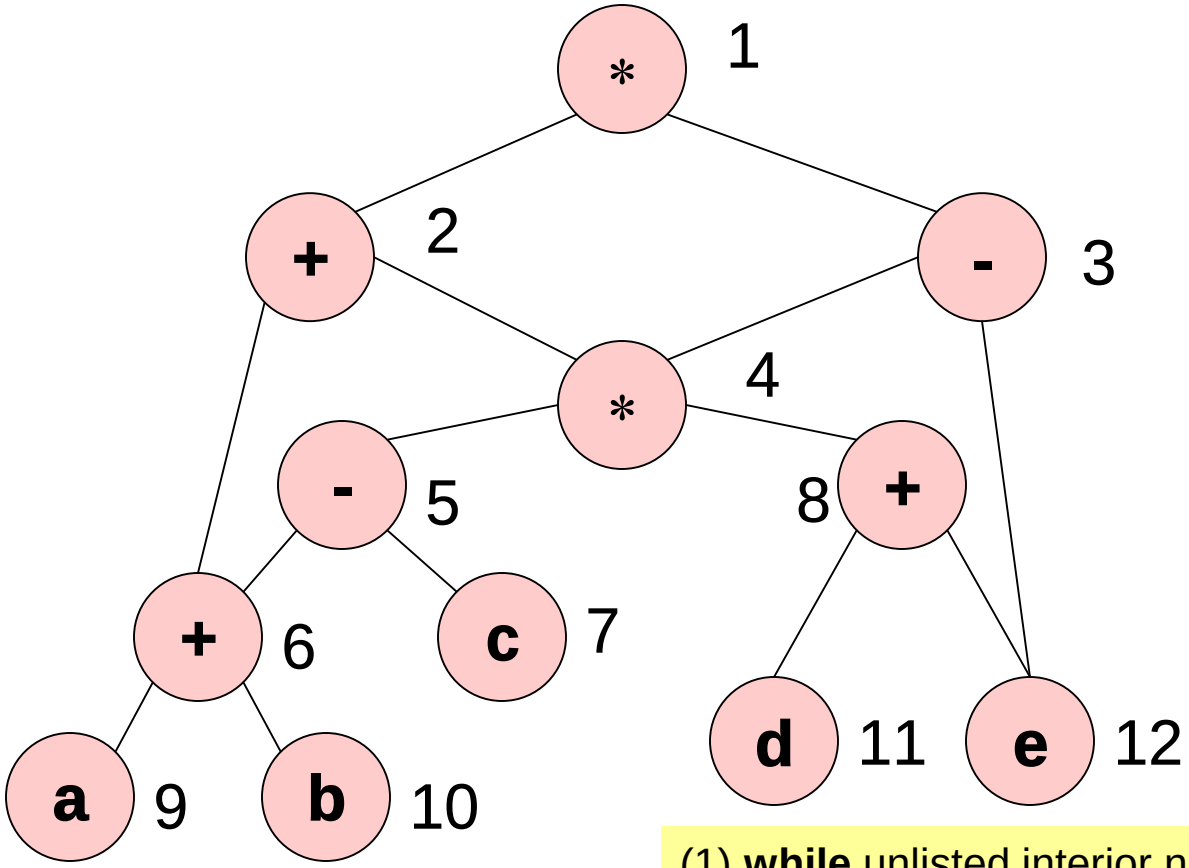


R0 (t1,t4)
R1 (t3)

Heuristic Node Listing Algorithm for a DAG

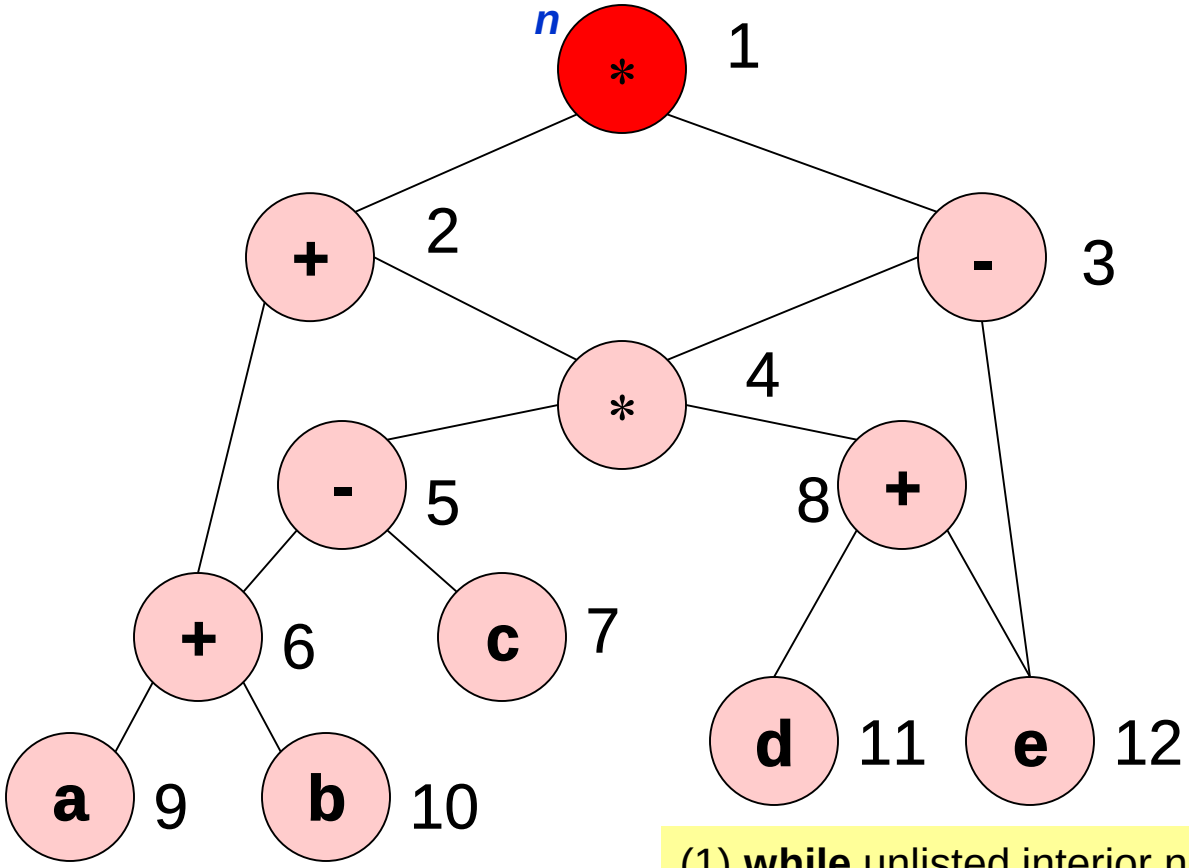
(1) **while** unlisted interior nodes remain
(2) select an unlisted node ***n***, all of whose parents have been listed;
(3) list ***n***;
(4) **while** the leftmost child ***m*** of ***n*** has no unlisted parents, and is not a leaf node do
/* since ***n*** was just listed, ***m*** is not yet listed */
(5) list ***m***;
(6) ***n*** := ***m***;
endwhile
endwhile

Node Listing Example



(1) **while** unlisted interior nodes remain
(2) select an unlisted node ***n***, all of whose parents have been listed;
(3) list ***n***;

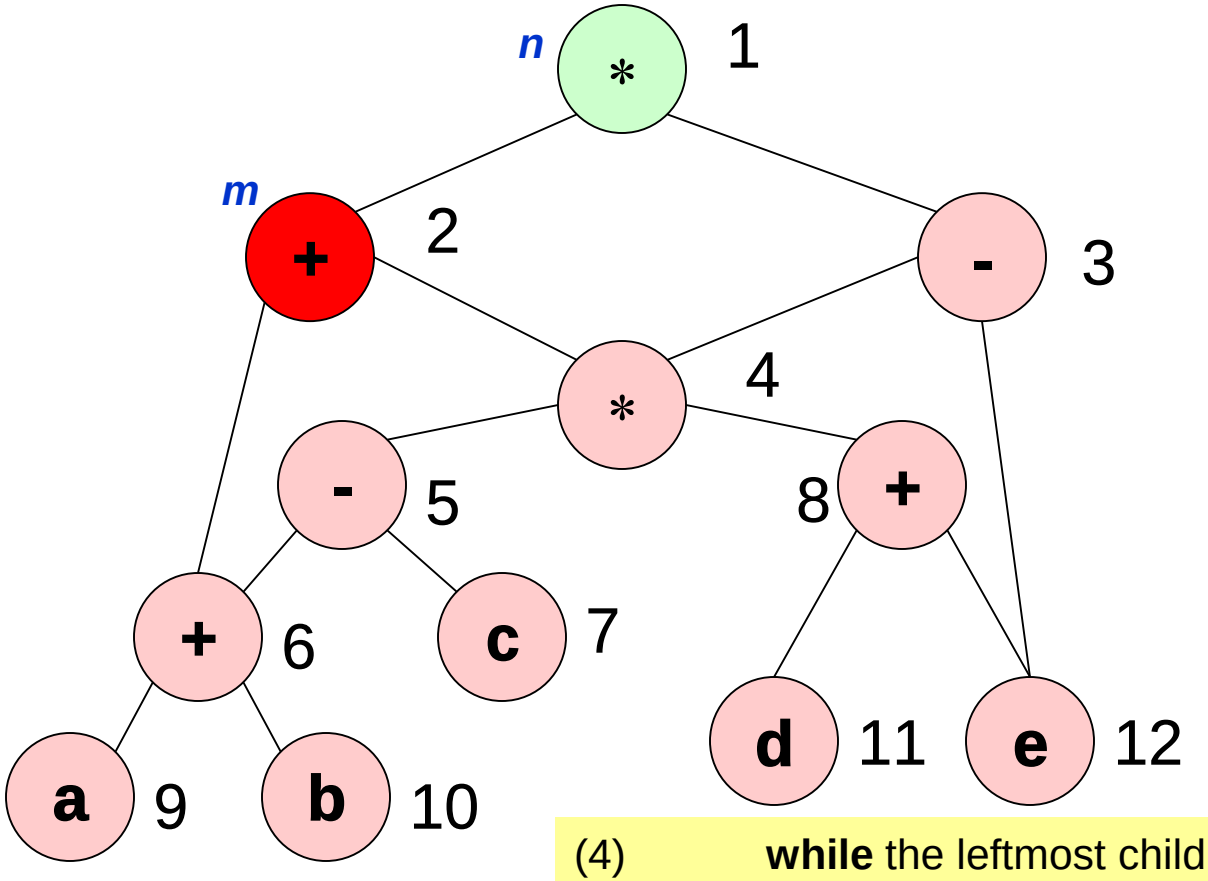
Node Listing Example



List:
1

(1) **while** unlisted interior nodes remain
(2) select an unlisted node ***n***, all of whose parents have been listed;
(3) list ***n***;

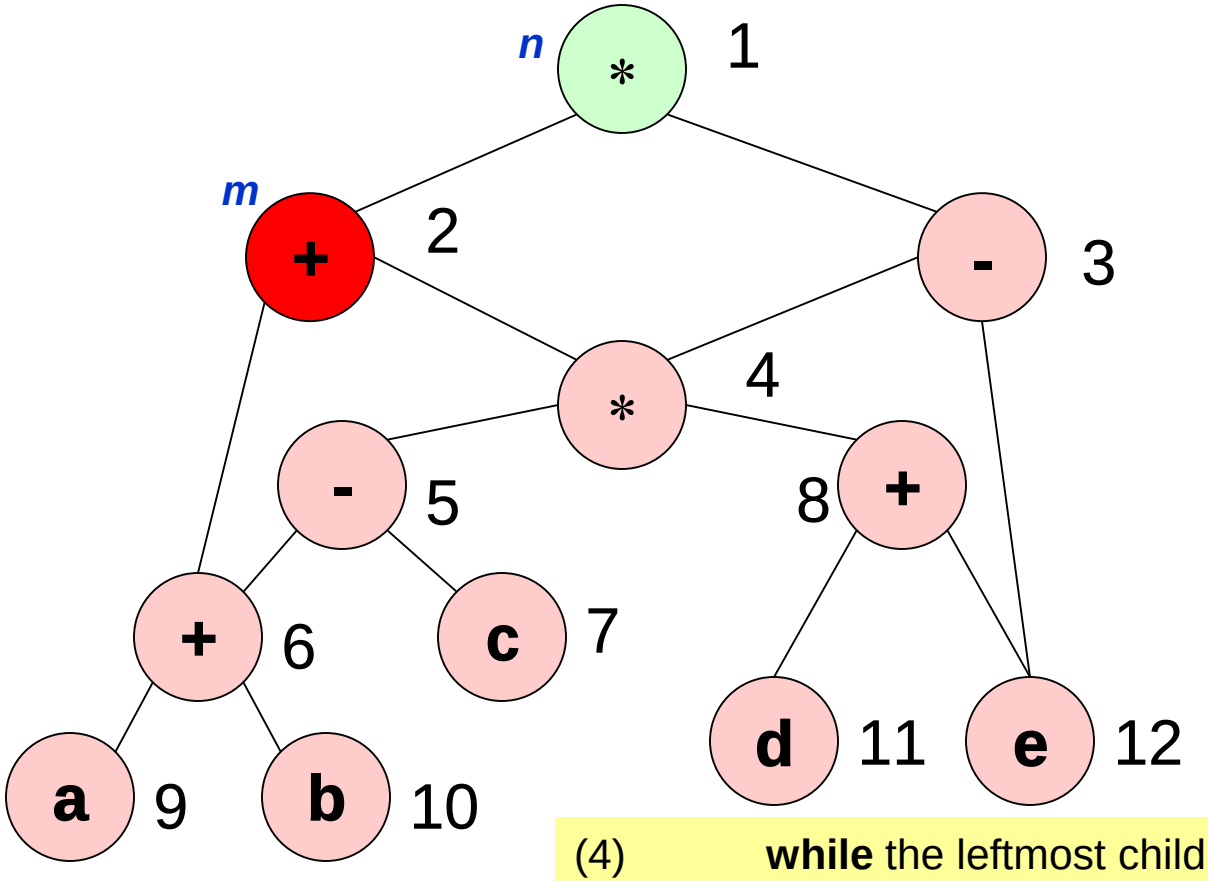
Node Listing Example



List:
1

(4) **while** the leftmost child ***m*** of ***n*** has no unlisted parents,
and is not a leaf node do
/* since ***n*** was just listed, ***m*** is not yet listed */
(5) list ***m***;
(6) ***n*** := ***m***;

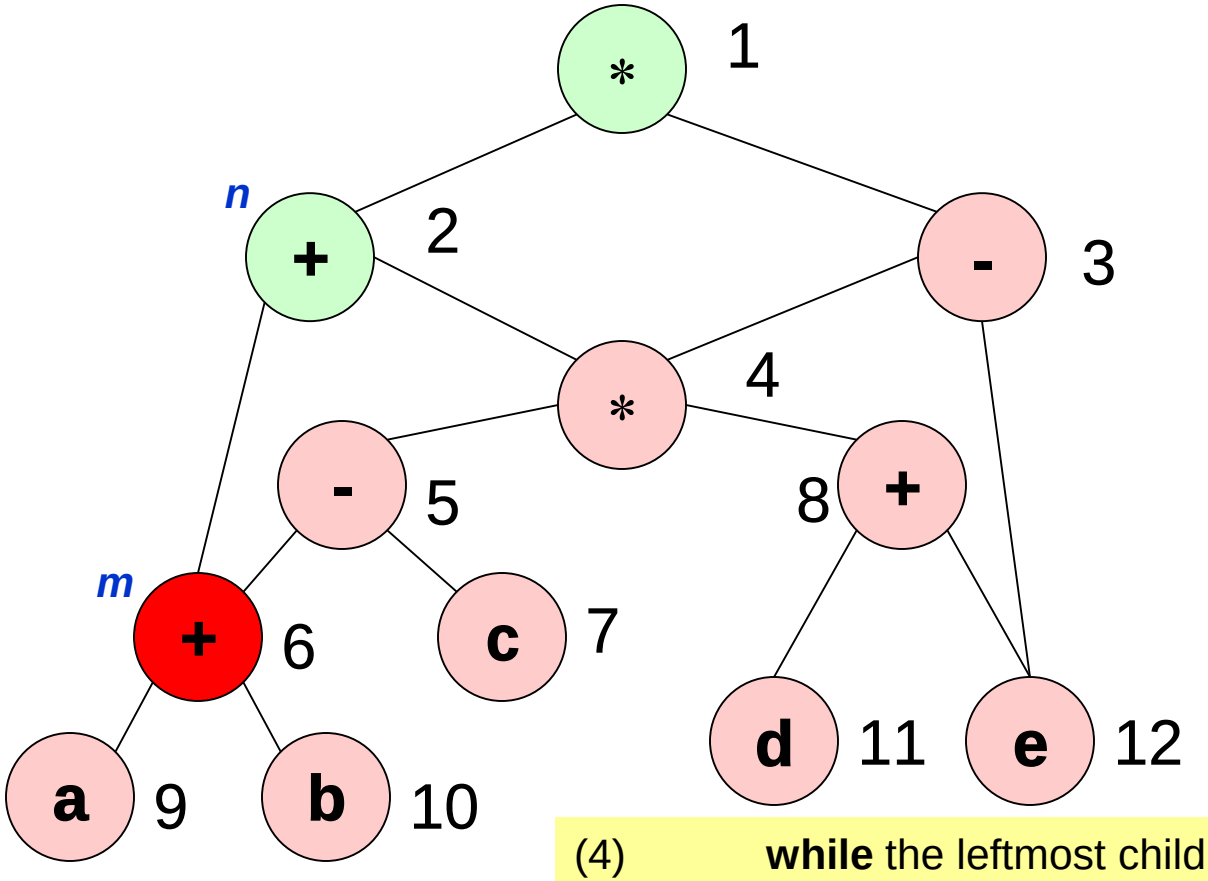
Node Listing Example



List:
1
2

(4) **while** the leftmost child ***m*** of ***n*** has no unlisted parents,
and is not a leaf node do
/* since ***n*** was just listed, ***m*** is not yet listed */
(5) list ***m***;
(6) ***n*** := ***m***;

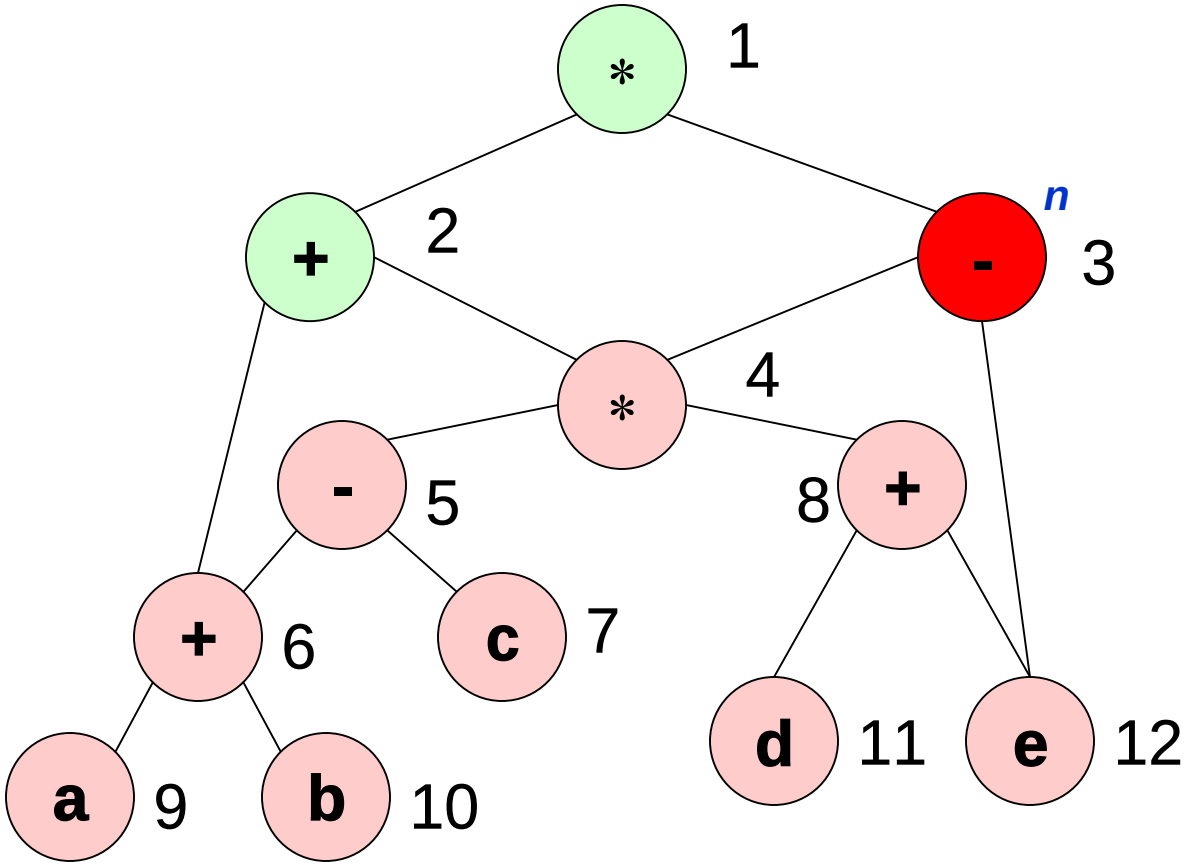
Node Listing Example



List:
1
2

(4) **while** the leftmost child ***m*** of ***n*** has no unlisted parents,
and is not a leaf node do
/* since ***n*** was just listed, ***m*** is not yet listed */
(5) list ***m***;
(6) ***n*** := ***m***;

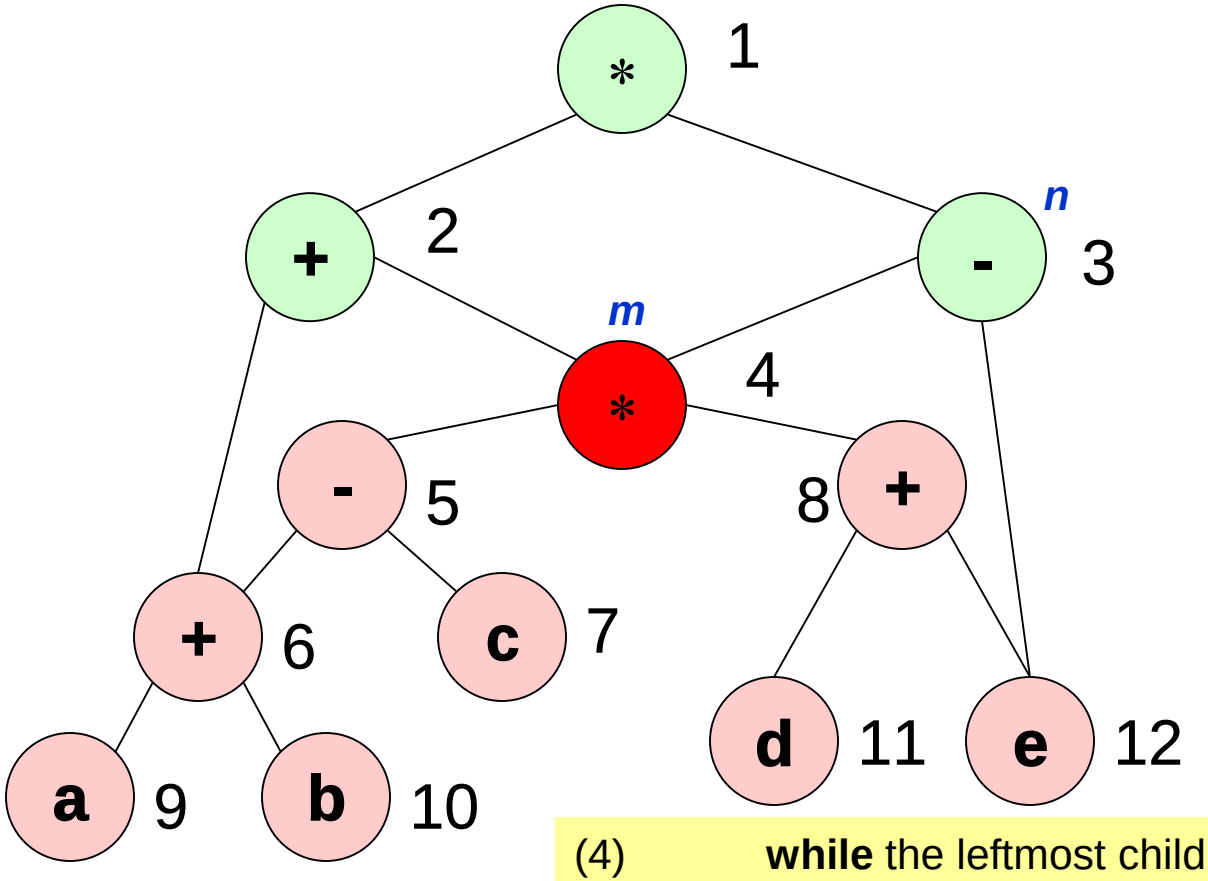
Node Listing Example



List:
1
2
3

(1) **while** unlisted interior nodes remain
(2) select an unlisted node ***n***, all of whose parents have been listed;
(3) list ***n***;

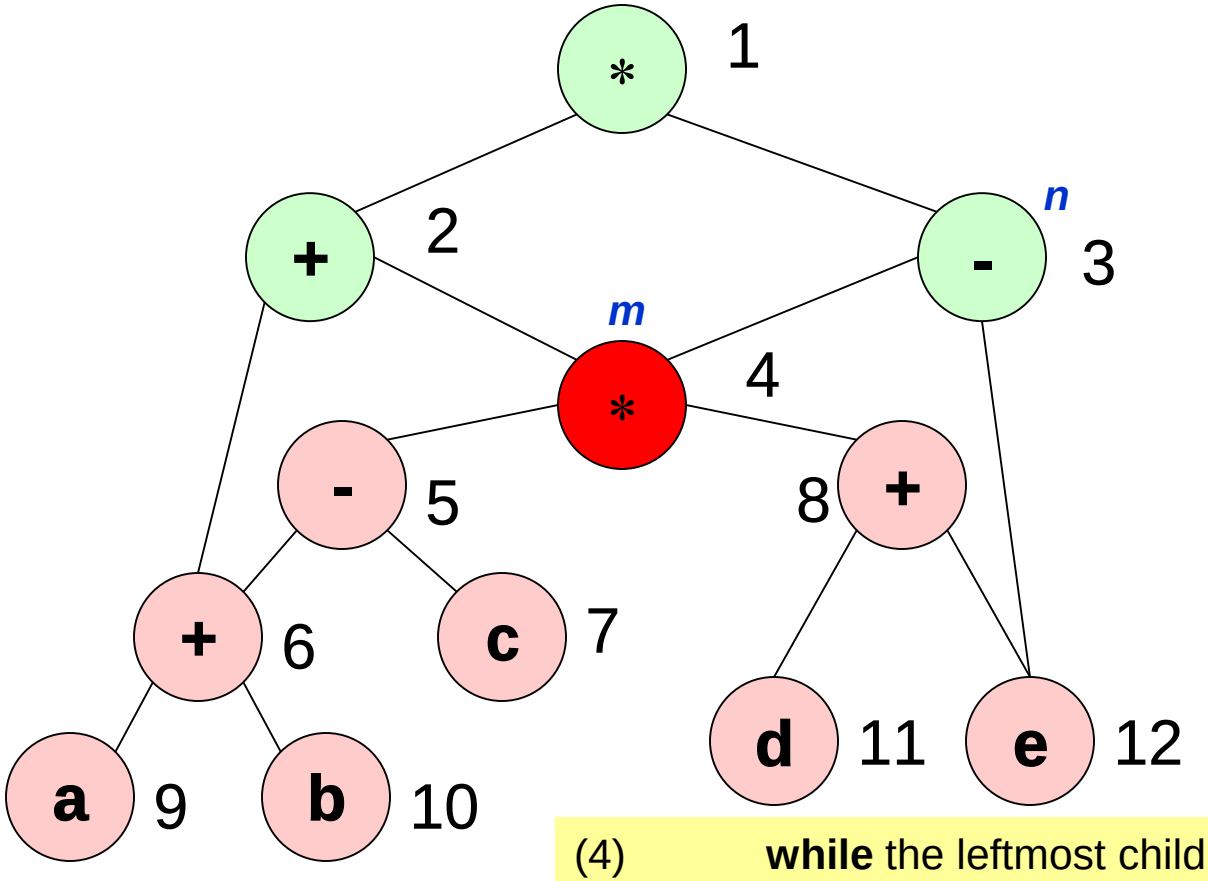
Node Listing Example



List:
1
2
3

(4) **while** the leftmost child ***m*** of ***n*** has no unlisted parents,
and is not a leaf node do
/* since ***n*** was just listed, ***m*** is not yet listed */
(5) list ***m***;
(6) ***n*** := ***m***;

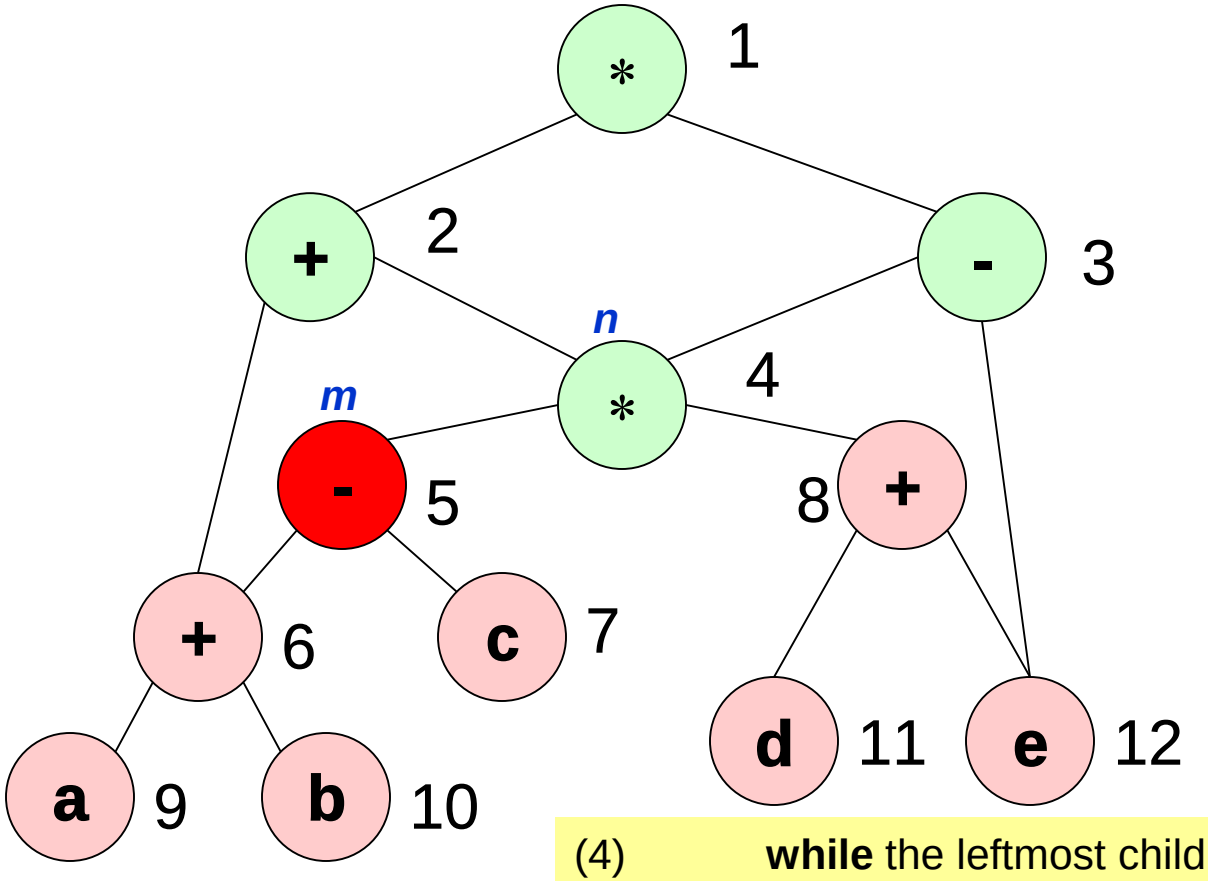
Node Listing Example



List:
1
2
3
4

(4) **while** the leftmost child ***m*** of ***n*** has no unlisted parents,
and is not a leaf node do
/* since ***n*** was just listed, ***m*** is not yet listed */
(5) list ***m***;
(6) ***n*** := ***m***;

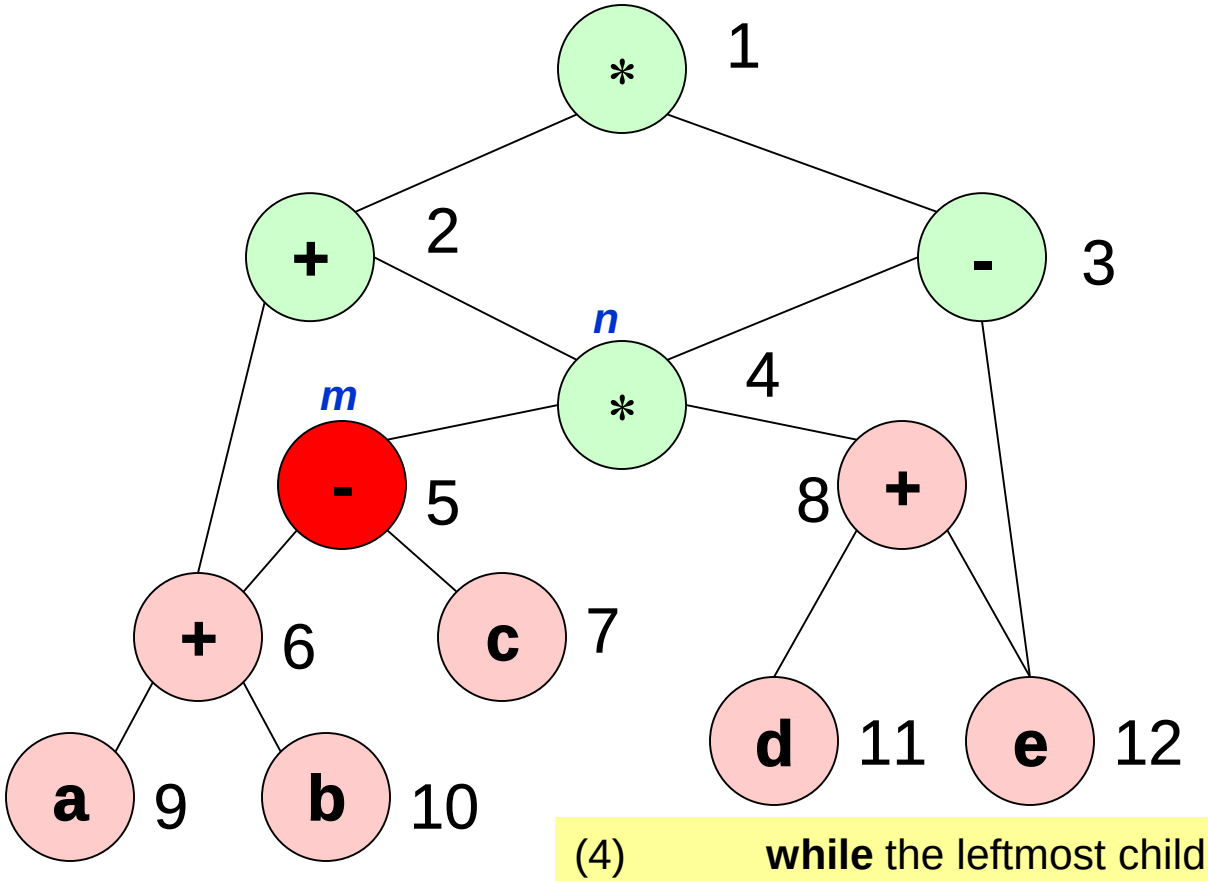
Node Listing Example



List:
1
2
3
4

(4) **while** the leftmost child ***m*** of ***n*** has no unlisted parents,
and is not a leaf node do
/* since ***n*** was just listed, ***m*** is not yet listed */
(5) list ***m***;
(6) ***n*** := ***m***;

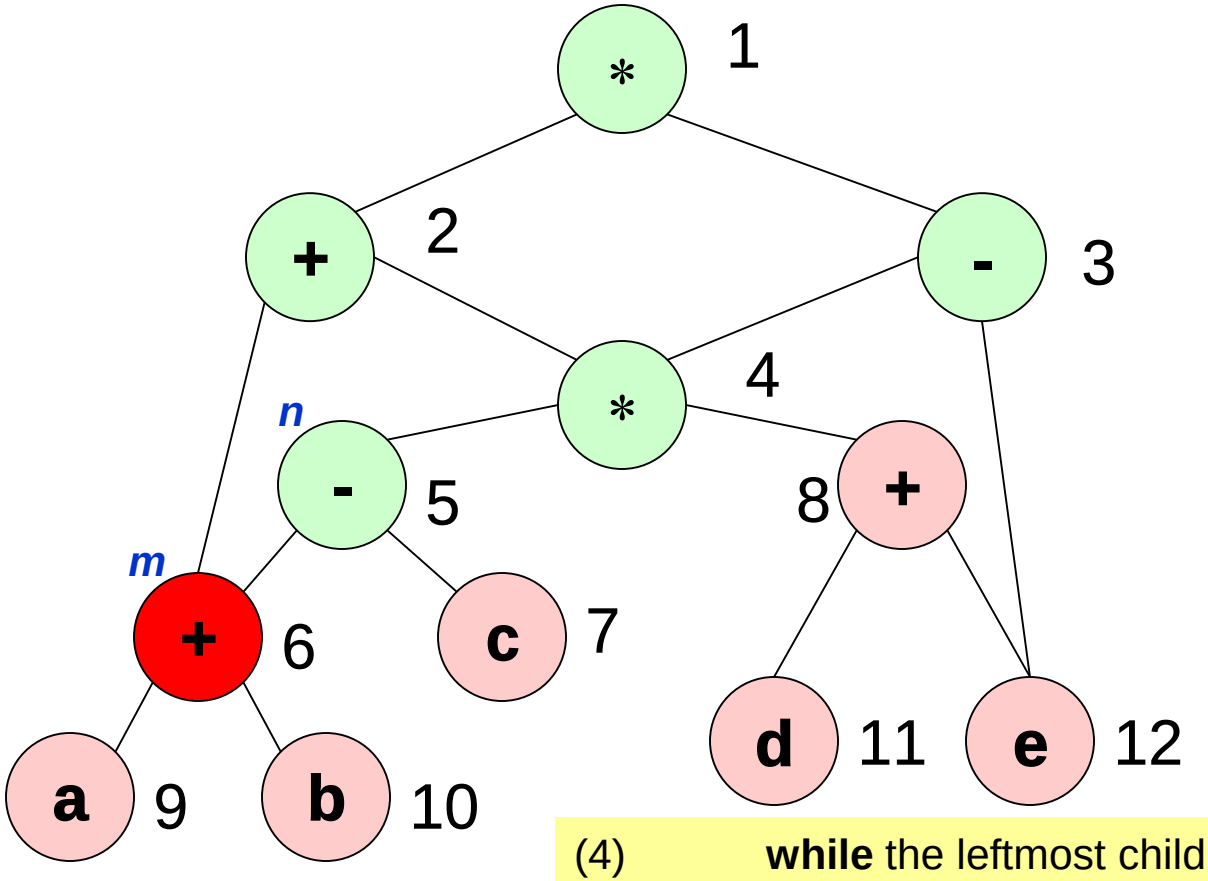
Node Listing Example



List:
1
2
3
4
5

(4) **while** the leftmost child ***m*** of ***n*** has no unlisted parents,
and is not a leaf node do
/* since ***n*** was just listed, ***m*** is not yet listed */
(5) list ***m***;
(6) ***n*** := ***m***;

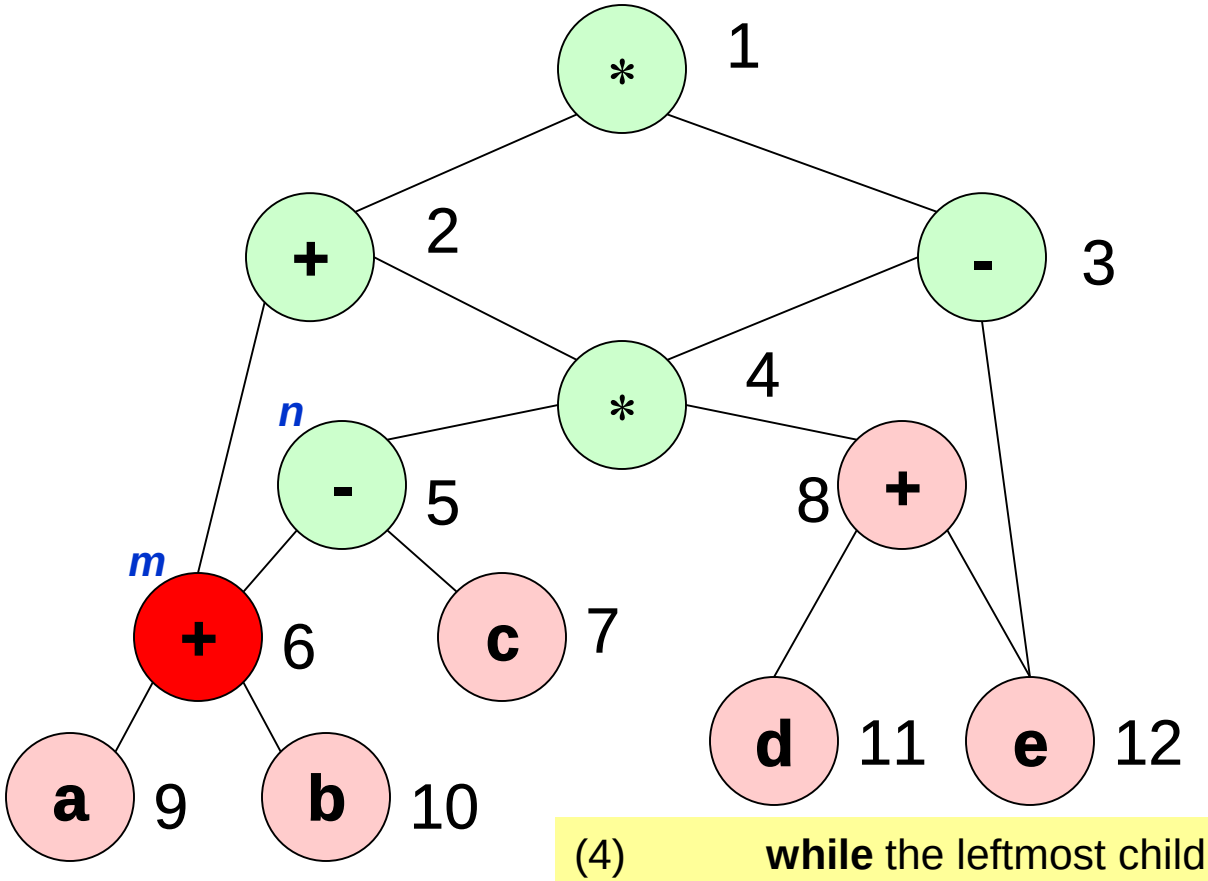
Node Listing Example



List:
1
2
3
4
5

(4) **while** the leftmost child ***m*** of ***n*** has no unlisted parents,
and is not a leaf node do
/* since ***n*** was just listed, ***m*** is not yet listed */
(5) list ***m***;
(6) ***n*** := ***m***;

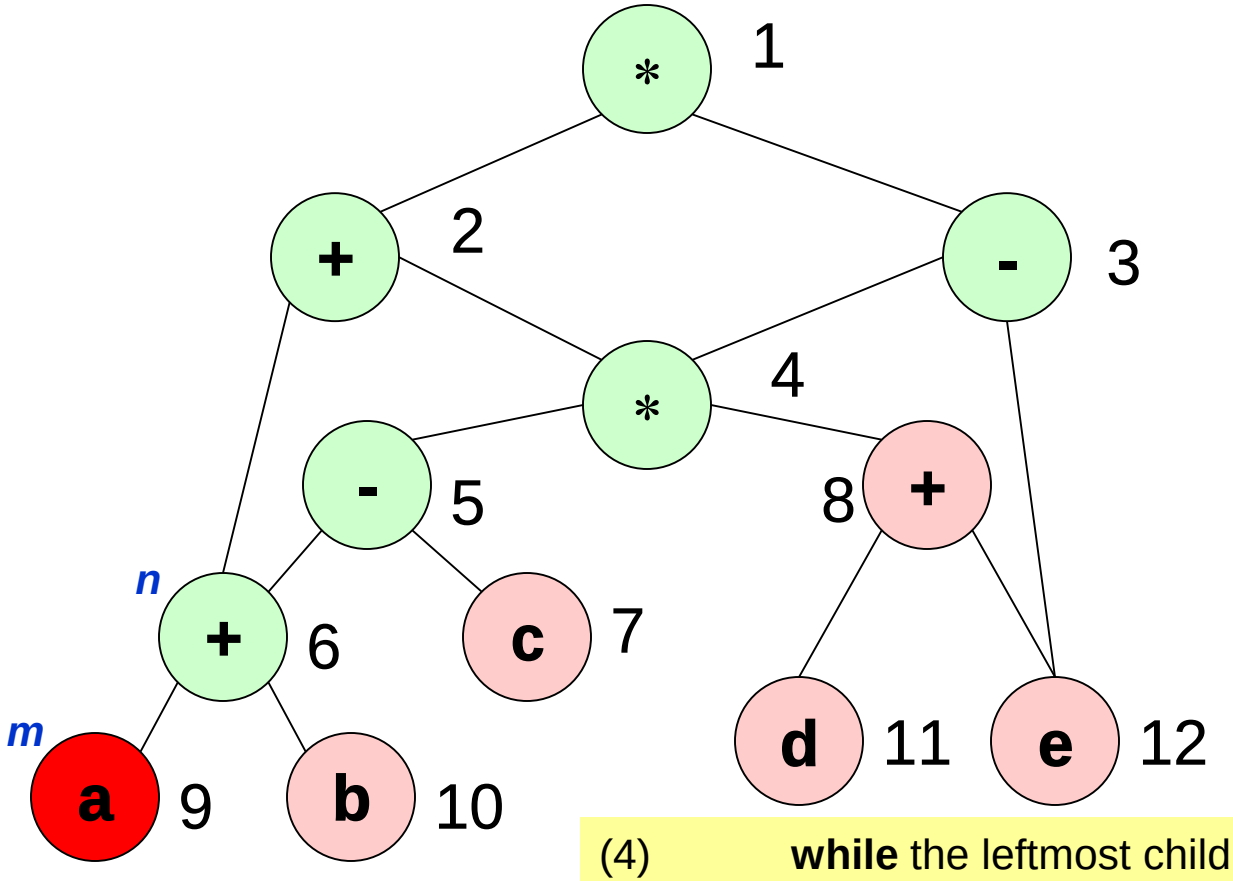
Node Listing Example



List:
1
2
3
4
5
6

(4) **while** the leftmost child ***m*** of ***n*** has no unlisted parents,
and is not a leaf node do
/* since ***n*** was just listed, ***m*** is not yet listed */
(5) list ***m***;
(6) ***n*** := ***m***;

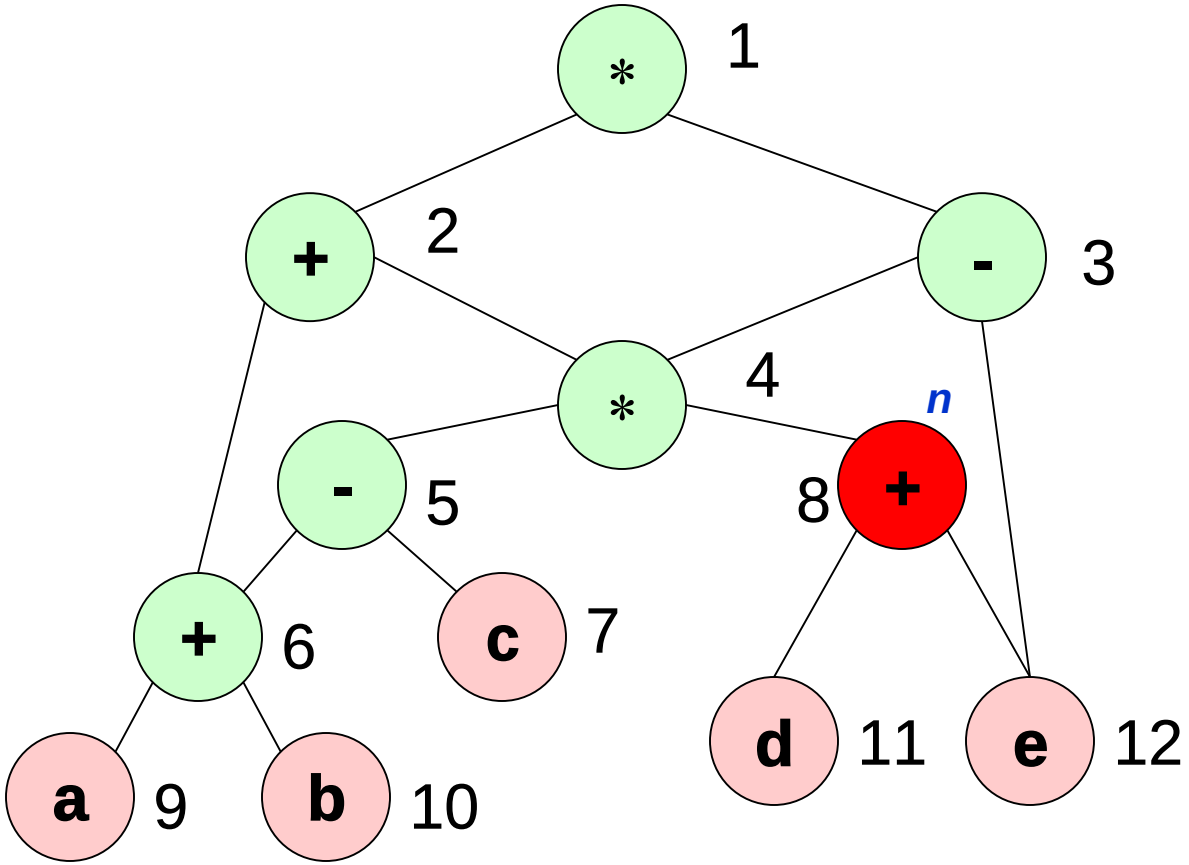
Node Listing Example



List:
1
2
3
4
5
6

(4) **while** the leftmost child ***m*** of ***n*** has no unlisted parents,
and is not a leaf node do
/* since ***n*** was just listed, ***m*** is not yet listed */
(5) list ***m***;
(6) ***n*** := ***m***;

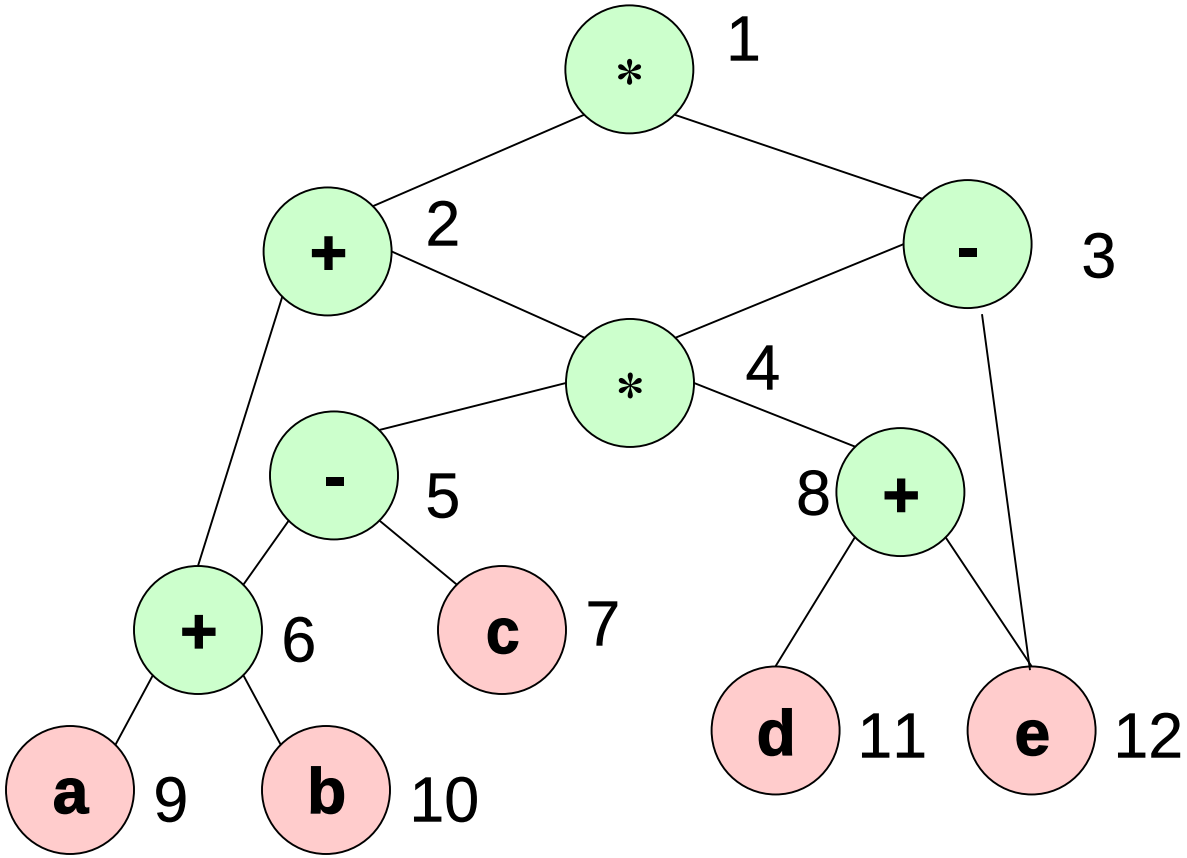
Node Listing Example



List:
1
2
3
4
5
6
8

(1) **while** unlisted interior nodes remain
(2) select an unlisted node ***n***, all of whose parents have been listed;
(3) list ***n***;

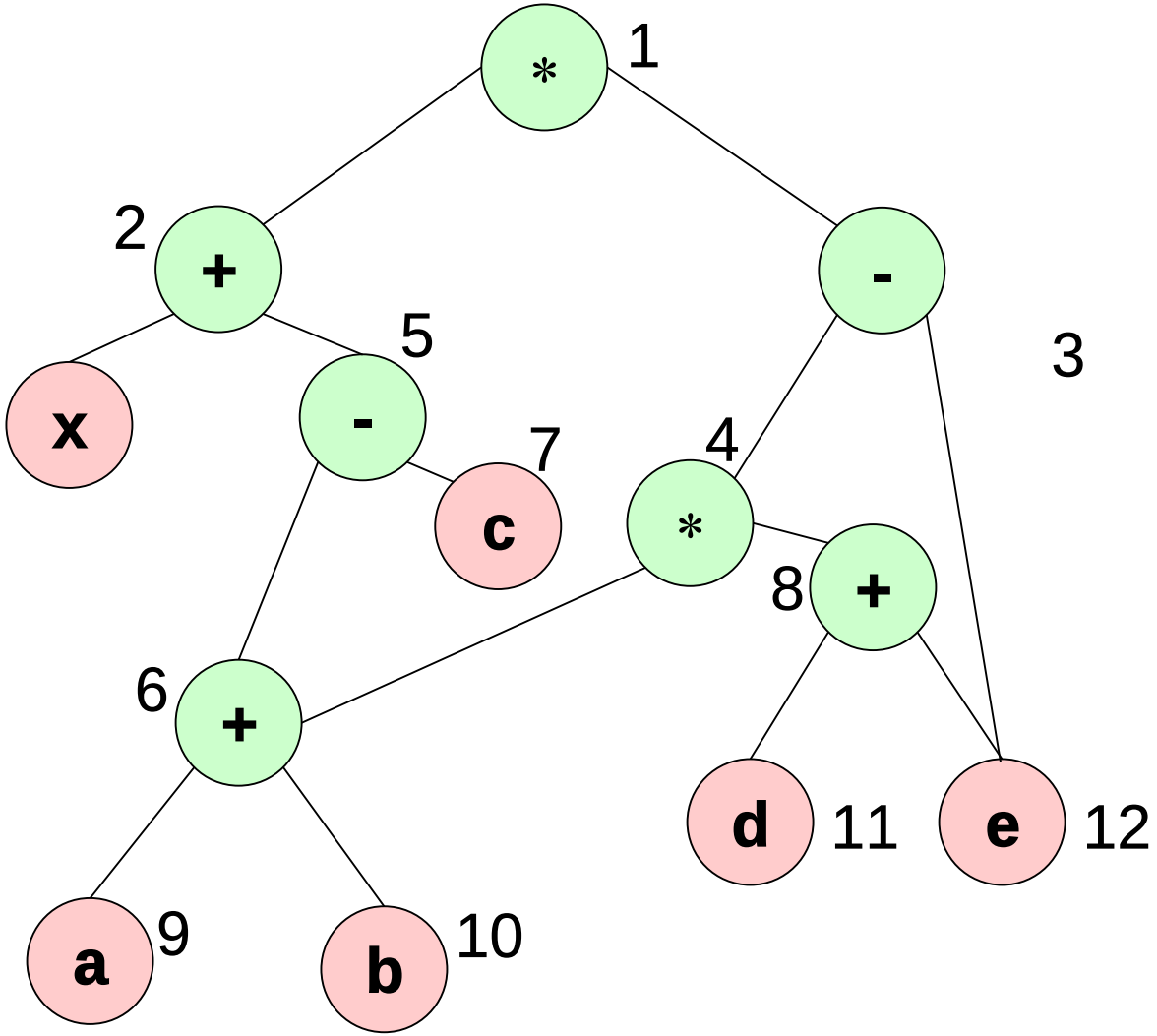
Node Listing Example



List:
1
2
3
4
5
6
8

Therefore the optimal evaluation order (regardless of the number of registers available) for the internal nodes is 8654321.

Node Listing Example2



List:
1
2
3
4
5
6
8

Optimal Code Generation for Trees

If the DAG representing the data flow in a basic block is a tree, then for some machine models, there is a simple algorithm (the SethiUllman algorithm) that gives the optimal order.

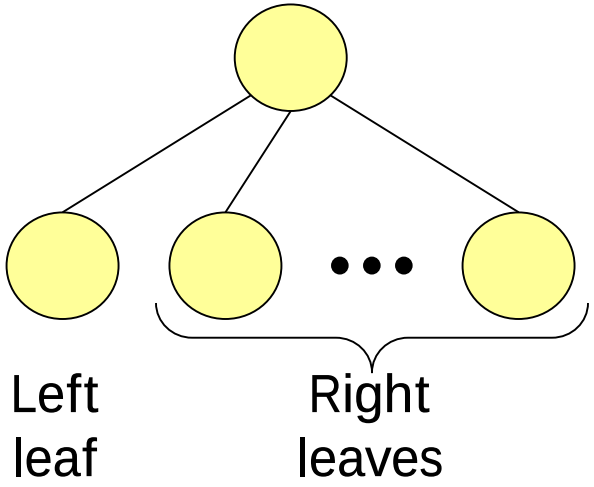
The order is optimal in the sense that it yields the shortest instruction sequence over all instruction sequences that evaluate the tree.

Sethi-Ullman Algorithm

Intuition:

1. Label each node according to the number of registers that are required to generate code for the node.
2. Generate code from top down always generating code first for the child that requires the most registers.

Sethi-Ullman Algorithm (Intuition)



Bottom-Up Labeling: visit a node after all its children are labeled.

Labeling Algorithm

(1) **if** n is a leaf **then**

(2) **if** n is the leftmost child of its parent **then**

(3) $label(n) := 1$

(4) **else** $label(n) := 0$

else begin / * n is an interior node * /

(5) let $c_1, c_2, \cdots, c_k$ be the children of n ordered by *label*

so that $label(c_1) \geq label(c_2) \geq \cdots \geq label(c_k)$

(6) $label(n) := \max_{1 \leq i \leq k} (label(c_i) + i - 1)$

end

Labeling Algorithm

$$label(c_1) \geq label(c_2) \geq \cdots \geq label(c_k)$$

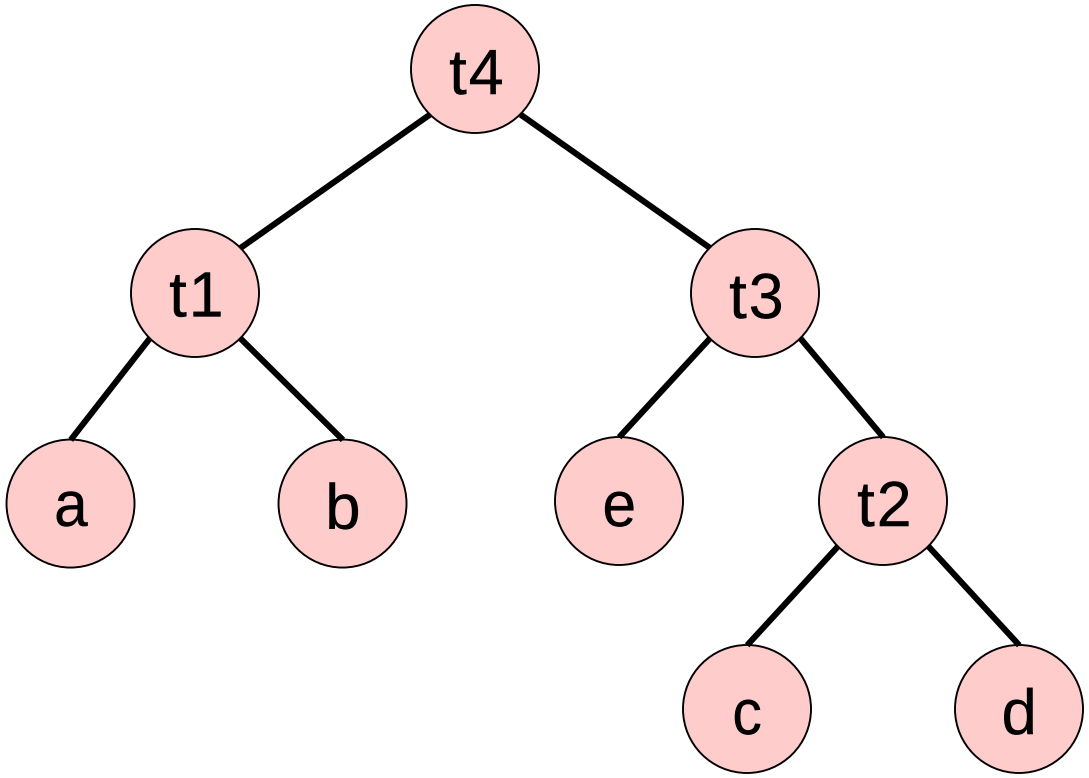
If $k = 1$ (a node with two children), then the following relation

$$label(n_1) := \max_{1 \leq i \leq k} (label(c_i) + i - 1)$$

becomes:

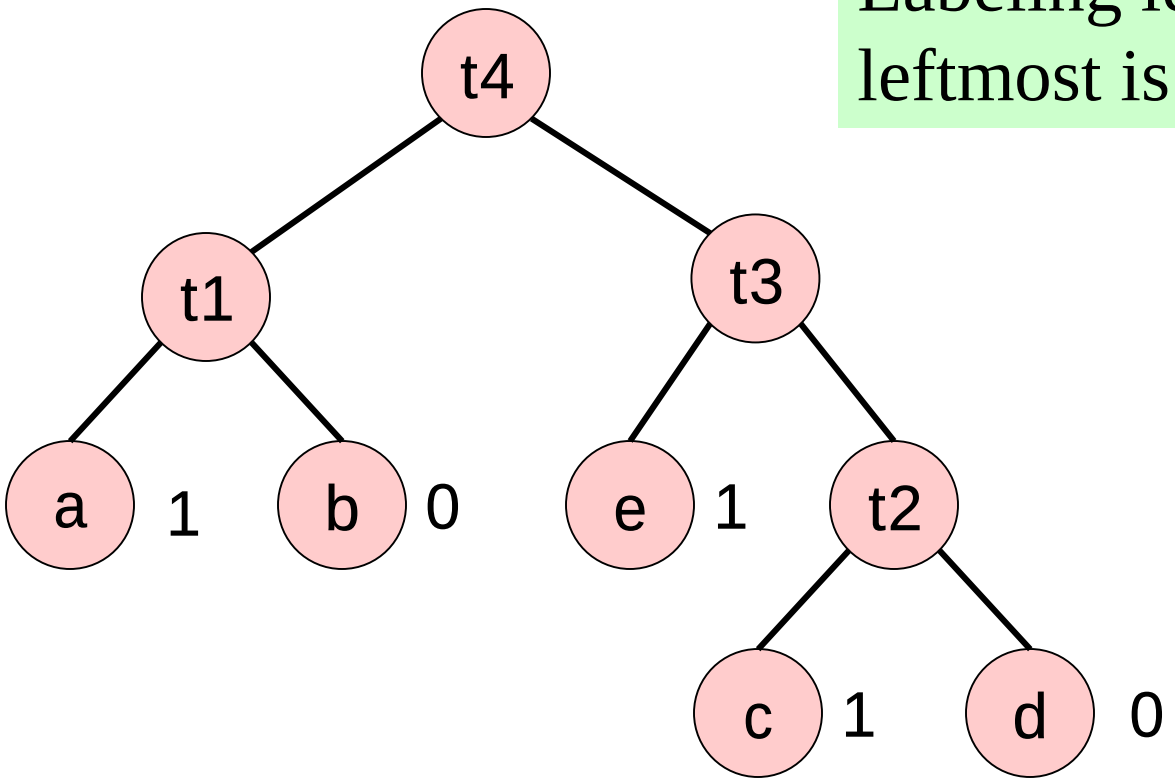
$$label(n) = \begin{cases} \max[label(c_1), label(c_2)] & \text{if } label(c_1) \neq label(c_2) \\ label(c_1) + 1 & \text{if } label(c_1) = label(c_2) \end{cases}$$

Example

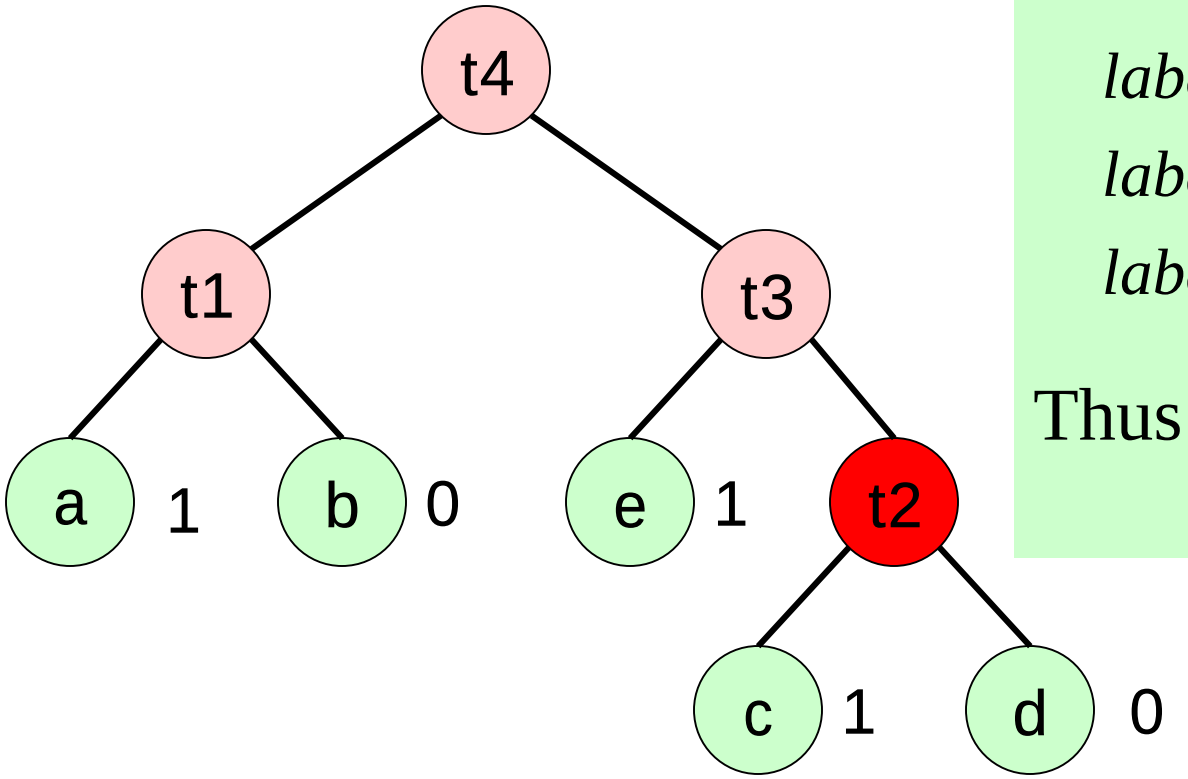


Example

Labeling leaves:
leftmost is 1, others are 0



Example



Labeling t2:

$$label(c) > label(d)$$

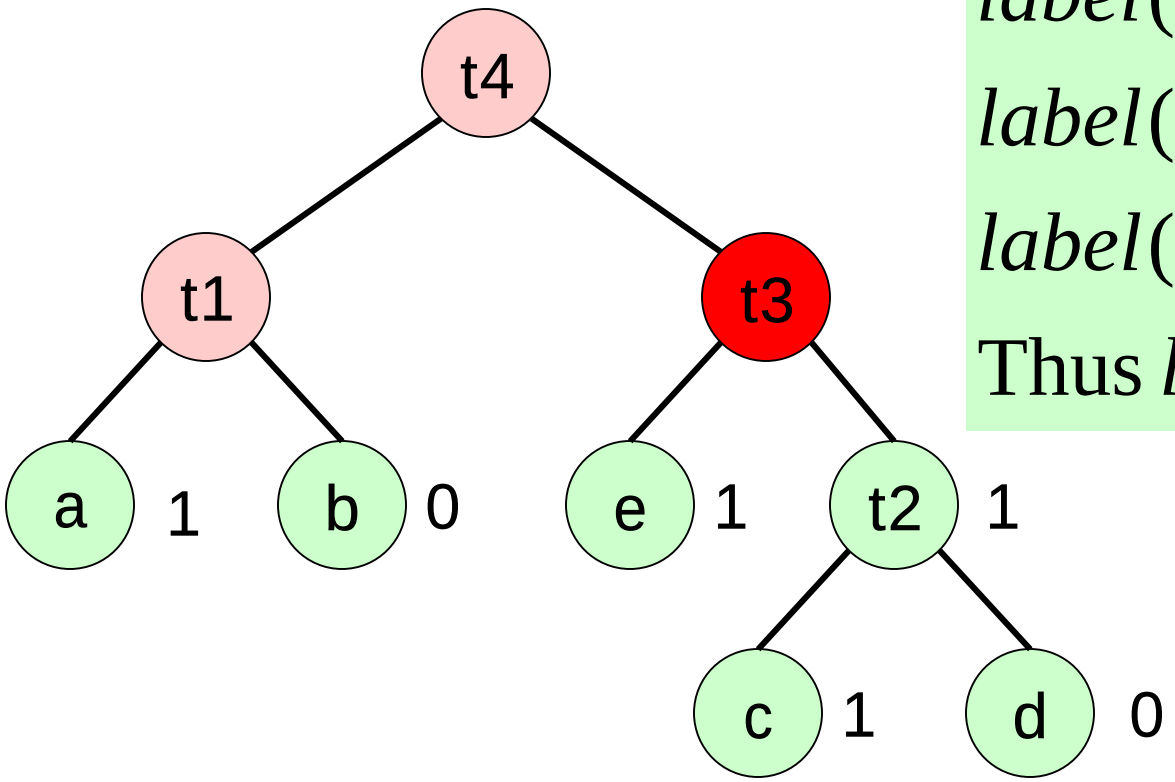
$$label(c) + 1 - 1 = 1$$

$$label(d) + 2 - 1 = 1$$

Thus

$$label(t2) = 1$$

Example



$$label(e) = label(t2)$$

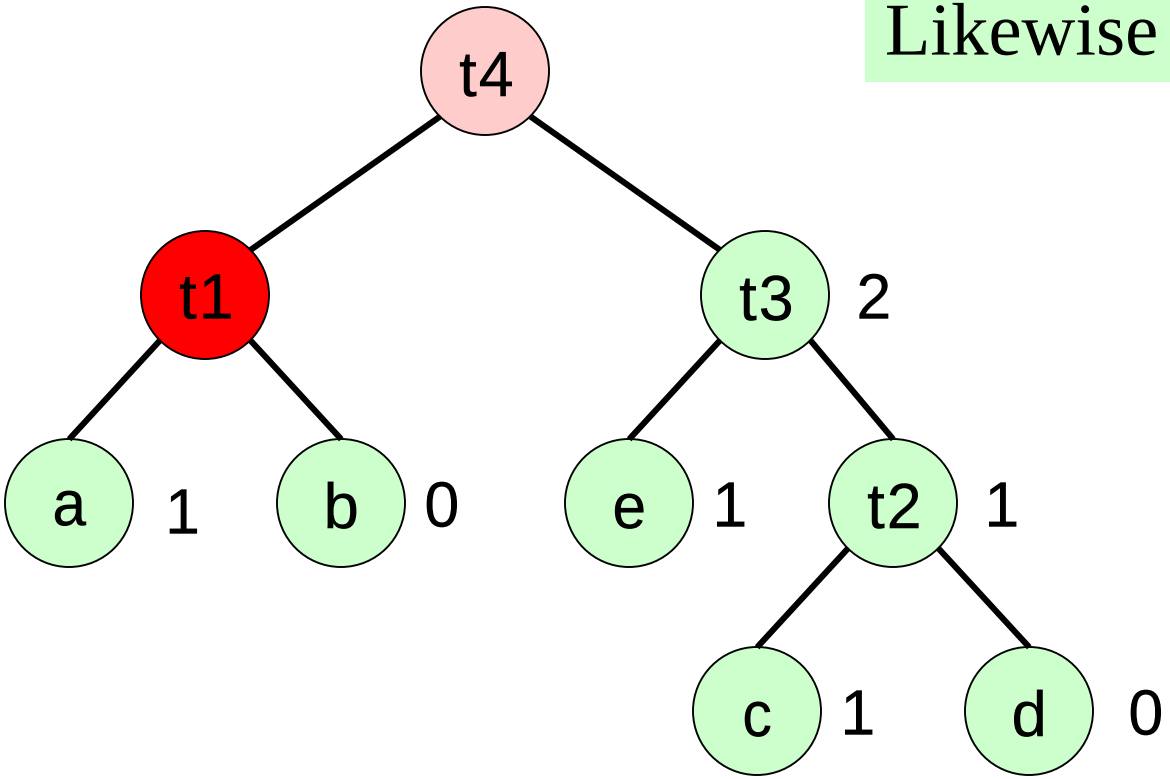
$$label(e) + 1 - 1 = 1$$

$$label(t2) + 2 - 1 = 2$$

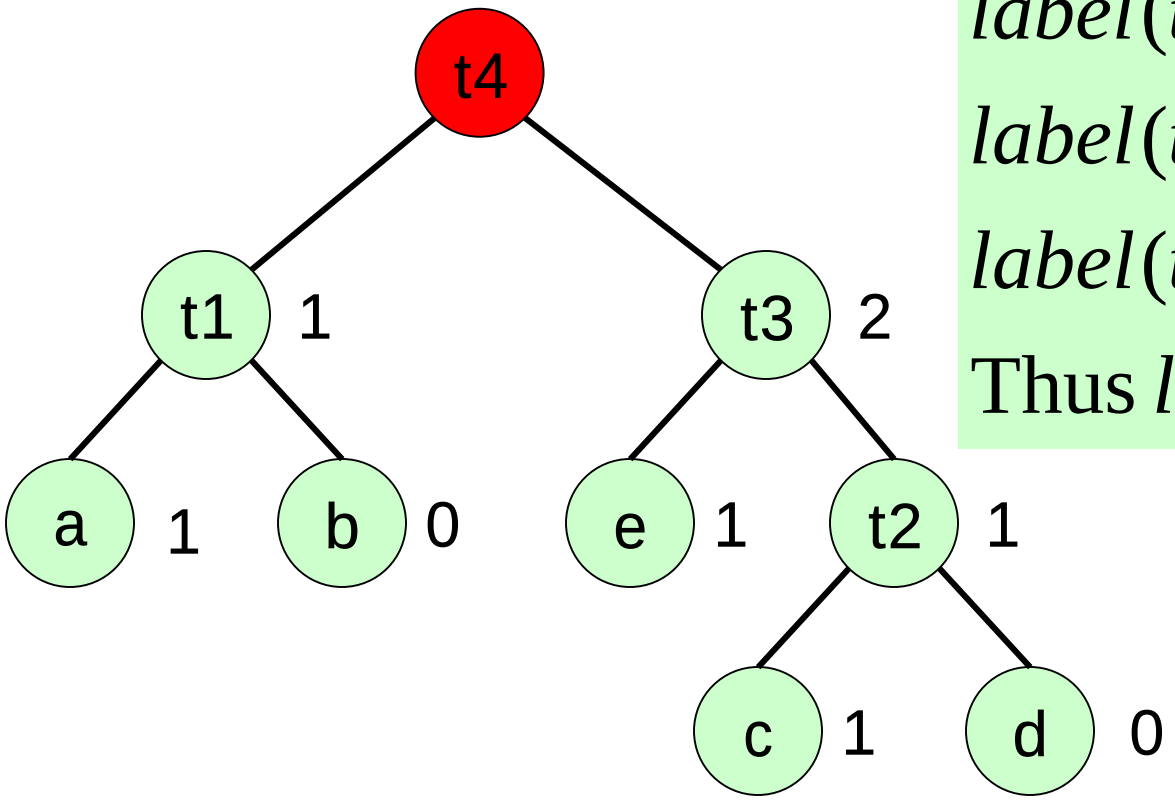
Thus $label(t3) = 2$

Example

Likewise labeling t2



Example



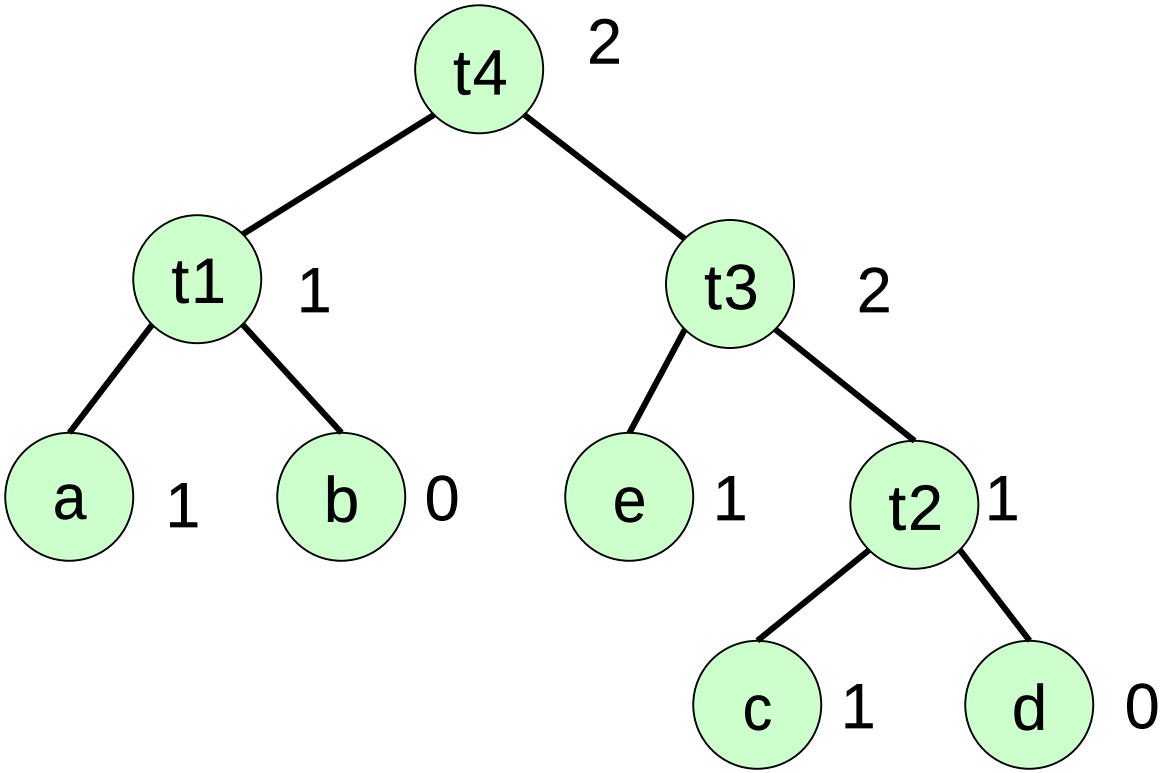
$$label(t2) > label(t1)$$

$$label(t2) + 1 - 1 = 2$$

$$label(t1) + 2 - 1 = 2$$

Thus $label(t4) = 2$

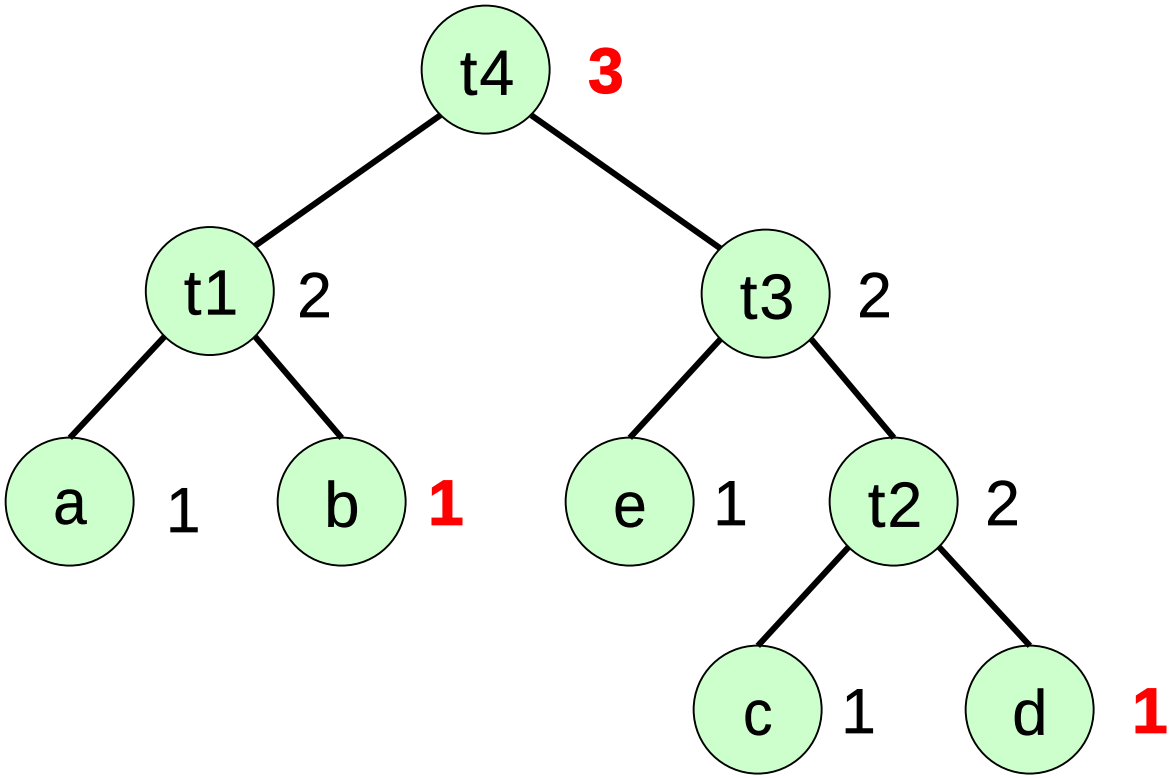
Example



Example-2

When Right & Left both go to registers

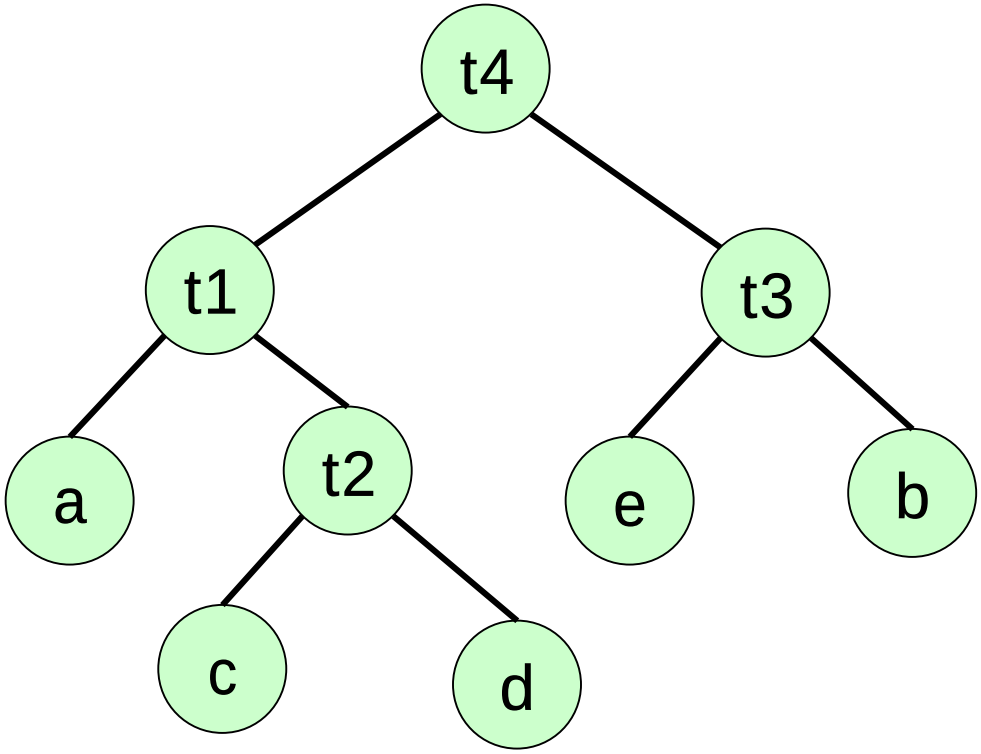
```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```



Example-2

When Right & Left both go to registers

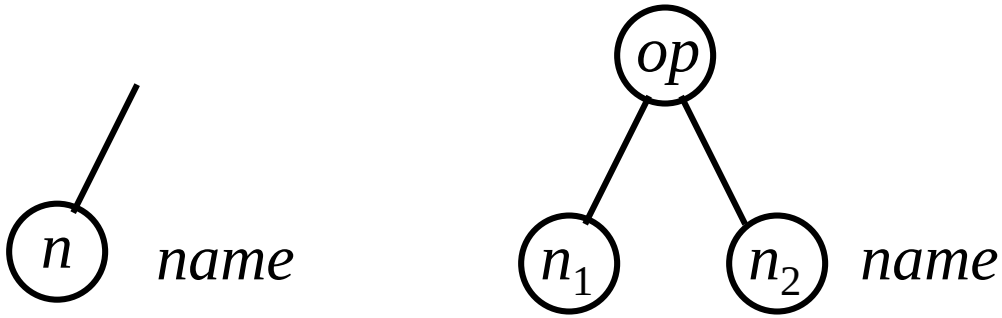
```
t1 := c + d
t2 := a – t1
t3 := e - b
t4 := t1 - t3
```

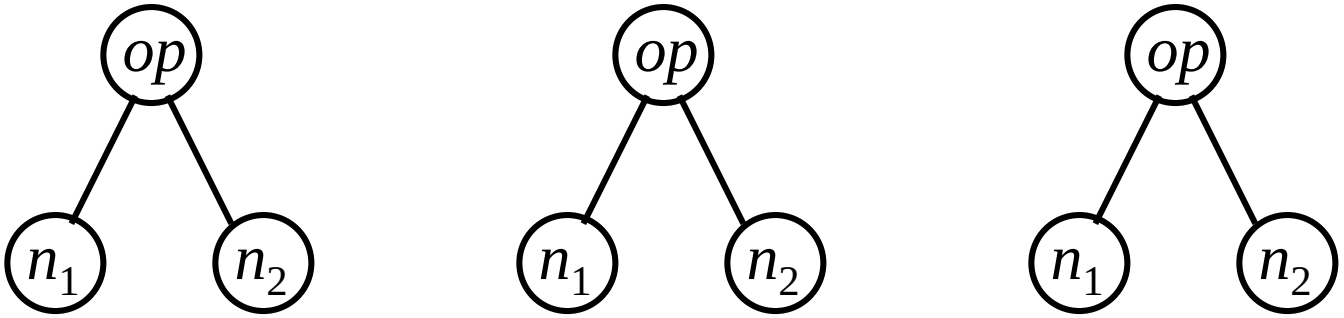


Code Generation From a Labeled Tree

- Use a stack *rstack* to allocate *registers* R0, R1, …, R(*n*-1)
- The value of a tree is always computed in the top register on *rstack*
- The function *swap*(*rstack*) interchanges the top two registers on *rstack*
- Use a stack *tstack* to allocate *temporary memory locations* T0, T1, ...

Cases Analysis





$label(n_1) < label(n_2)$ $label(n_2) \leq label(n_1)$ both labels $\geq r$

The Function *gencode*

procedure *gencode*(*n*);
begin
  if *n* is a left leaf representing operand *name*
        and *n* is the leftmost child of its parent **then**
    print 'MOV' || *name* || ',' || *top*(*rstack*)
  else if *n* is an interior node with operator *op*,
        left child n_1, and right child n_2 **then**
    if label(n_2) = 0 **then** /* case 1 */
    else if $1 \leq$ *label*(n_1) < *label*(n_2) and *label*(n_2) < *r* **then** /* case 2 */
    else if $1 \leq$ *label*(n_2) $\leq$ *label*(n_1) and *label*(n_1) < *r* **then** /* case 3 */
    else /* case 4, both labels $\geq$ *r* */
end

The Function *gencode*

/* case 1 */
begin
    let *name* be the operand represented by n_2;
    gencode(n_1);
    print *op* || *name* || ',' || *top*(*rstack*)
end

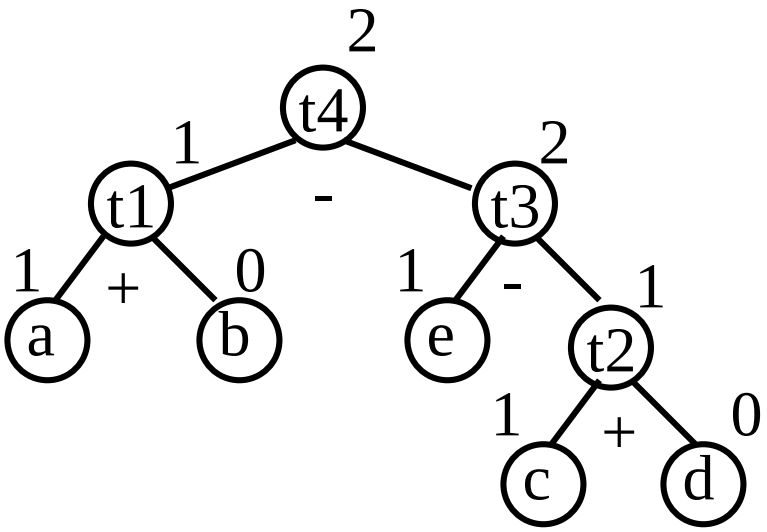
/* case 2 */ (n2>n1)
begin
    swap(*rstack*); *gencode*(n_2);
    R := *pop*(*rstack*); *gencode*(n_1);
    print *op* || *R* || ',' || *top*(*rstack*);
    push(*rstack*, *R*); *swap*(*rstack*);
end

The Function *gencode*

```
/* case 3 */   (n1>=n2)
begin
    gencode(n1);
    R := pop(rstack);   gencode(n2);
    print op || top(rstack) || ',' || R;
    push(rstack, R);
end

/* case 4 */  (n1,n2>r)
begin
    gencode(n2);    T := pop(tstack);
    print 'MOV' || top(rstack) || ',' || T;
    gencode(n1);    push(tstack,  T);
    print op || T || ',' || top(rstack);
end
```

An Example



gencode(t4) [R1, R0] /* 2 */
gencode(t3) [R0, R1] /* 3 */
gencode(e) [R0, R1] /* 0 */
print MOV e, R1
gencode(t2) [R0] /* 1 */
gencode(c) [R0] /* 0 */
print MOV c, R0
print ADD d, R0
print SUB R0, R1
gencode(t1) [R0] /* 1 */
gencode(a) [R0] /* 0 */
print MOV a, R0
print ADD b, R0
print SUB R1, R0