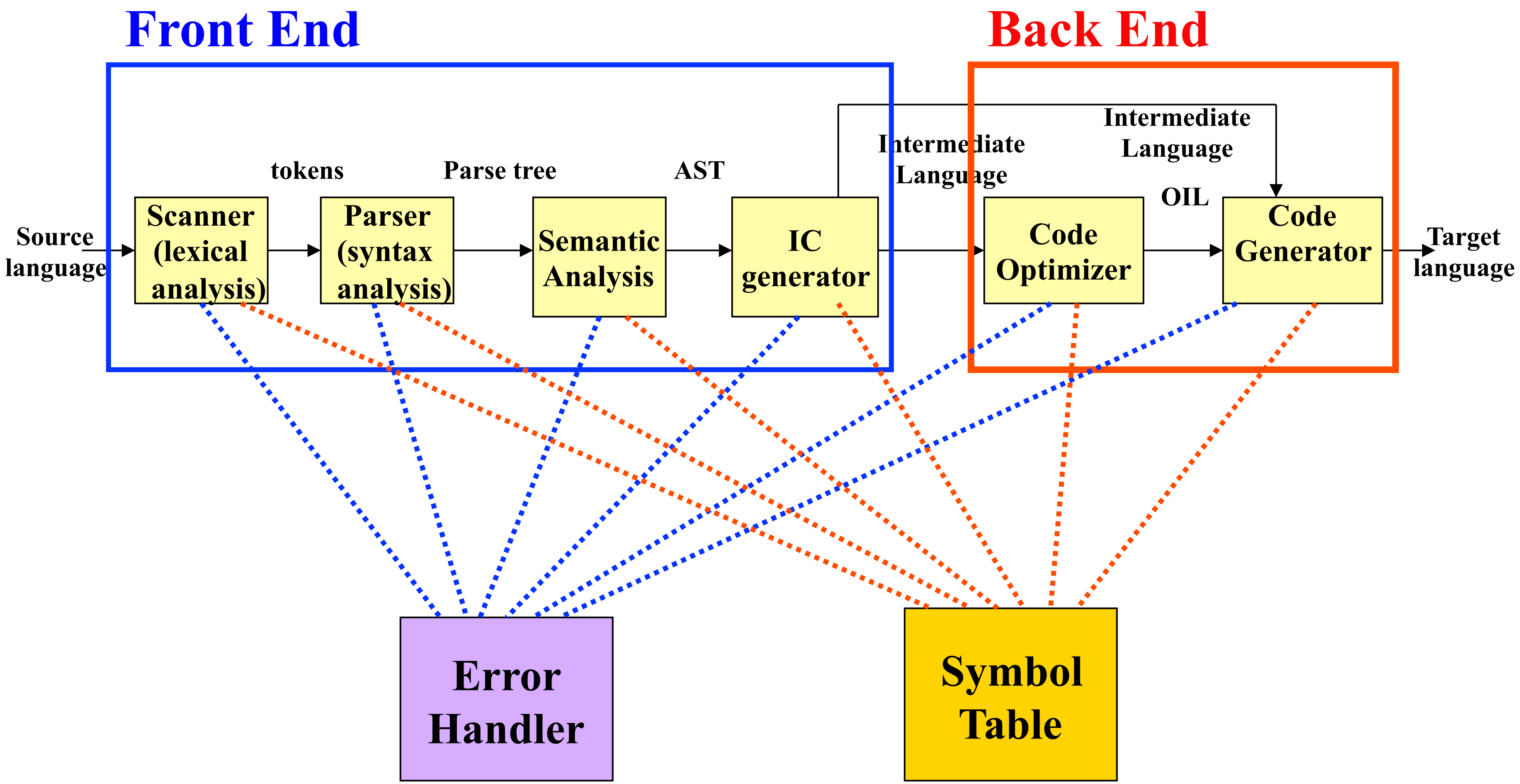
Language Processing Systems

Prof. Mohamed Hamada

Software Engineering Lab.
The University of Aizu
Japan

Compiler Architecture



Code Generation

Code Generation

The code generation problem is the task of mapping intermediate code to machine code.

Requirements:

- Correctness
- Efficiency

Issues:

- Input language: intermediate code (optimized or not)
- Target architecture: must be **well** understood
- Interplay between
  - Instruction Selection
  - Instruction Scheduling (Evaluation order)
  - Register Allocation
  - Memory management

Instruction Selection

- There may be a large number of ‘candidate’ machine instructions for a given IC instruction
  - each has own cost and constraints
  - cost may be influenced by surrounding context
  - different architectures have different needs that must be considered: speed, power constraints, space …

Instruction Scheduling

- Choosing the order of instructions to best utilize resources
- Architecture
  - RISC (pipeline)
  - Vector processing
  - Superscalar and VLIW
- Memory hierarchy
  - Ordering to decrease memory fetching
  - Latency tolerance – doing something when data does have to be fetched

Register Allocation

- Using registers yields a shorter and a faster instructions than using memory locations.
- The use of registers is divided into:
  - Register allocation: selecting variables that will reside in registers
  - Register assignment: picking up a specific register

Memory management

- Memory management is the process of mapping names in the source program to addresses of data objects in run-time memory

- This process is done cooperatively by the front end and the code generator

Target Machine

In this course we will consider the following (virtual) target machine.

General Characteristics

- Byte-addressable with 4-byte words
- N general -purpose registers: R0, R1, ... , R_{n-1}.
- Two-address instructions in the form:

op source, destination

Where op is an operator code, and source and destination are data fields.

Examples of **op** are: MOV, ADD, SUB, MUL, DIV, ...

Target Machine

MOV s, d (move source to destination)

ADD s, d (add source to destination)

SUB s, d (subtract source from destination)

MUL s, d (multiply source to destination)

DIV s, d (divide source by destination)

Target Machine

Example

If destination d = $\boxed{a}$ Then:

MOV s, d	Will cause d =	$\boxed{s}$
ADD s, d	Will cause d =	$\boxed{a + s}$
SUB s, d	Will cause d =	$\boxed{a - s}$
MUL s, d	Will cause d =	$\boxed{a * s}$
DIV s, d	Will cause d =	$\boxed{a / s}$

Target Machine

Storing values: Examples of storing the contents of registers into memory locations can be as follows.

MOV R0, m (stores the contents or register R0 into memory location m)

MOV 4(R0), m (stores the value contents(4+contents(R0)) into memory location m)

MOV *4(R0), m (stores the value contents(contents(4+contents(R0))) into memory location m)

MOV #n, R0 (Loads the constant number n into register R0)

Basic Code Generation

Idea: Deal with the instructions from beginning to end. For each instruction,

- Use registers whenever possible.
- A non-live value in a register can be discarded, freeing that register.

Data Structures:

- Register Descriptor - register status (empty, full) and contents (one or more "values")
- Address descriptor - the location (or locations) where the current value for a variable can be found (register, stack, memory)

.

Data Dependency Graph

To discover data dependencies among statements (or instructions) the compiler uses the data dependency graph.

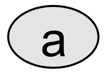
Example **Consider the following set of instructions**

```
(a)  t1 := ld(x);
(b)  t2 := t1 + 4;
(c)  t3 := t1 * 8;
(d)  t4 := t1 - 4;
(e)  t5 := t1 / 2;
(f)  t6 := t2 * t3;
(g)  t7 := t4 - t5;
(h)  t8 := t6 * t7;
(i)  st(y,t8);
```

The data dependency graph for it can be constructed as follows:

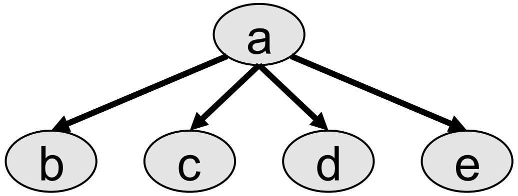
Data Dependency Graph

```
(a)  t1 := ld(x);
(b)  t2 := t1 + 4;
(c)  t3 := t1 * 8;
(d)  t4 := t1 - 4;
(e)  t5 := t1 / 2;
(f)  t6 := t2 * t3;
(g)  t7 := t4 - t5;
(h)  t8 := t6 * t7;
(i)  st(y,t8);
```



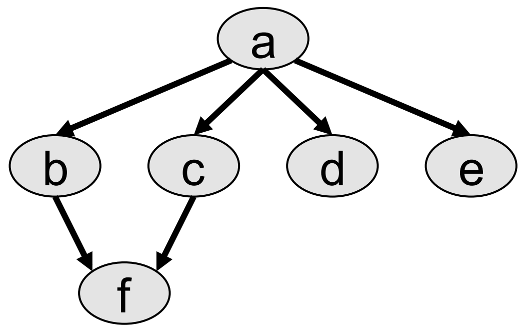
Data Dependency Graph

```
(a)  t1 := ld(x);
(b)  t2 := t1 + 4;
(c)  t3 := t1 * 8;
(d)  t4 := t1 - 4;
(e)  t5 := t1 / 2;
(f)  t6 := t2 * t3;
(g)  t7 := t4 - t5;
(h)  t8 := t6 * t7;
(i)  st(y,t8);
```



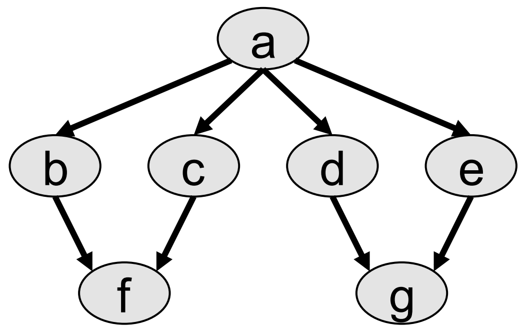
Data Dependency Graph

```
(a)  t1 := ld(x);
(b)  t2 := t1 + 4;
(c)  t3 := t1 * 8;
(d)  t4 := t1 - 4;
(e)  t5 := t1 / 2;
(f)  t6 := t2 * t3;
(g)  t7 := t4 - t5;
(h)  t8 := t6 * t7;
(i)  st(y,t8);
```



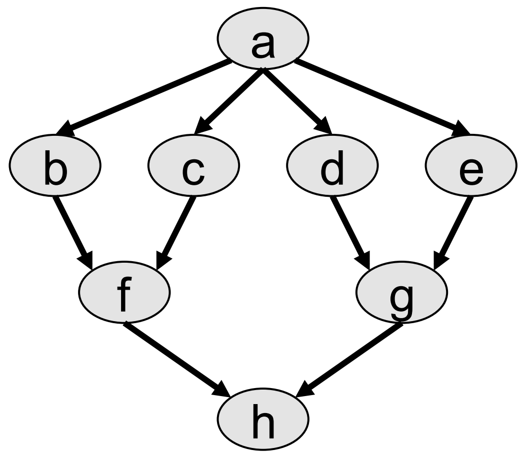
Data Dependency Graph

```
(a)  t1 := ld(x);
(b)  t2 := t1 + 4;
(c)  t3 := t1 * 8;
(d)  t4 := t1 - 4;
(e)  t5 := t1 / 2;
(f)  t6 := t2 * t3;
(g)  t7 := t4 - t5;
(h)  t8 := t6 * t7;
(i)  st(y,t8);
```



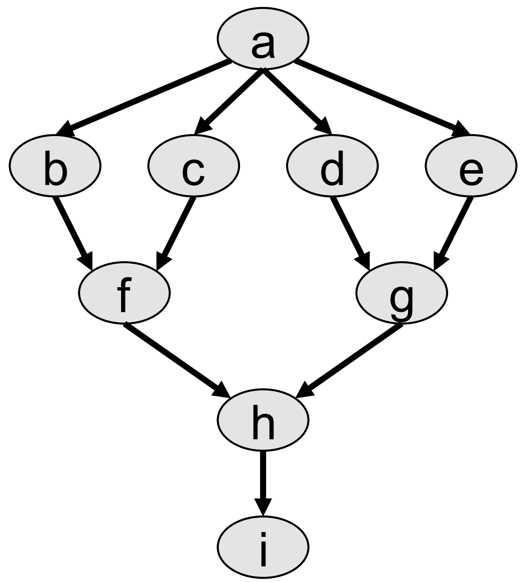
Data Dependency Graph

```
(a)  t1 := ld(x);
(b)  t2 := t1 + 4;
(c)  t3 := t1 * 8;
(d)  t4 := t1 - 4;
(e)  t5 := t1 / 2;
(f)  t6 := t2 * t3;
(g)  t7 := t4 - t5;
(h)  t8 := t6 * t7;
(i)  st(y,t8);
```



Data Dependency Graph

```
(a)  t1 := ld(x);
(b)  t2 := t1 + 4;
(c)  t3 := t1 * 8;
(d)  t4 := t1 - 4;
(e)  t5 := t1 / 2;
(f)  t6 := t2 * t3;
(g)  t7 := t4 - t5;
(h)  t8 := t6 * t7;
(i)  st(y,t8);
```



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.

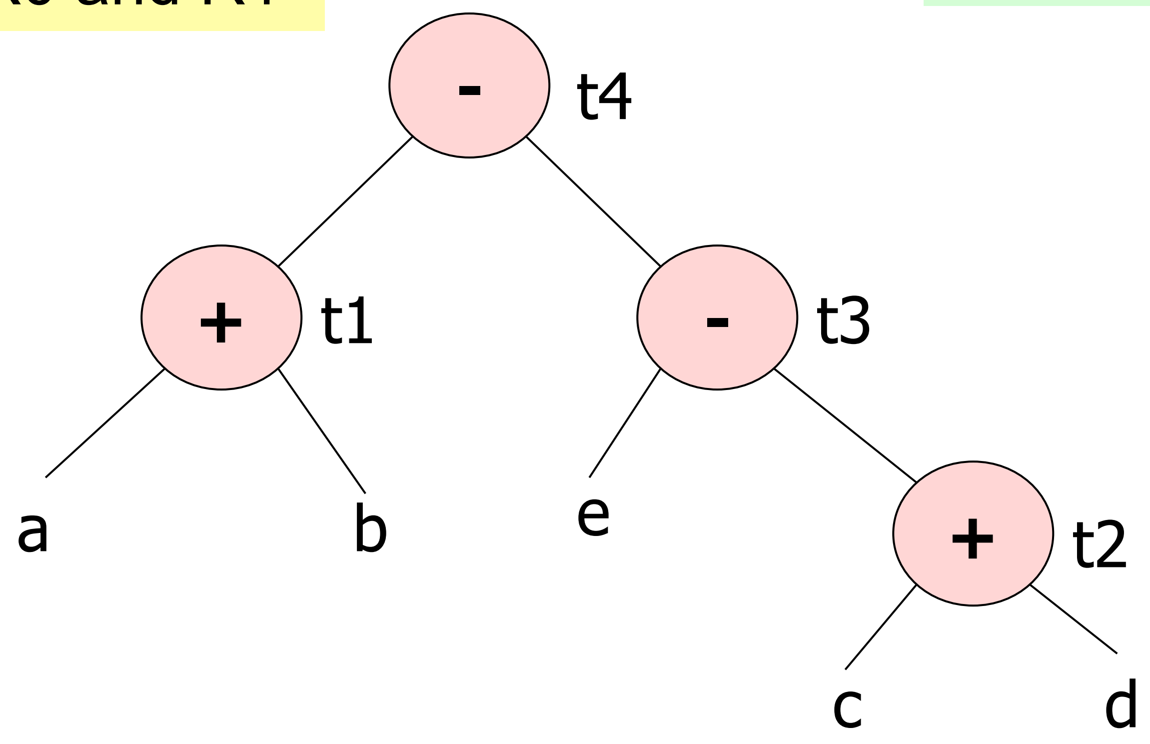
Example

Consider the expression: (a + b) – (e – (c + d))

And its corresponding intermediate code:

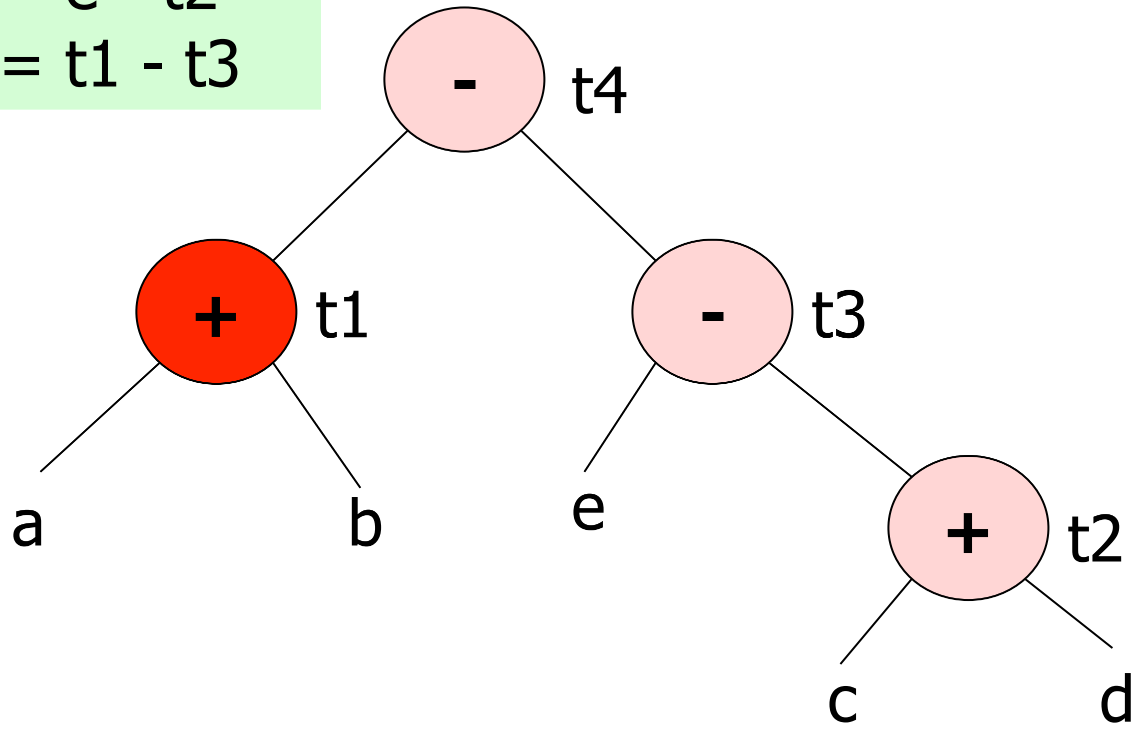
```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```

Generate code for a machine with two Registers R0 and R1



Example

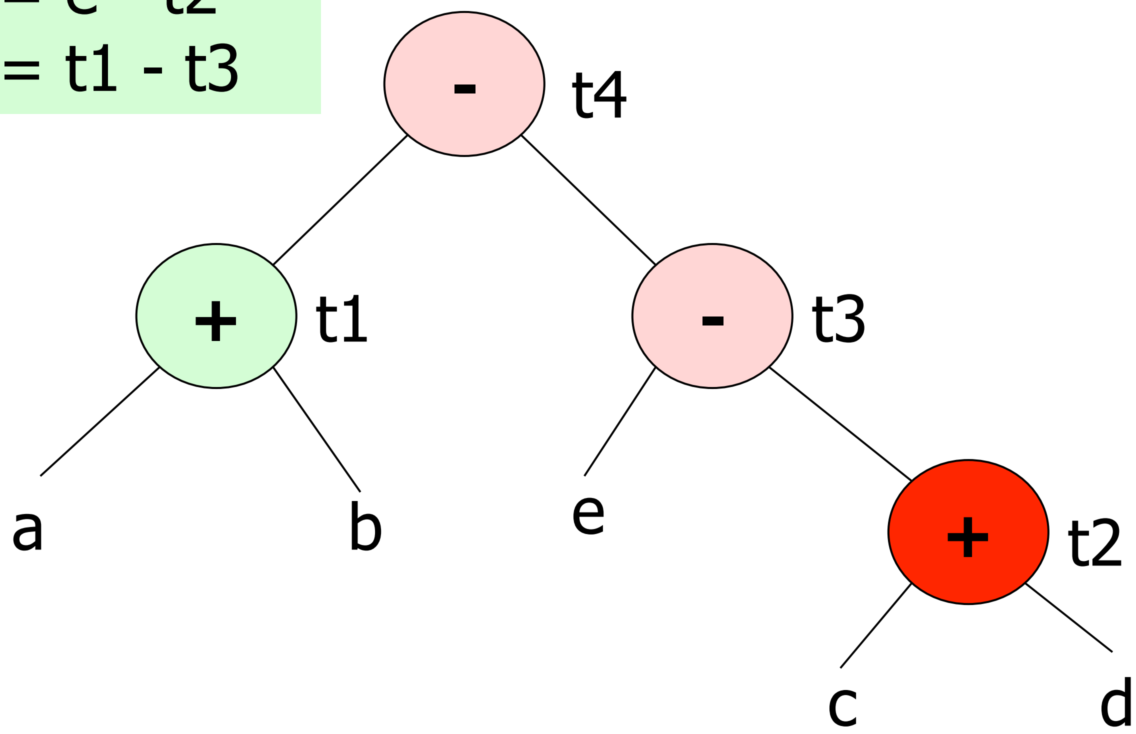
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3



MOV a, R0
ADD b, R0

Example

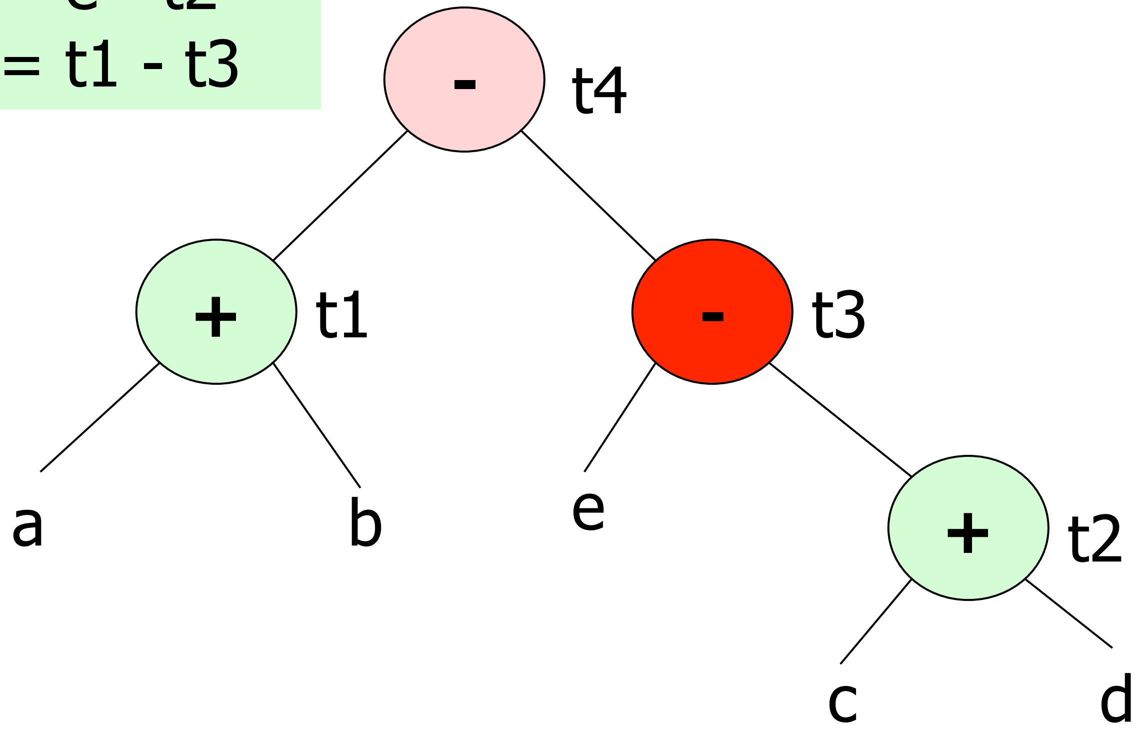
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3



MOV	a, R0
ADD	b, R0
MOV	c, R1
ADD	d, R1

Example

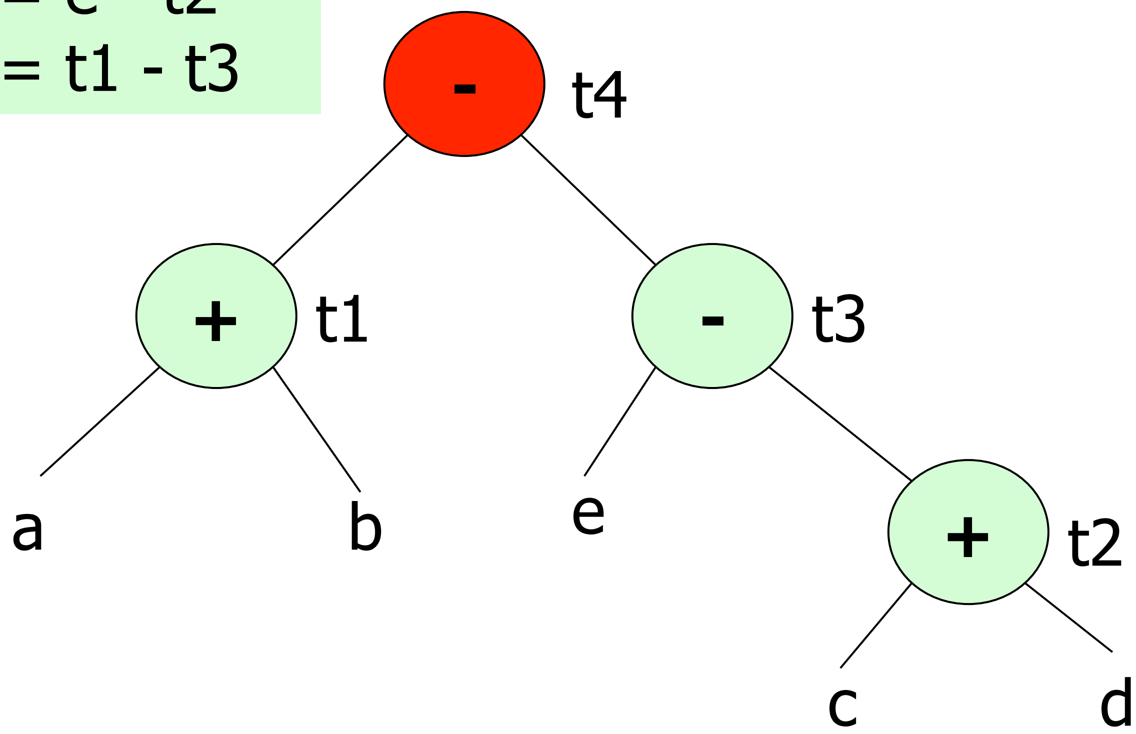
```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```




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

Example

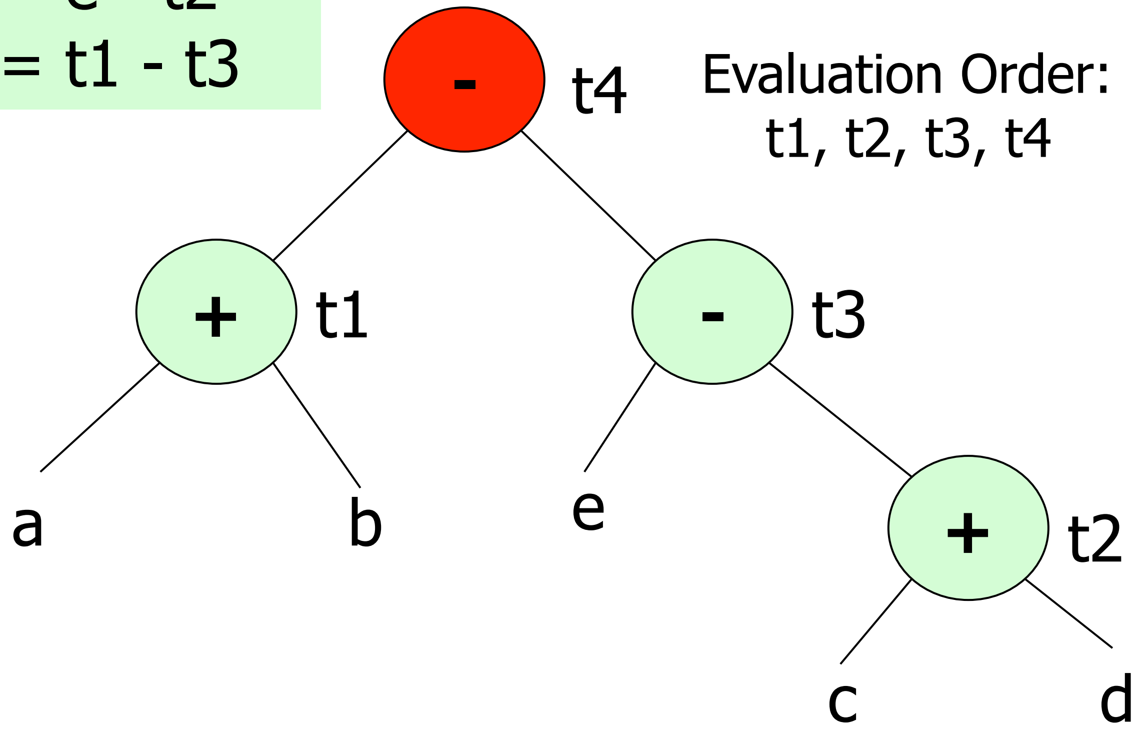
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3




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

Example

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



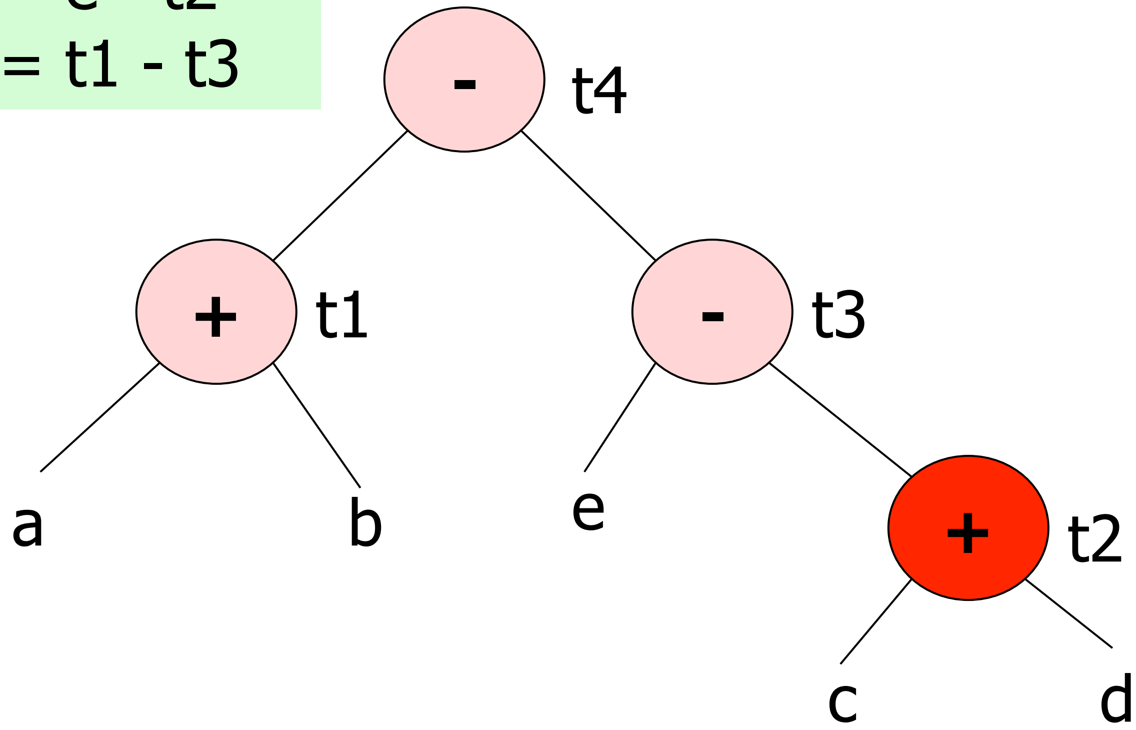
Evaluation Order:
t1, t2, t3, t4

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

10 instructions

Example (can we do better?)

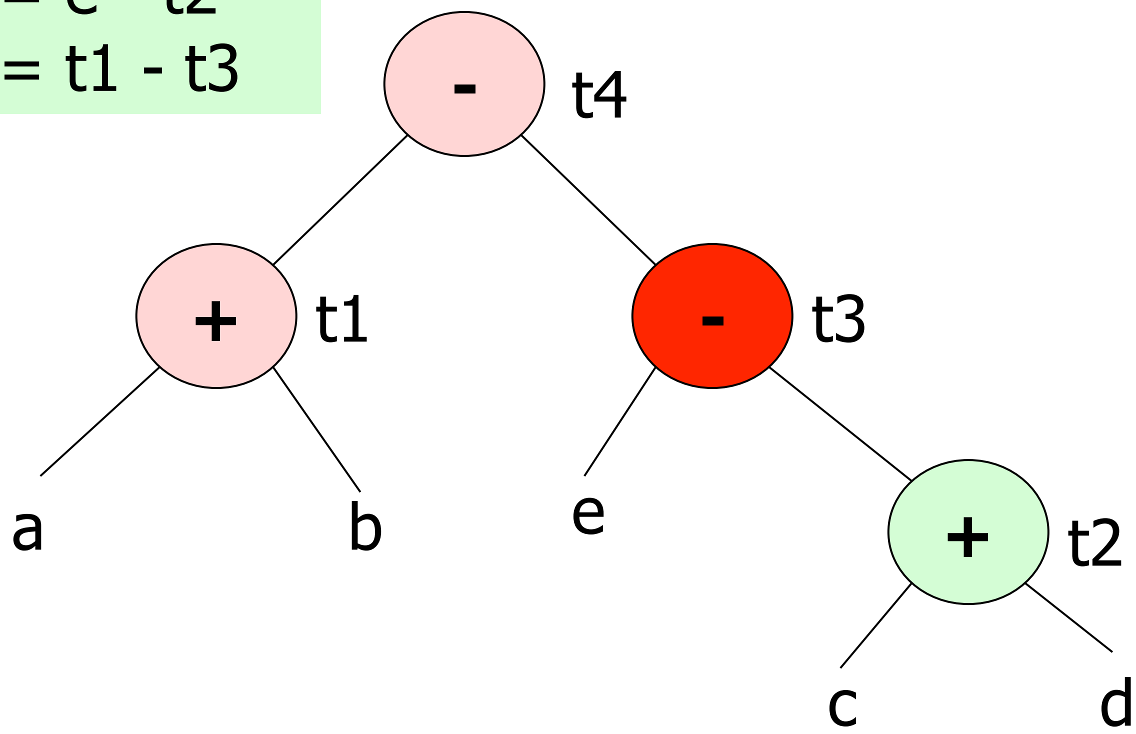
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3



MOV c, R0
ADD d, R0

Example (can we do better?)

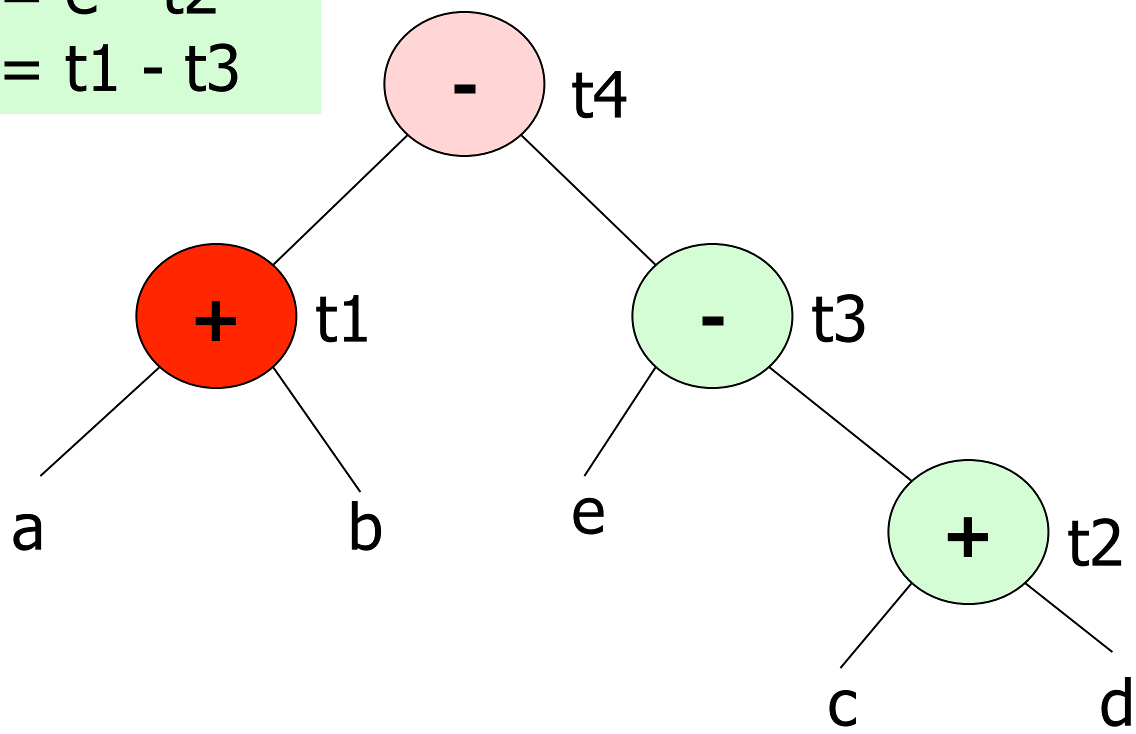
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3



MOV	c, R0
ADD	d, R0
MOV	e, R1
SUB	R0,R1

Example (can we do better?)

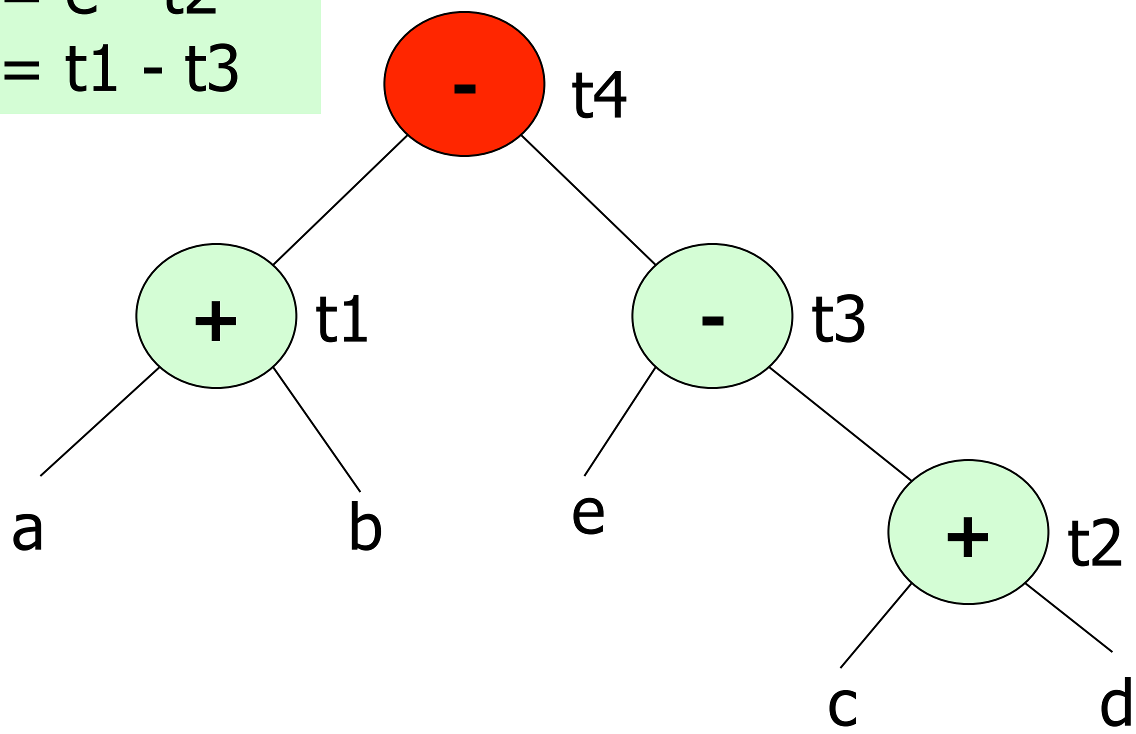
```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```




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

Example
(can we do better?)

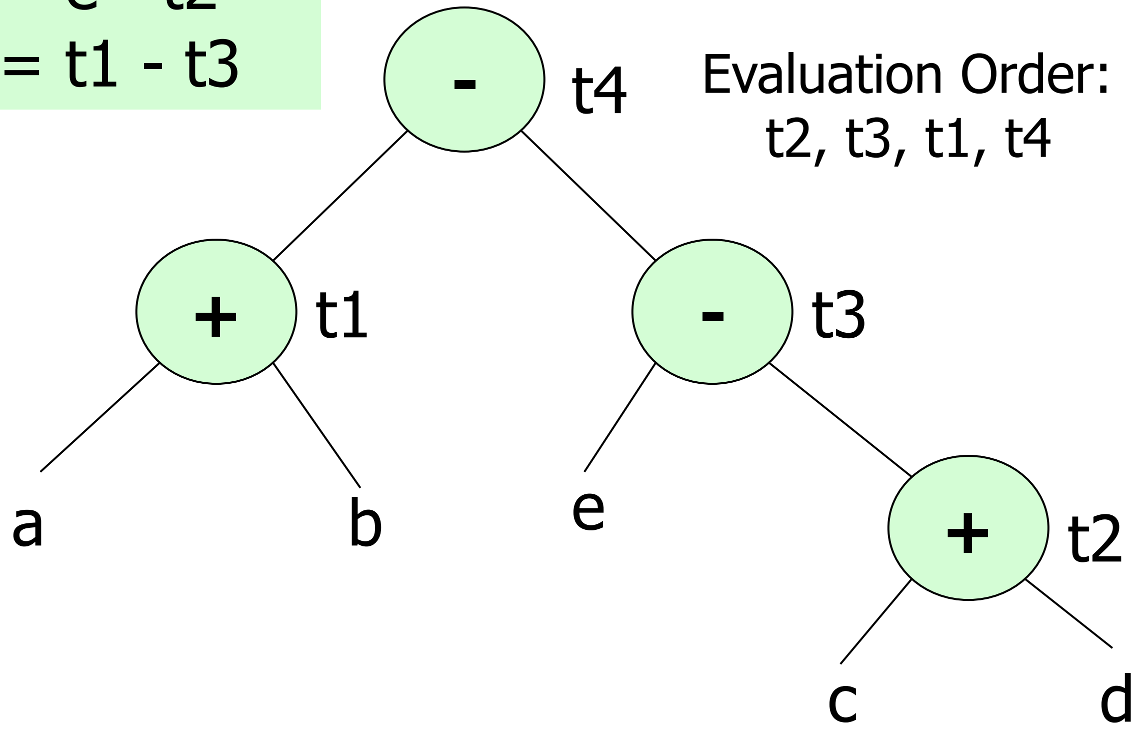
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3




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

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

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



Evaluation Order:
t2, t3, t1, t4

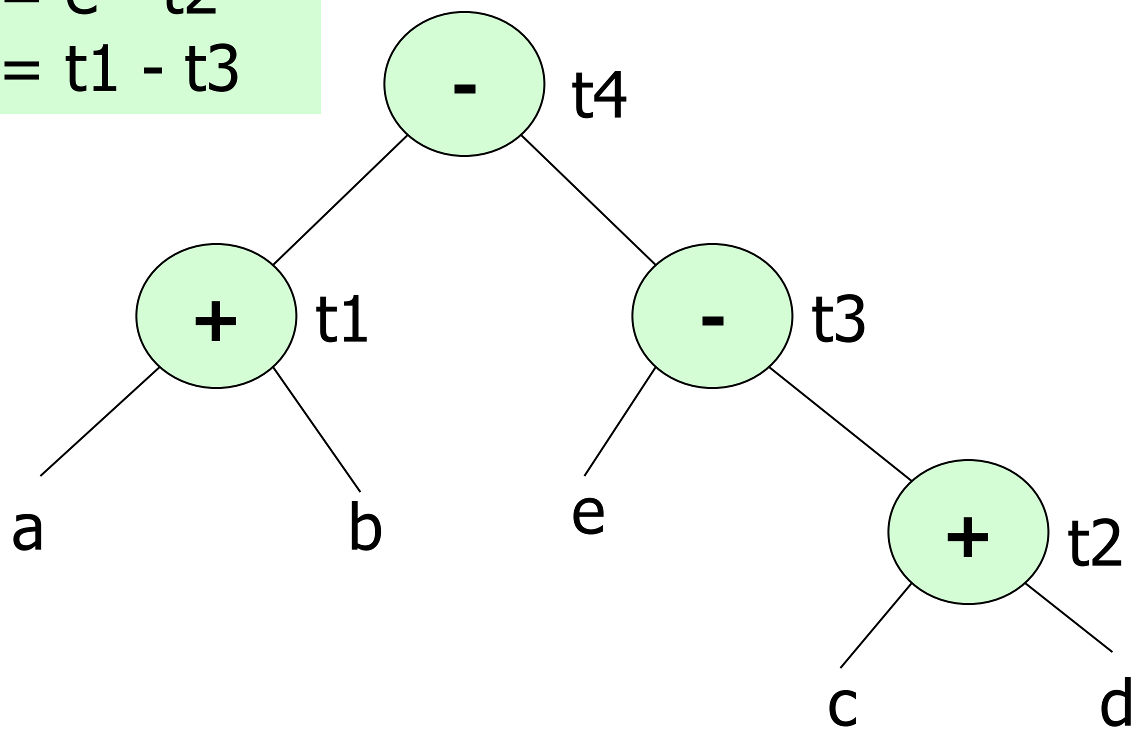
```
MOV     c, R0
ADD     d, R0
MOV     e, R1
SUB     R0, R1
MOV     a, R0
ADD     b, R0
SUB     R1, R0
MOV     R0, t4
```

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).



Heuristic Node Listing Algorithm for a DAG

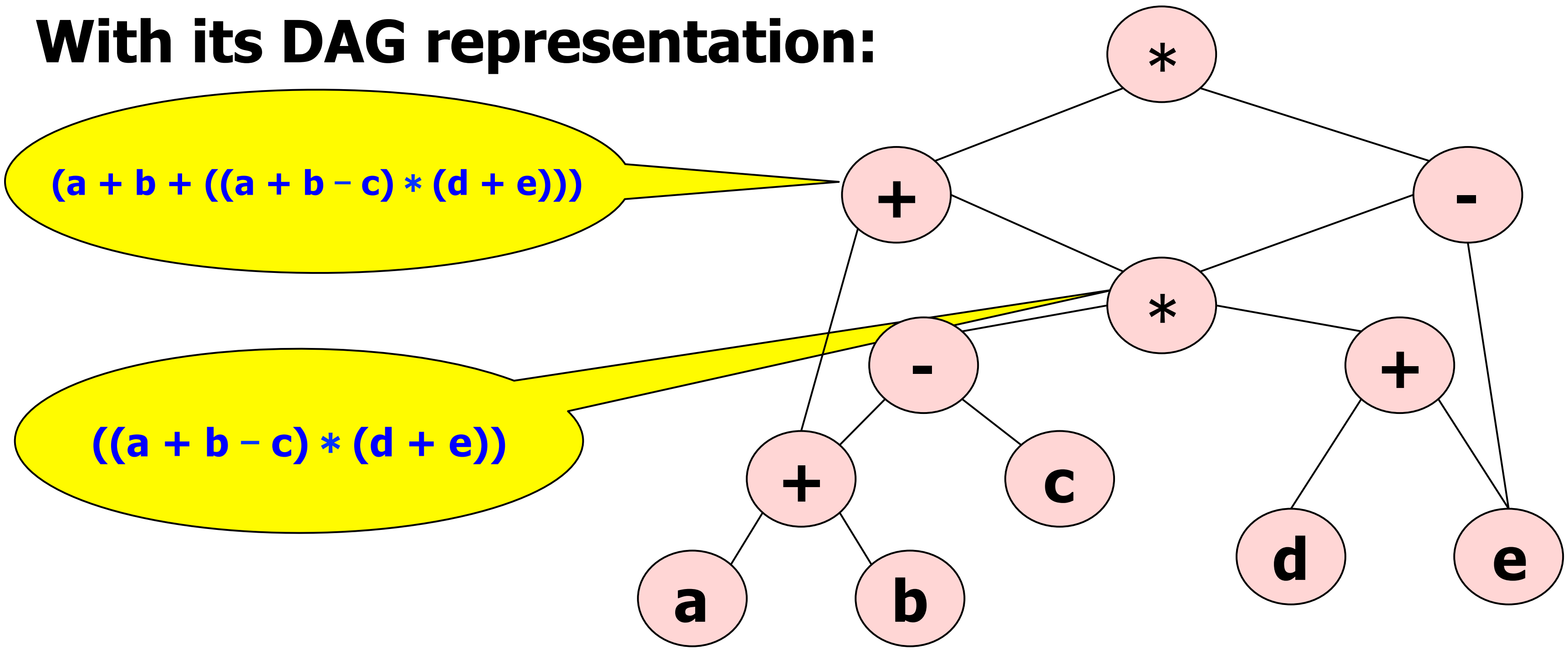
```
(1) while unlisted interior nodes remain
(2)       select an unlisted node n, all of whose parents
          have been listed;
(3)       list n;
(4)       while the leftmost child m of n has no unlisted parents,
                 and is not a leaf node do
                 /* since n was just listed,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

Node Listing Example

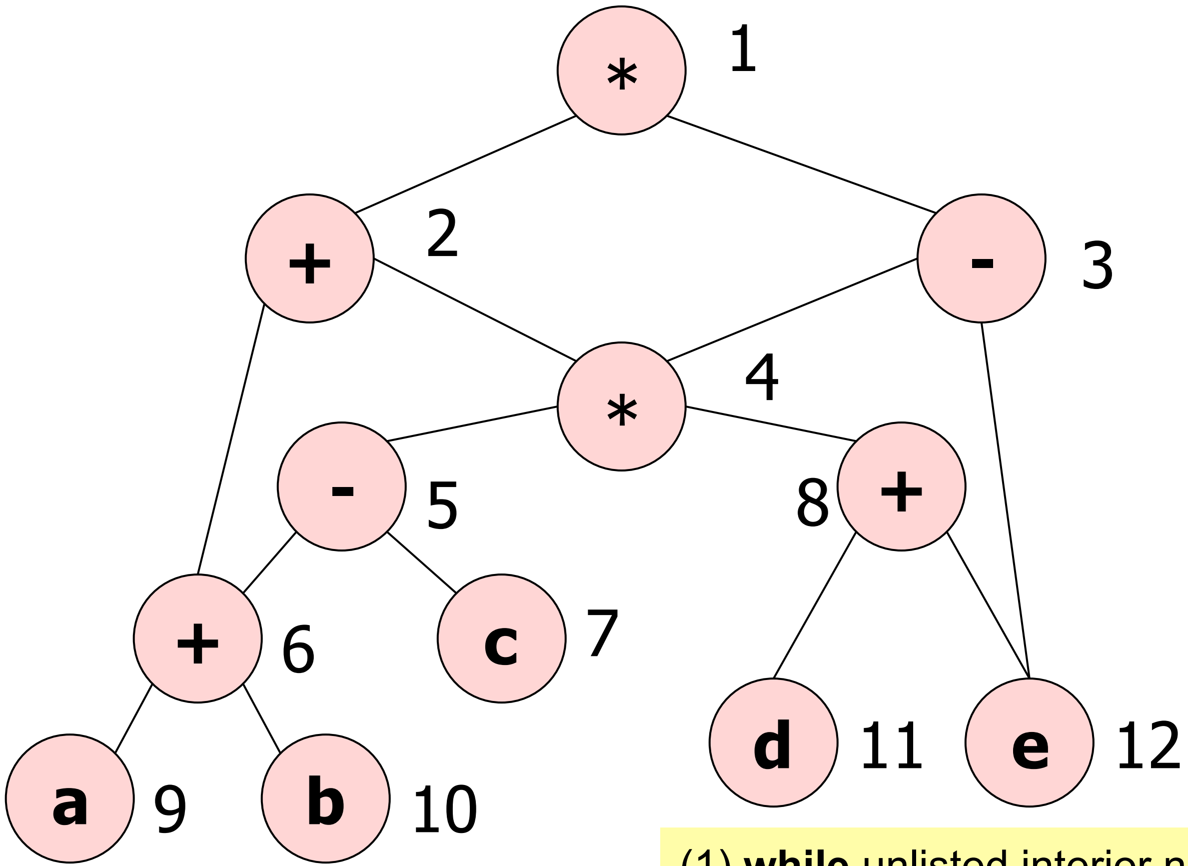
Consider the expression:

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

With its DAG representation:

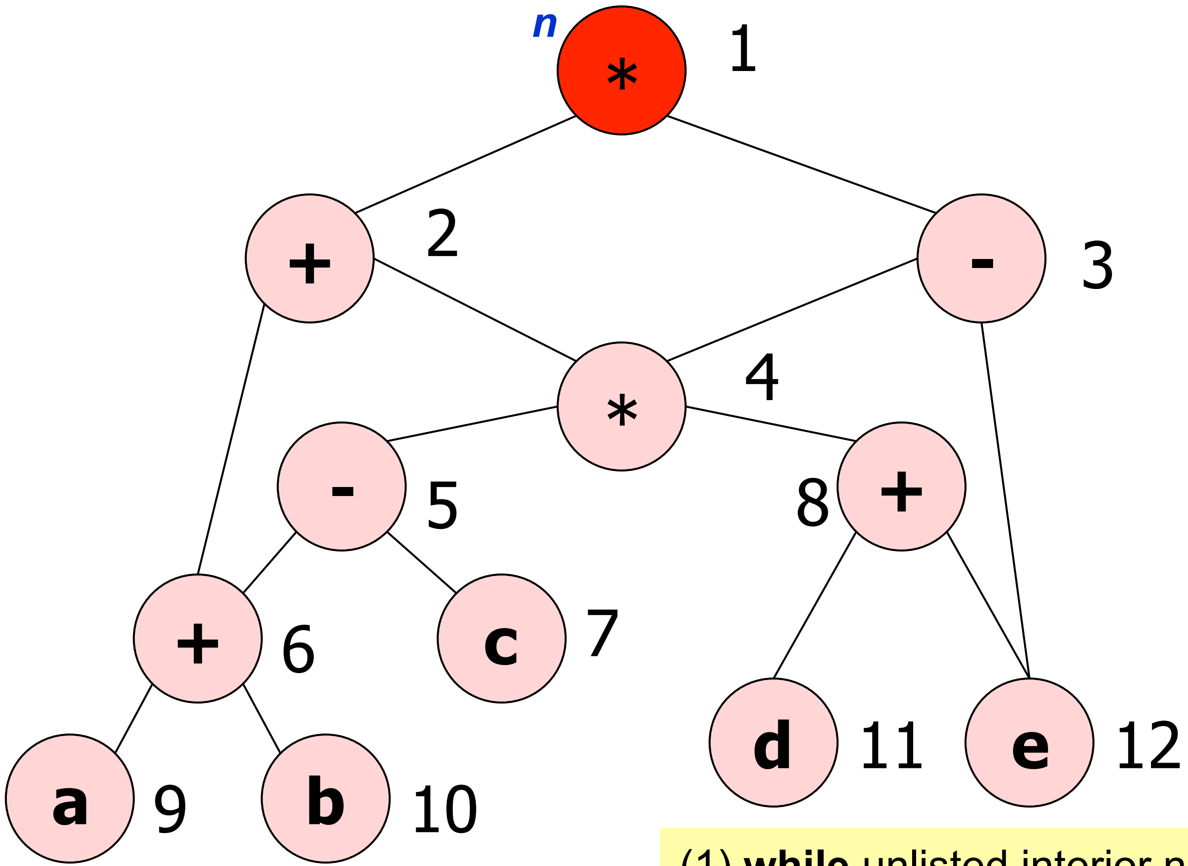


Node Listing Example



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

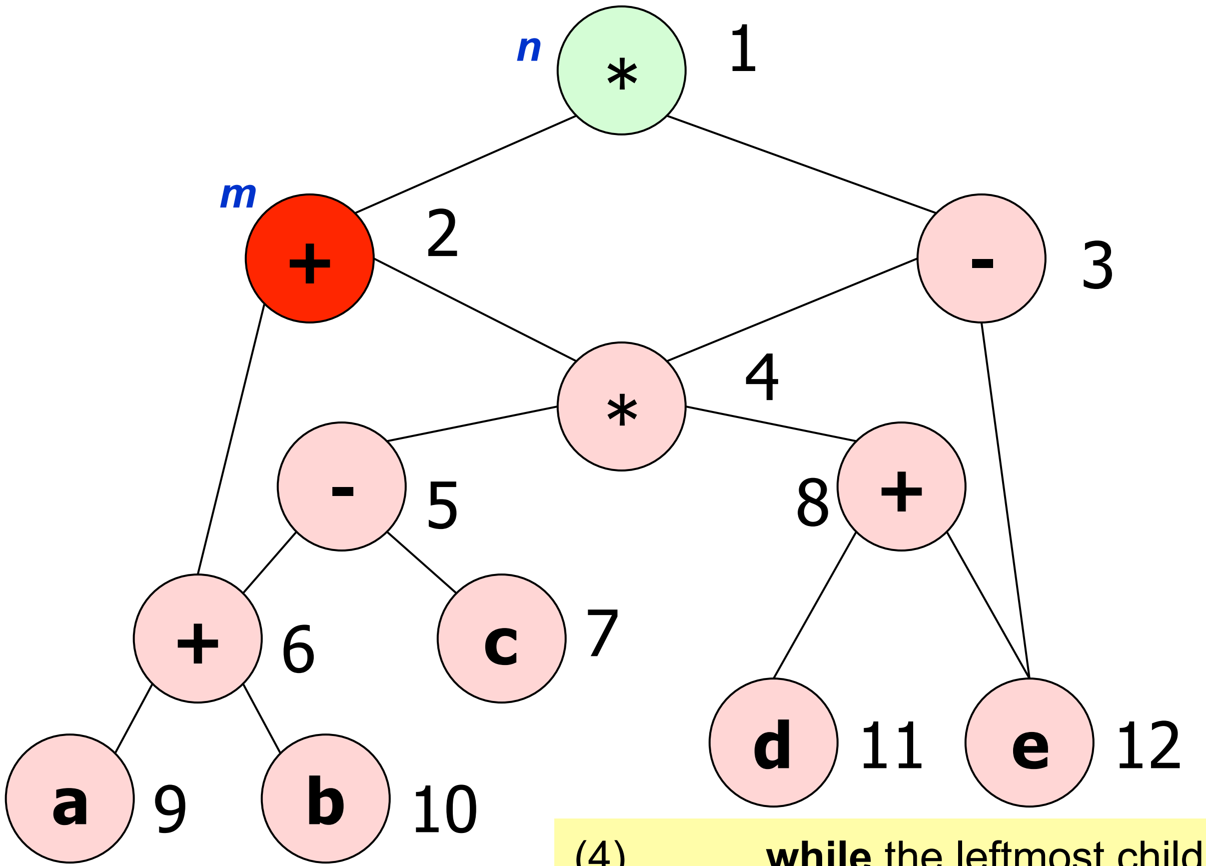
Node Listing Example



List:
1

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

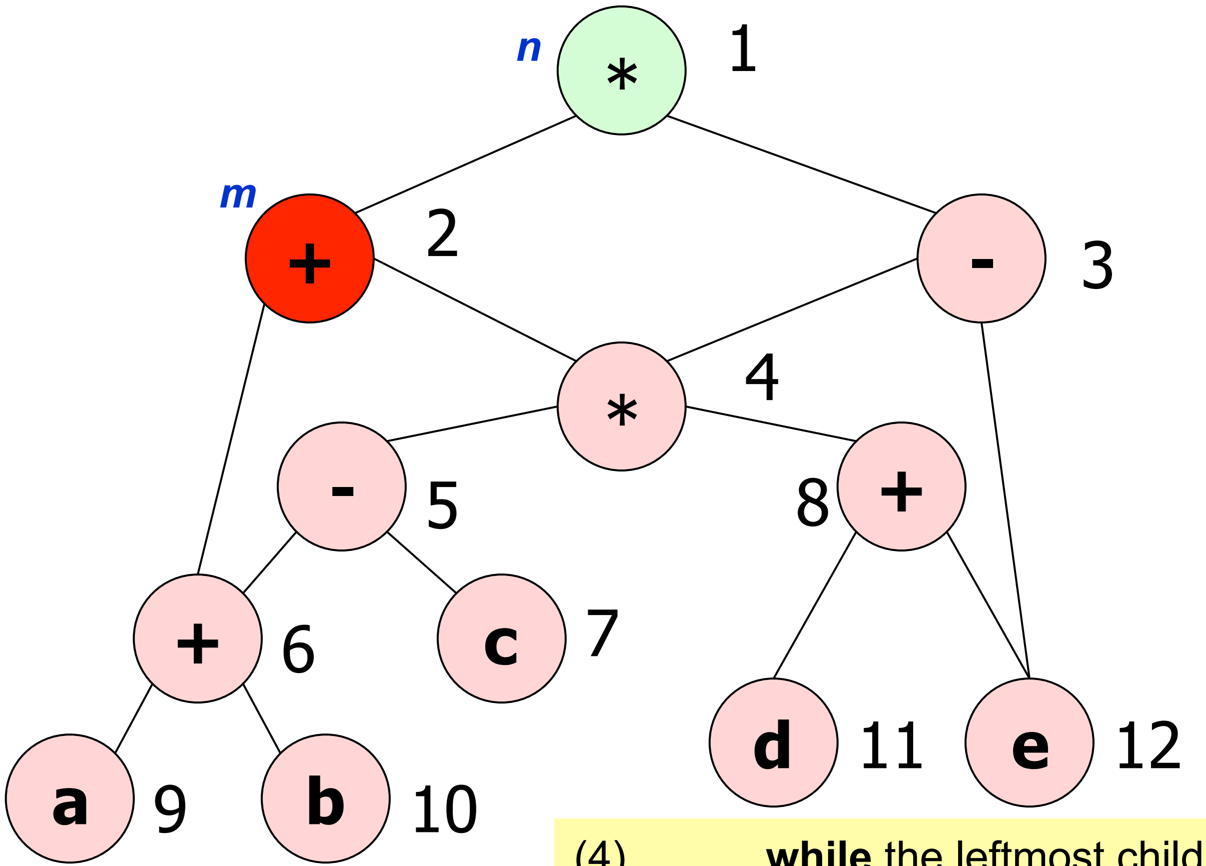
Node Listing Example



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***;

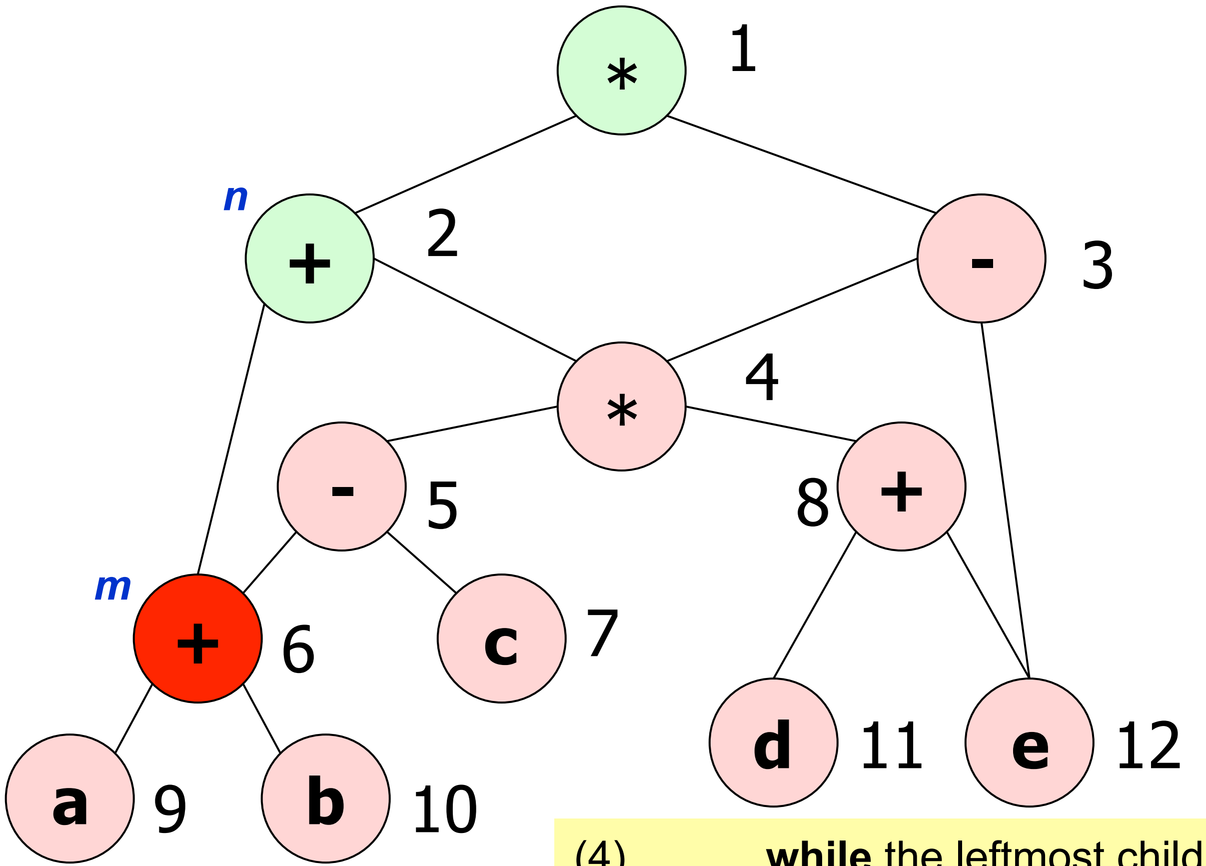
Node Listing Example



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***;

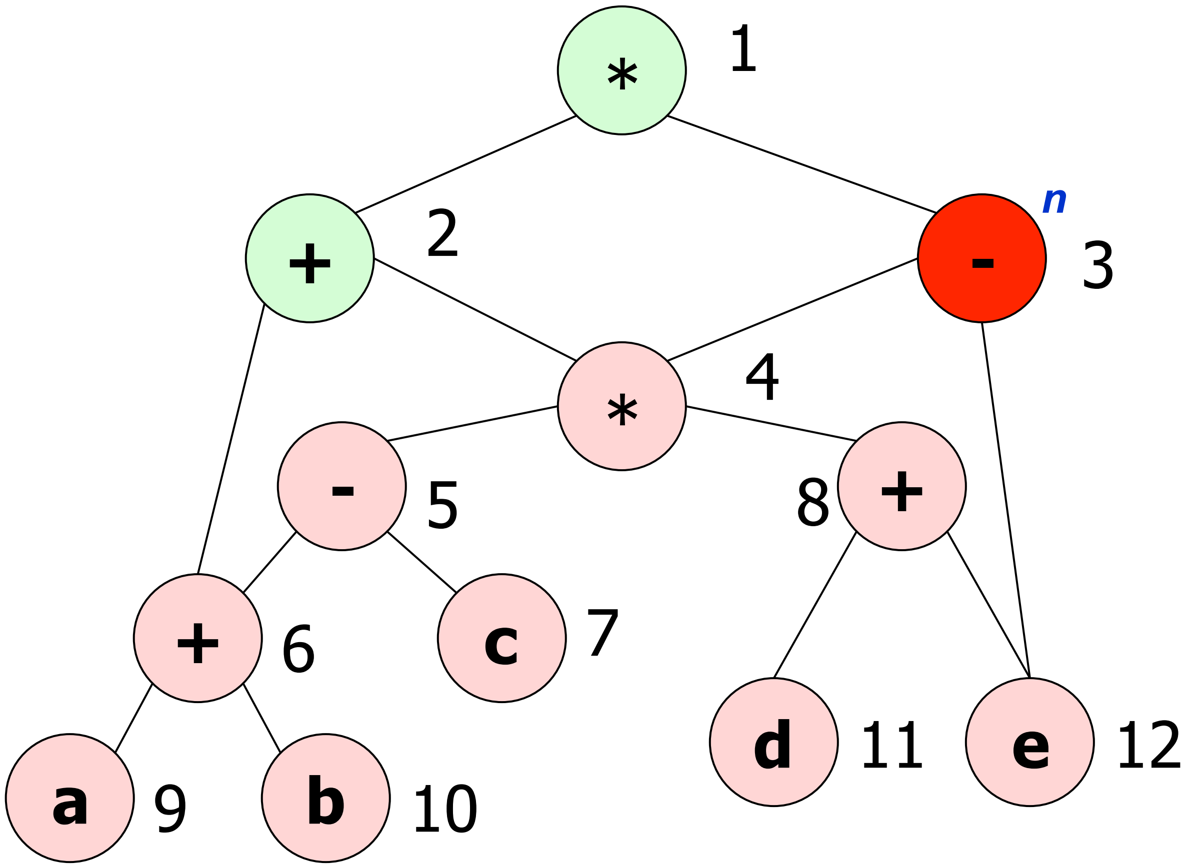
Node Listing Example



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***;

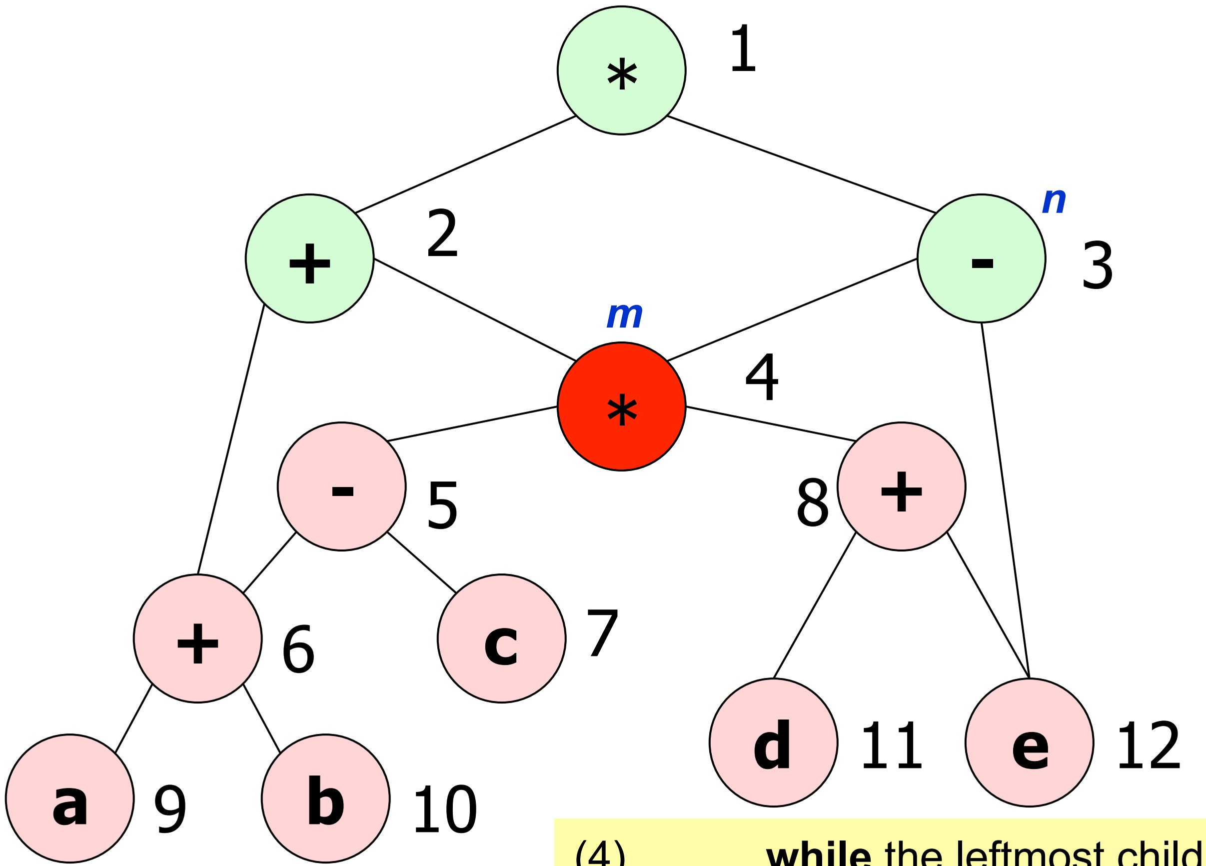
Node Listing Example



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***;

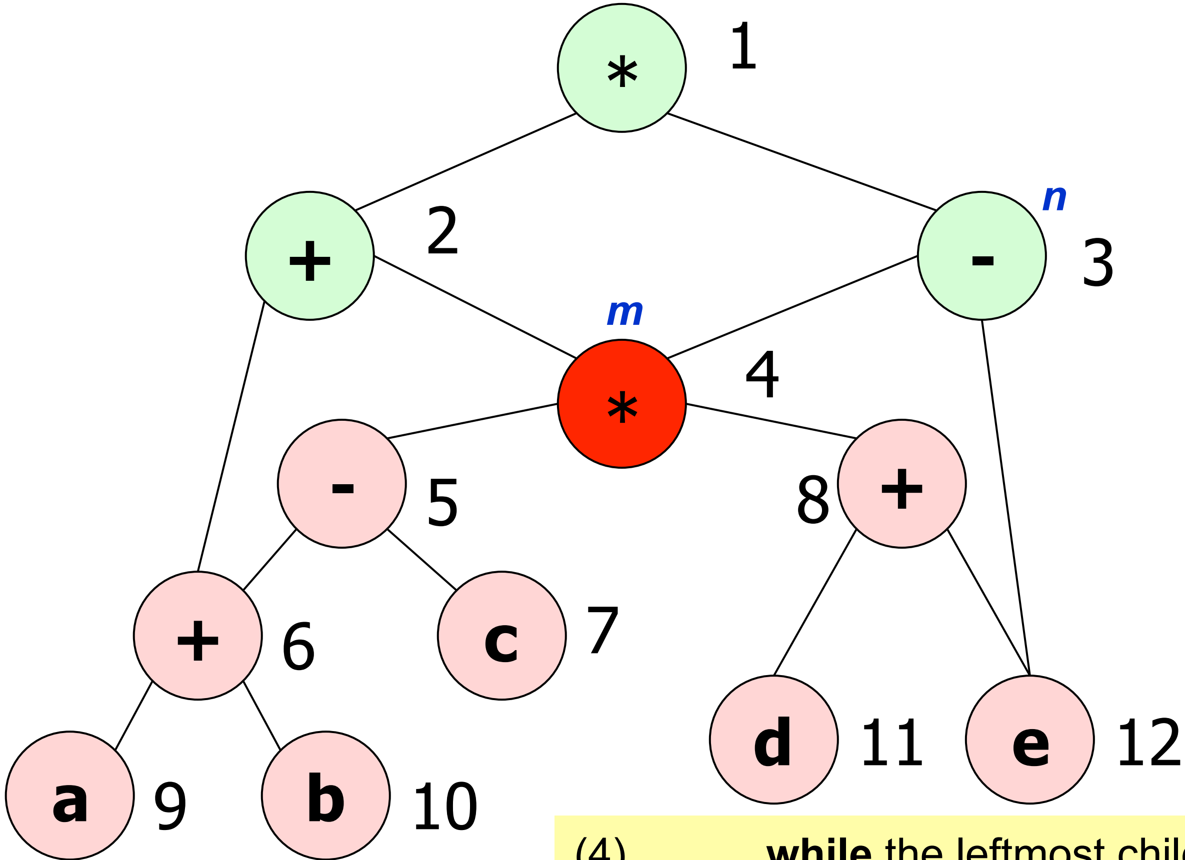
Node Listing Example



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***;

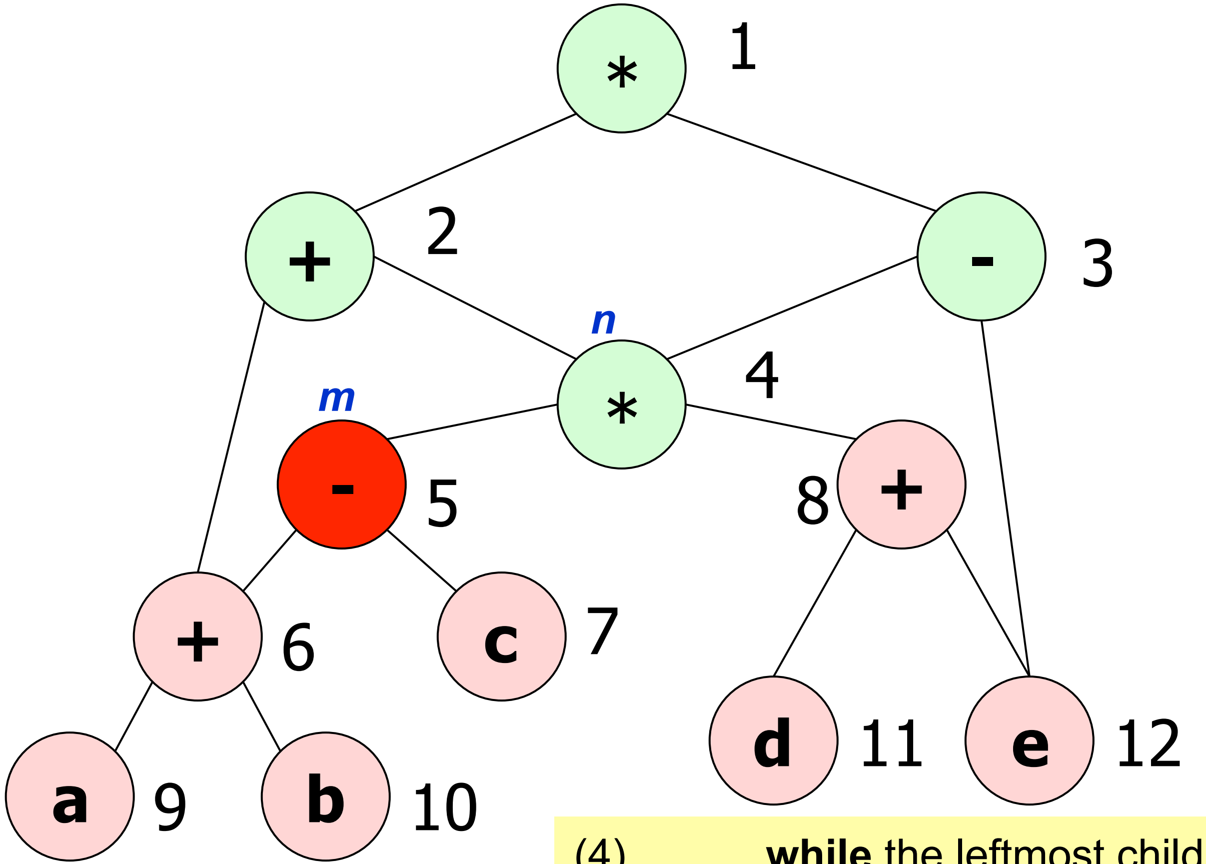
Node Listing Example



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;
```

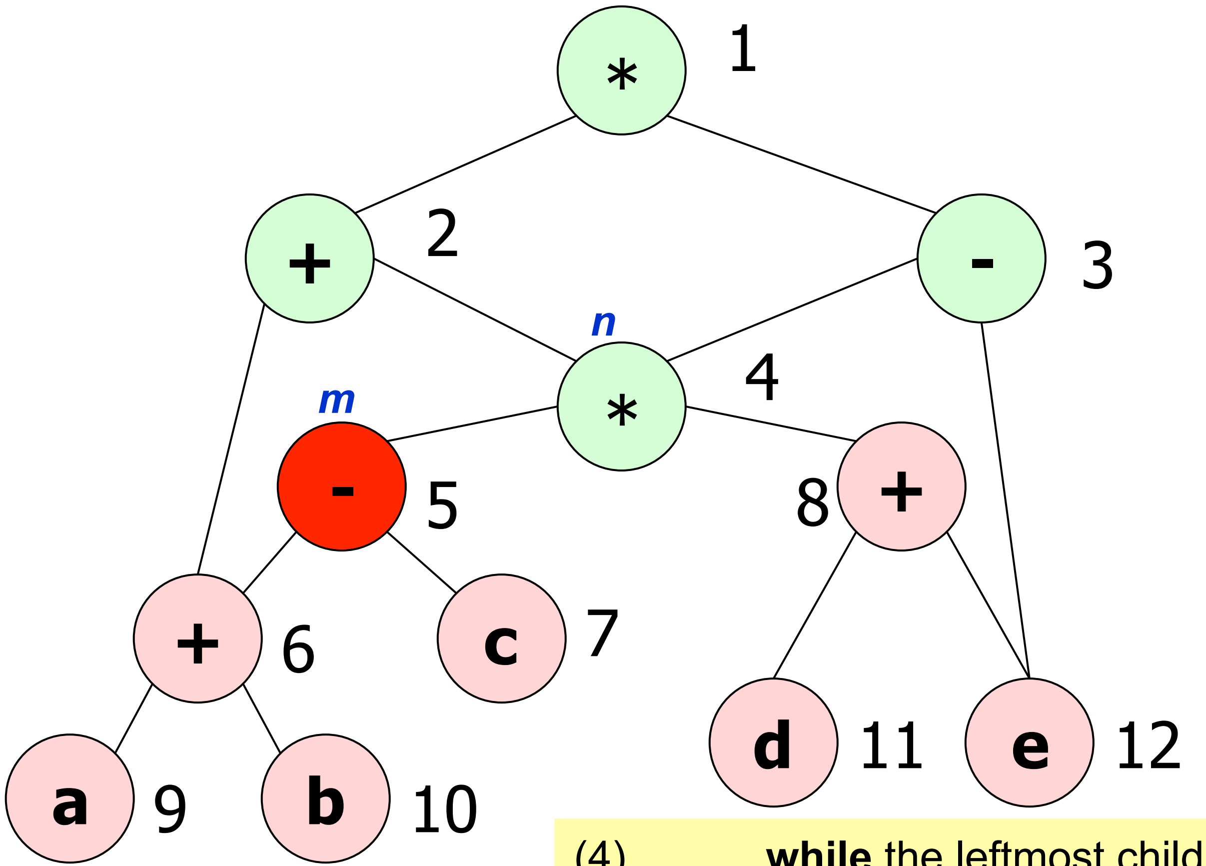
Node Listing Example



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***;

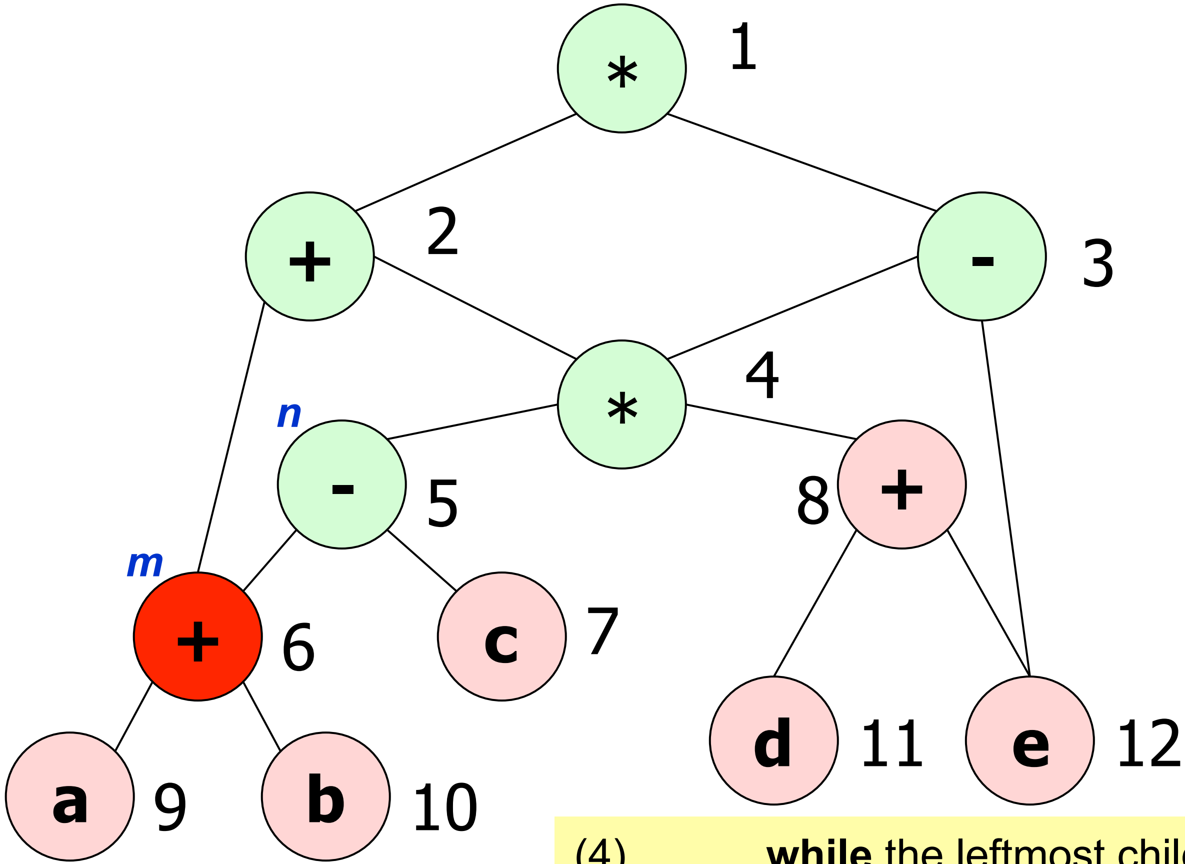
Node Listing Example



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***;

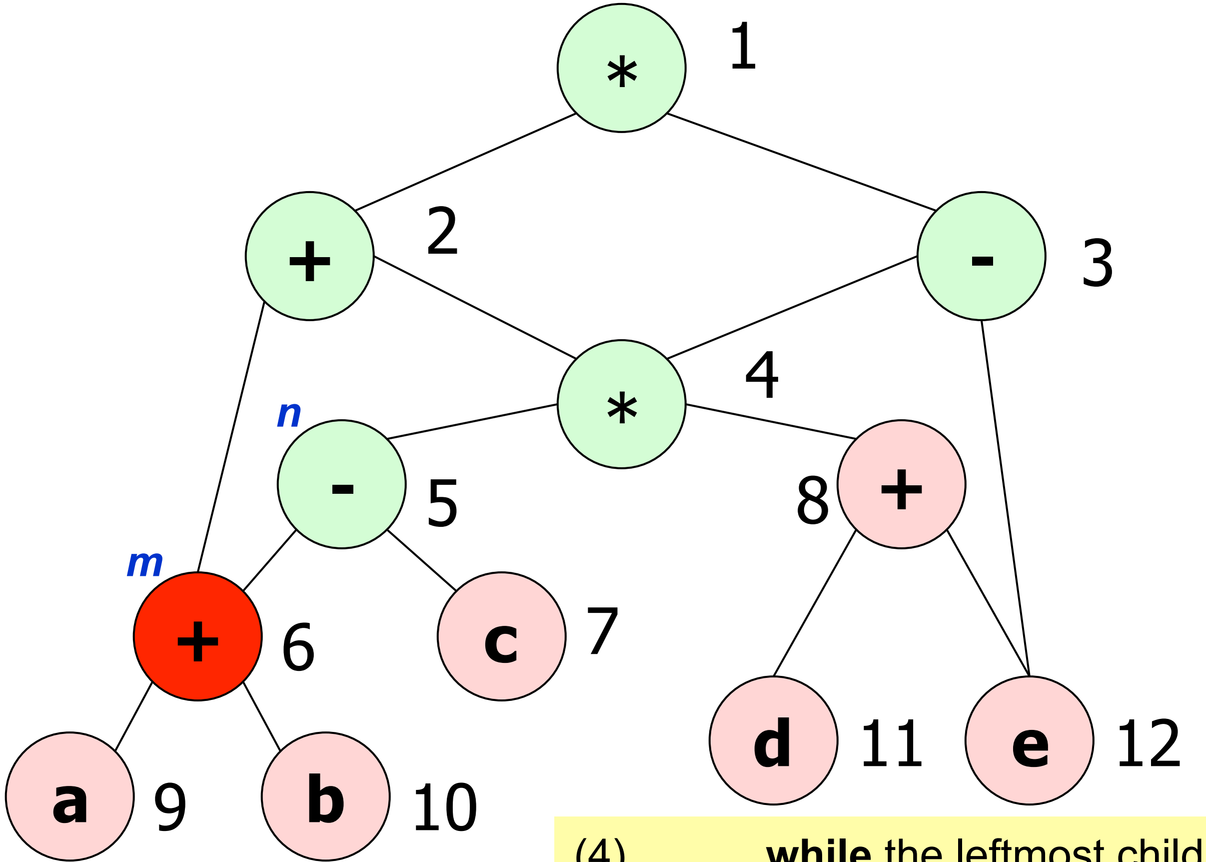
Node Listing Example



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***;

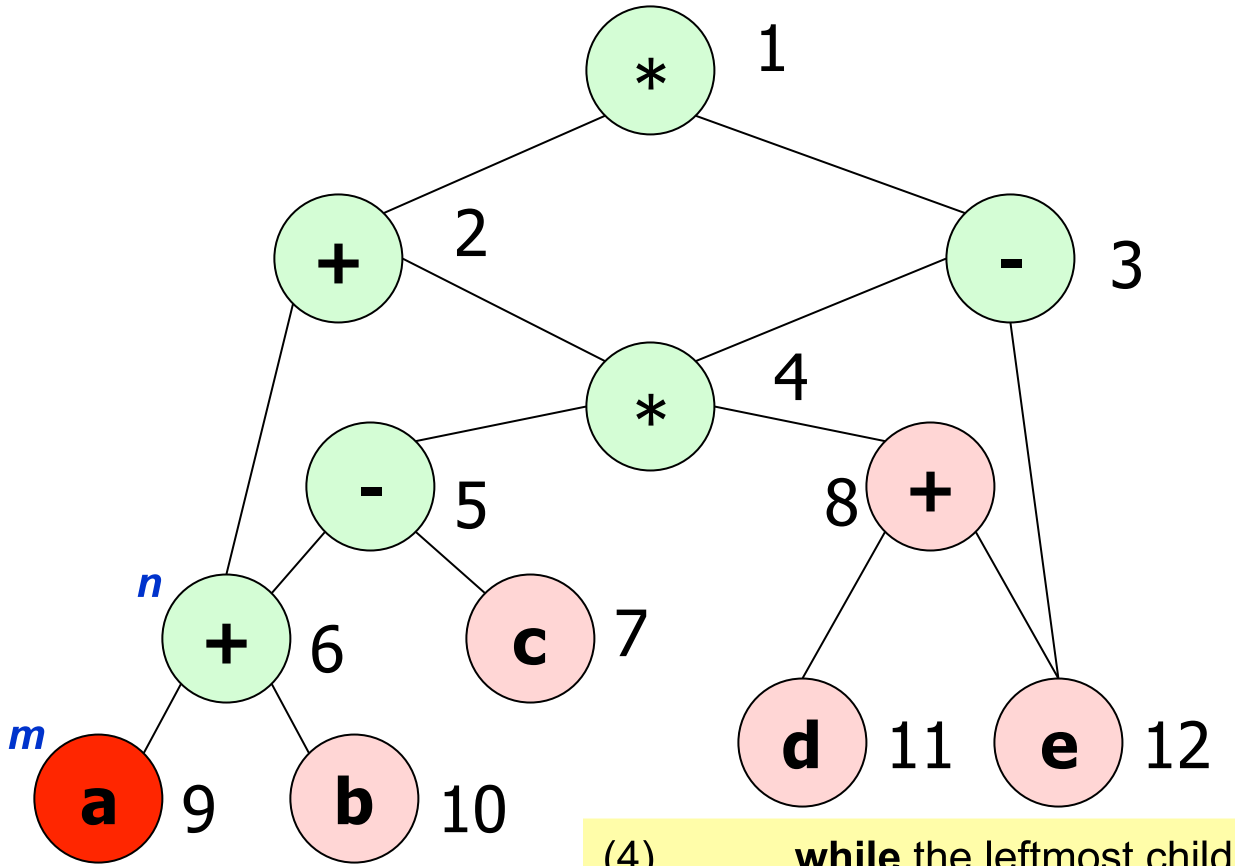
Node Listing Example



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***;

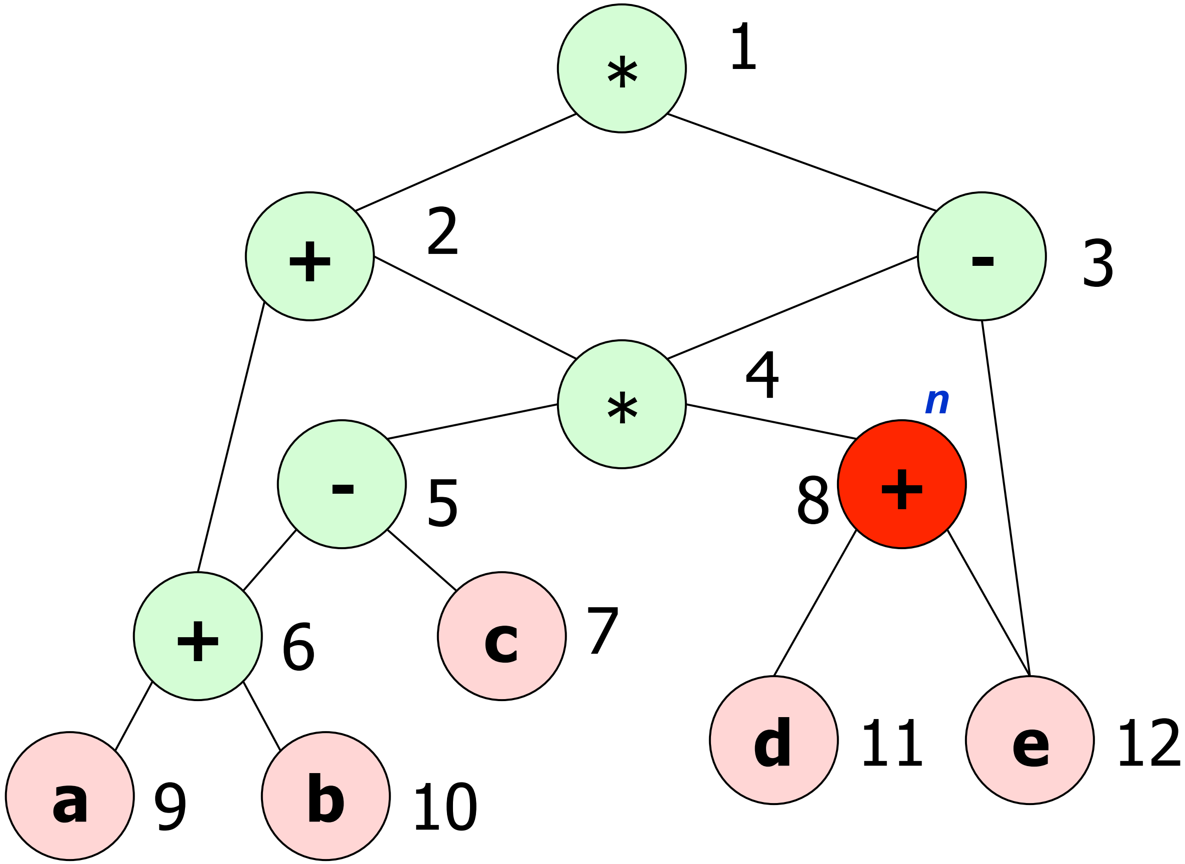
Node Listing Example



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;
```

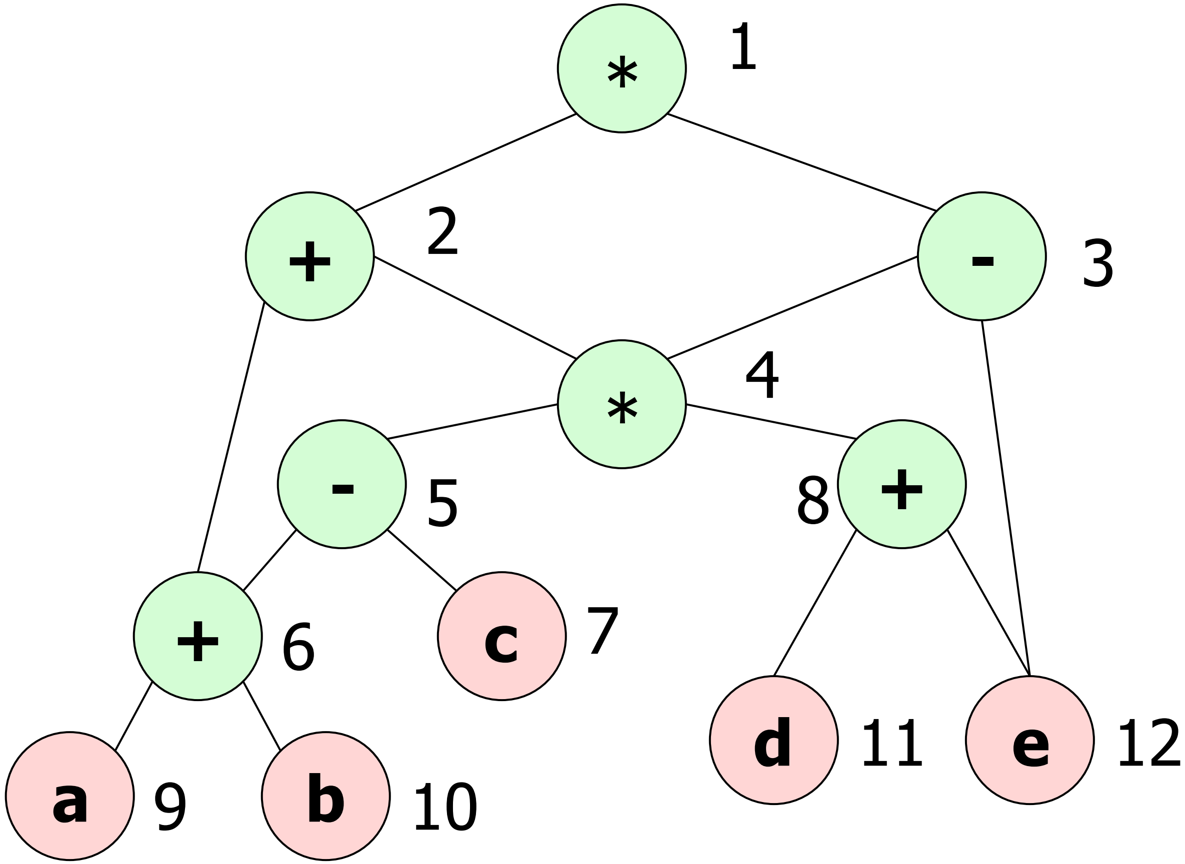
Node Listing Example



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***;

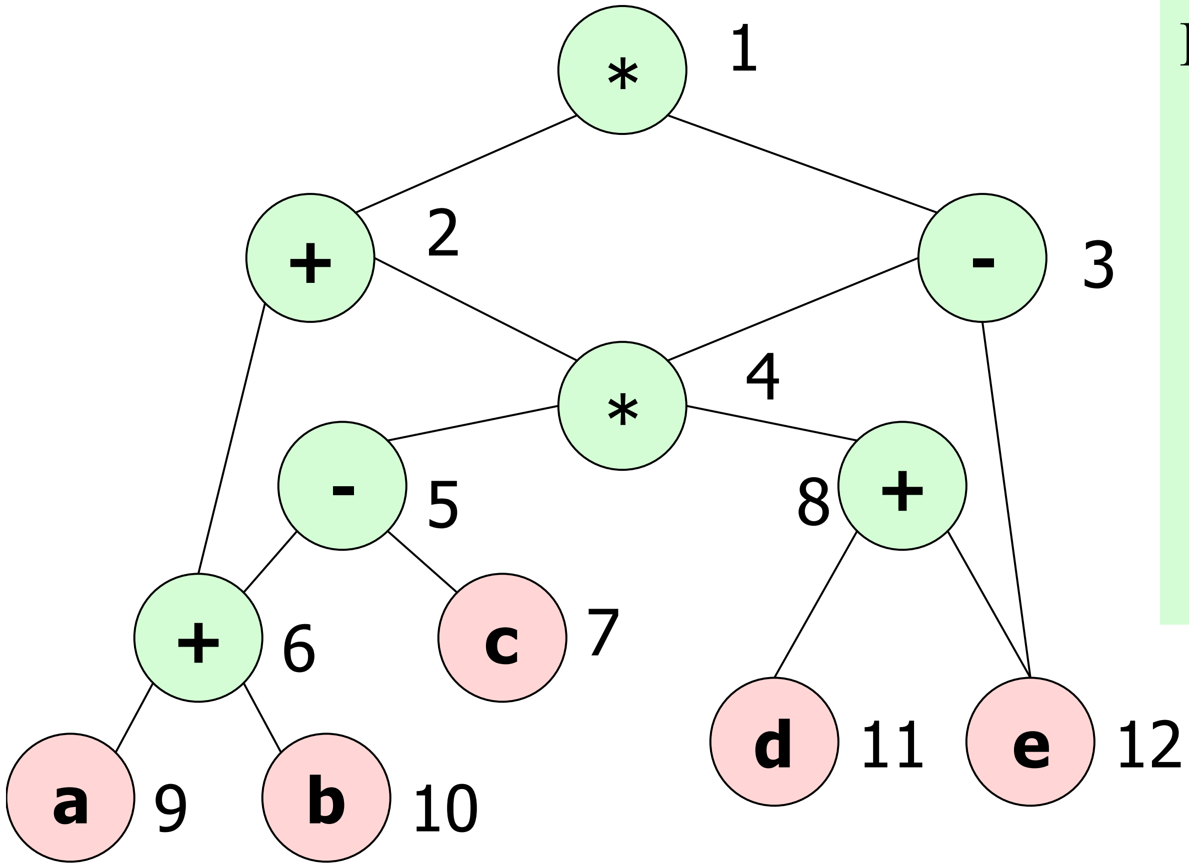
Node Listing Example



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 Example



List:	
1	
2	(a + b + ((a + b − c) ∗ (d + e)))
3	(((a + b − c) ∗ (d + e)) − e)
4	((a + b − c) ∗ (d + e))
5	a + b − c
6	a + b
8	d + e

1: (a + b + ((a + b − c) ∗ (d + e))) ∗ (((a + b − c) ∗ (d + e)) - e)

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.

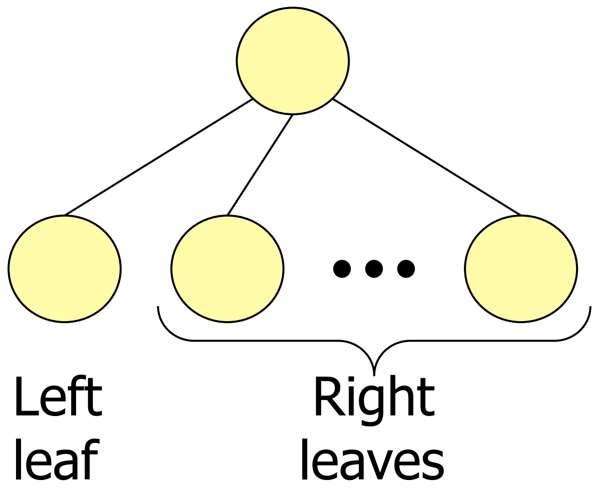
The order is optimal in the sense that it yields the shortest instruction sequence over all instruction sequences that evaluate the tree.

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) $label(n) := \max\limits_{1 \leq i \leq k} \left(label(c_i) + i - 1 \right)$

end

Labeling Algorithm

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

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

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

becomes:

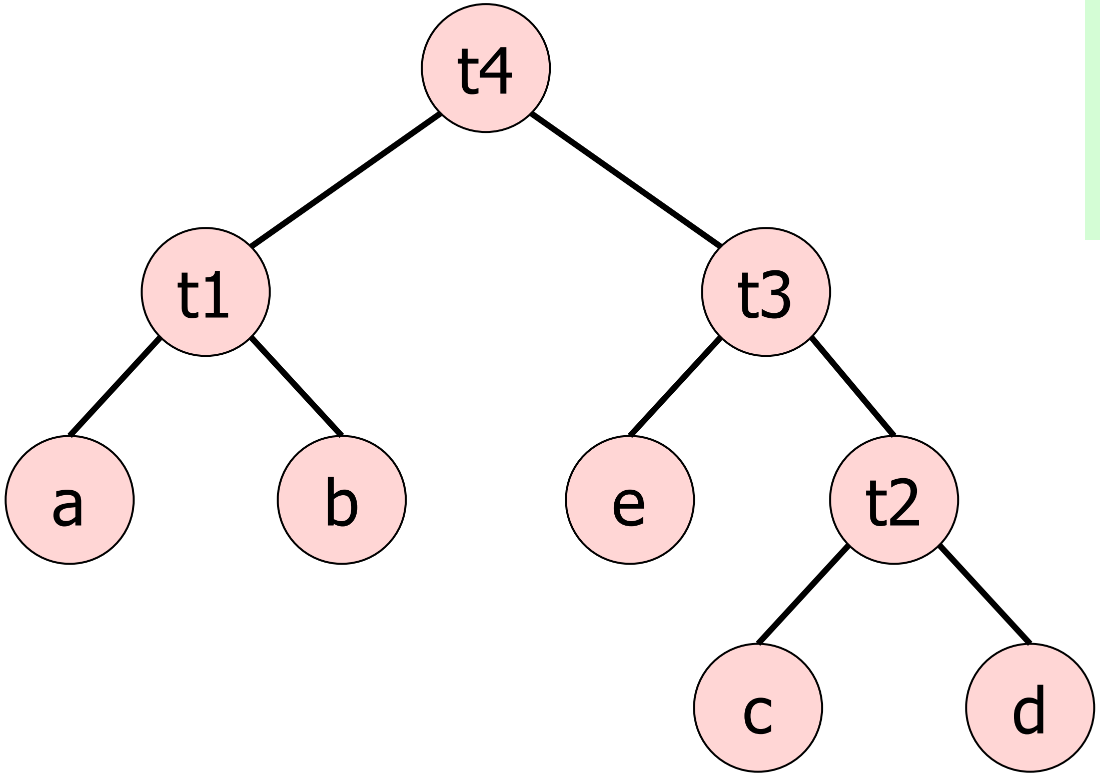
$$label(n) = \begin{cases} \max\left[label(c_1), label(c_2)\right] & \text{if } label(c_1) \neq label(c_2) \\ label(c_1) + 1 & \text{if } label(c_1) = label(c_2) \end{cases}$$

Example

Consider the expression: (a + b) – (e – (c + d))

And its corresponding intermediate code:

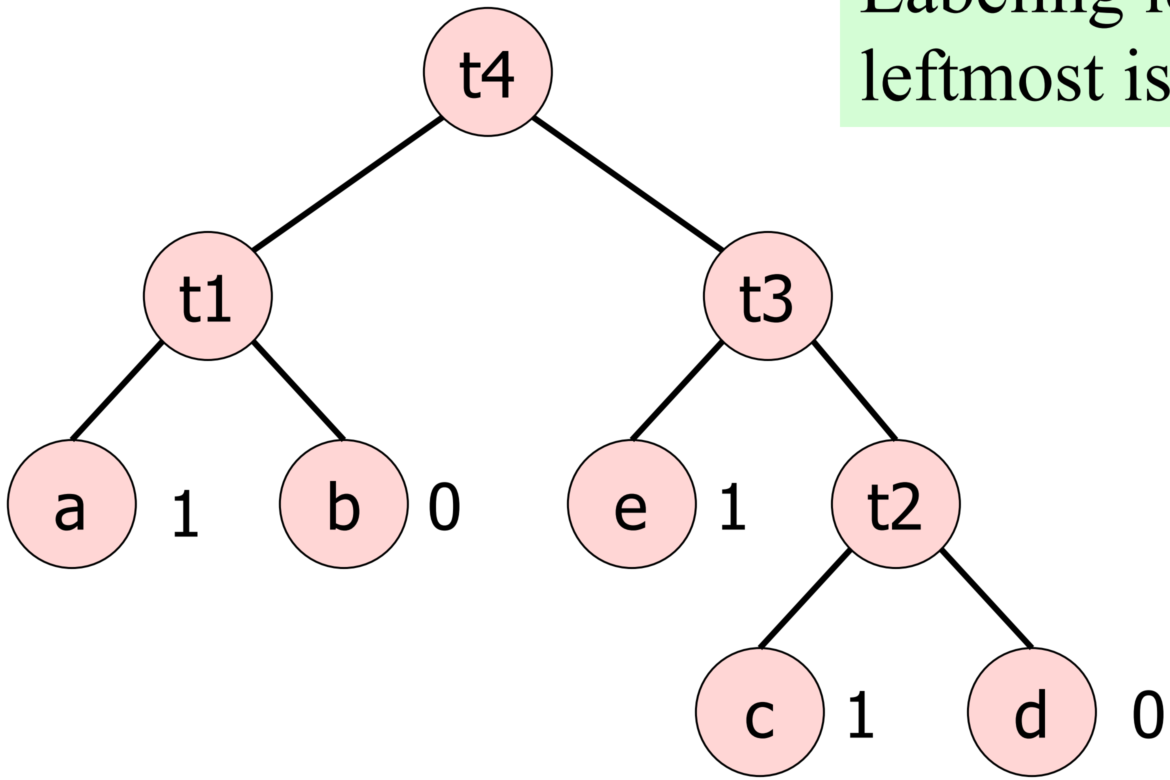
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3



Generate code for a machine with two Registers R0 and R1

Example

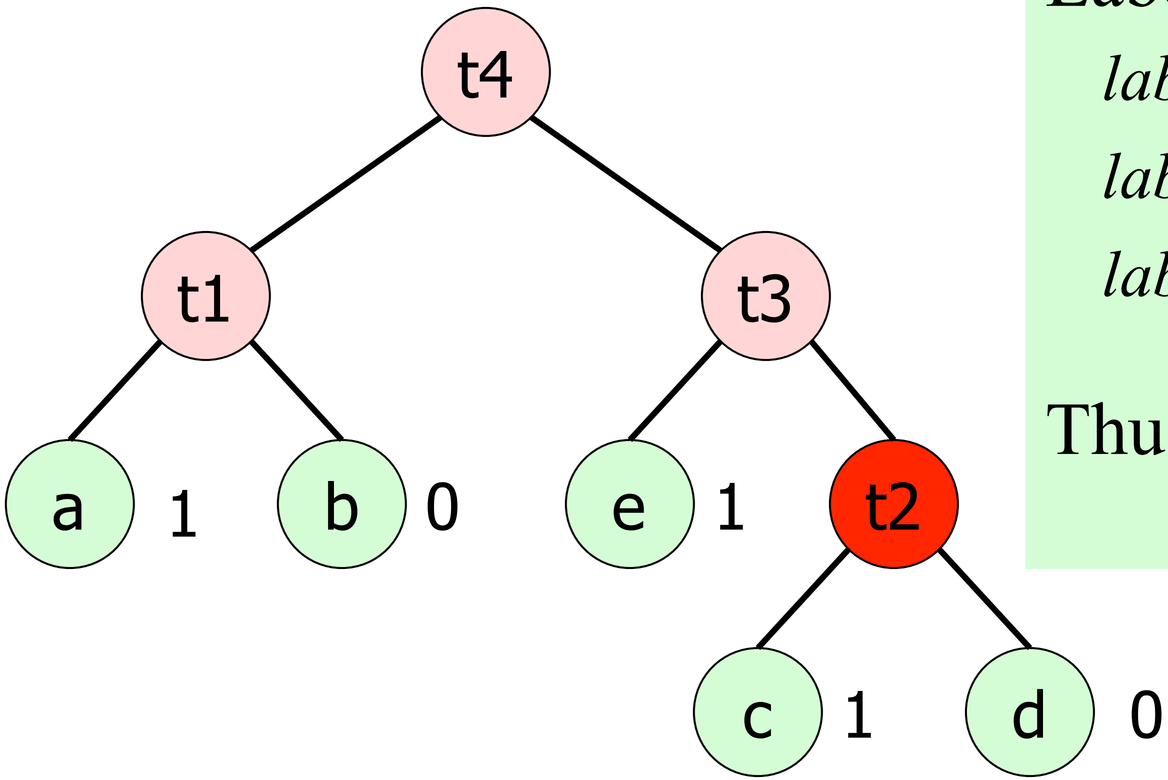
Labeling leaves:
leftmost is 1, others are 0



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



Labeling t2:

$label(c) > label(d)$

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

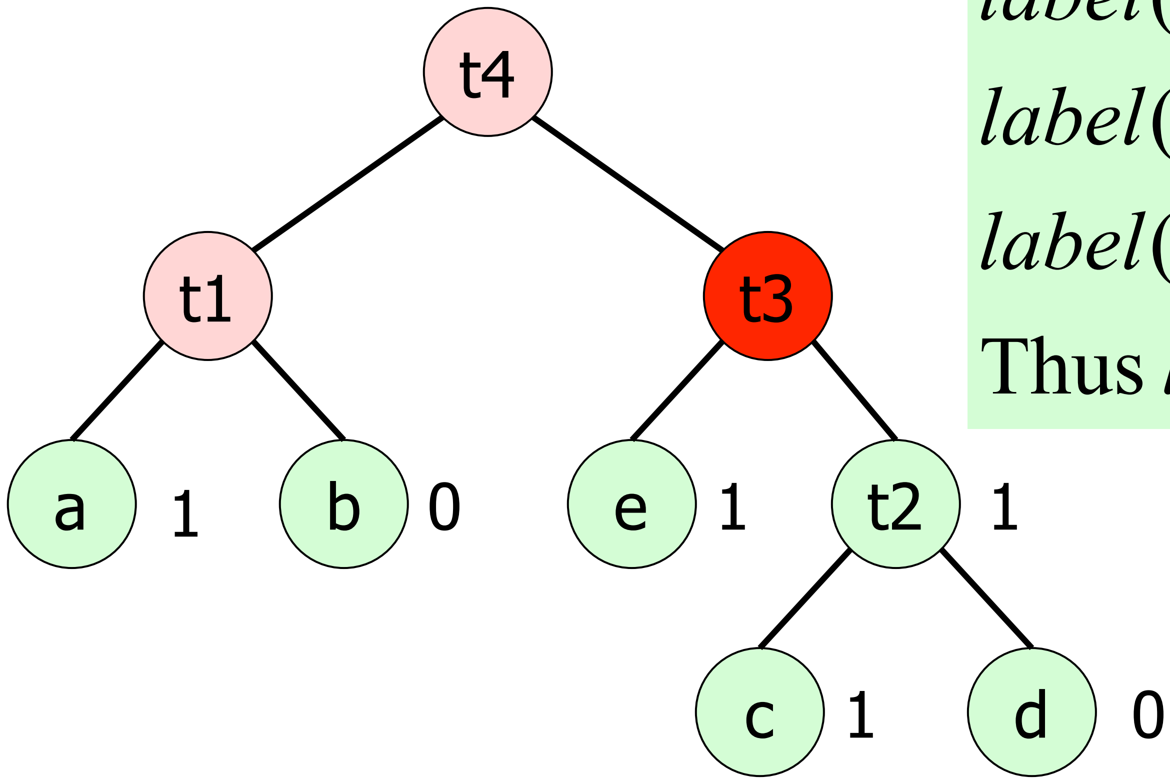
$label(d) + 2 - 1 = 1$

Thus $label(t2) = 1$

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



$label(e) = label(t2)$

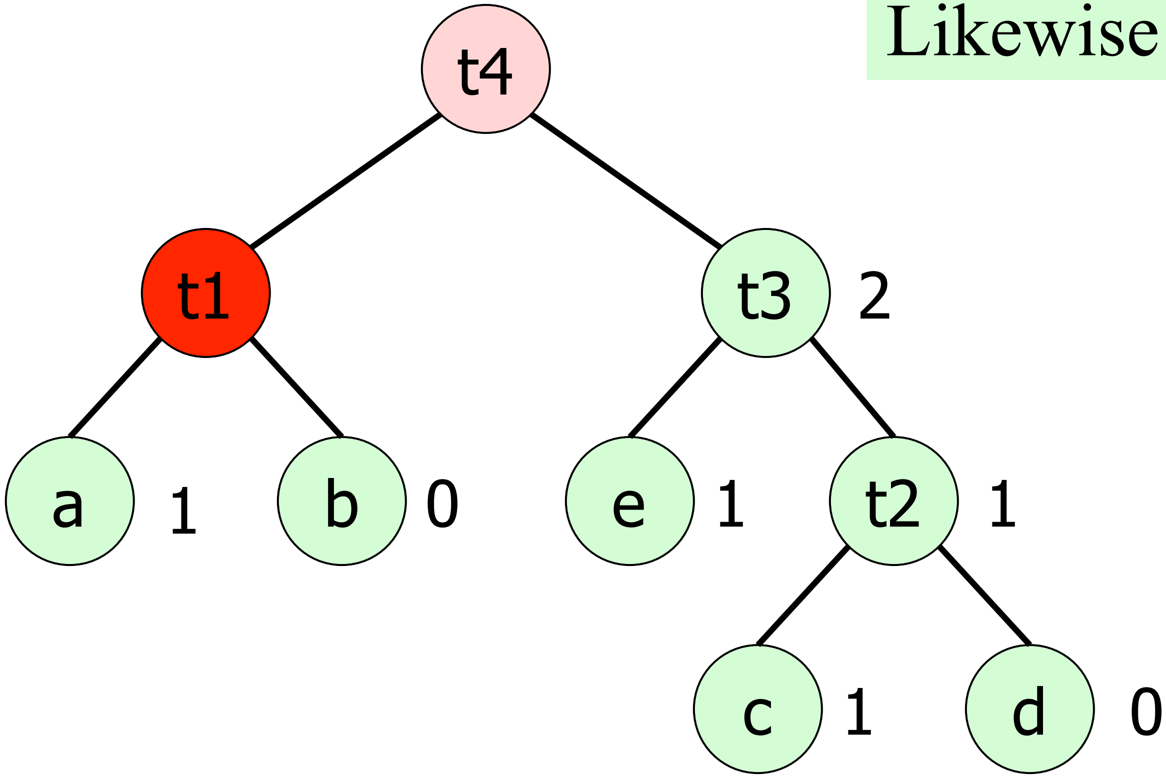
$label(e) + 1 - 1 = 1$

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

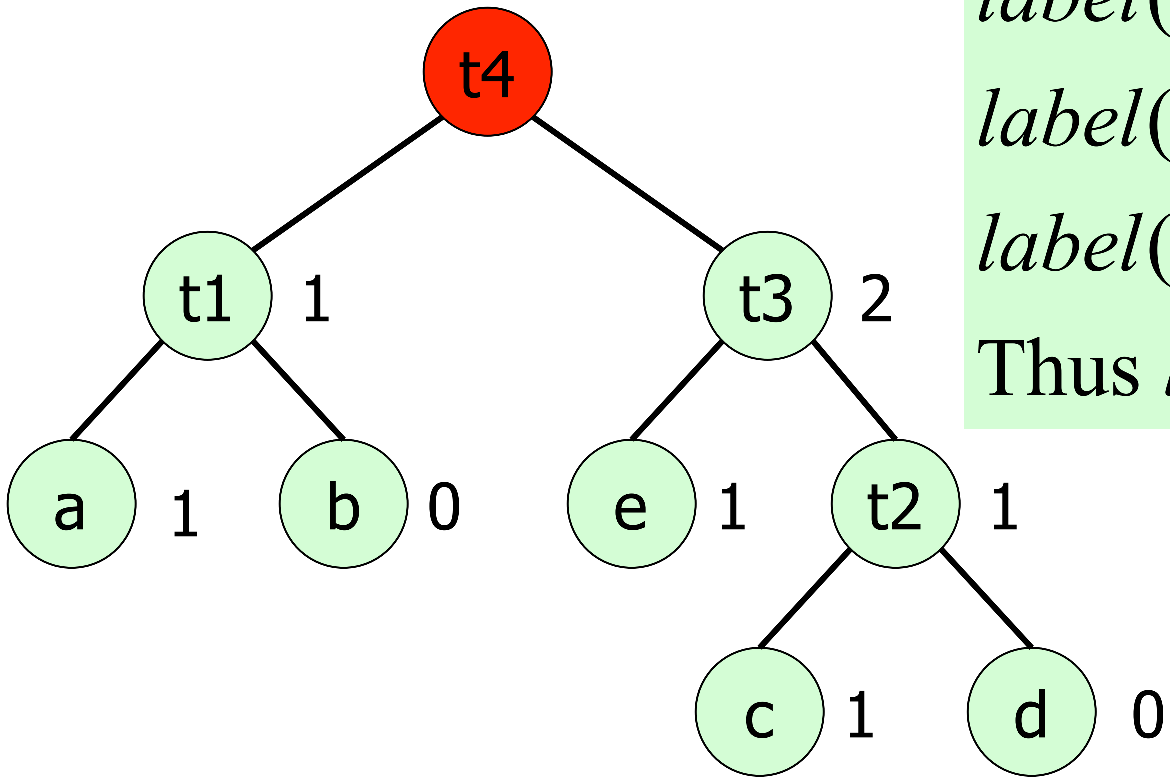
Thus $label(t3) = 2$

Example

Likewise labeling t1



Example



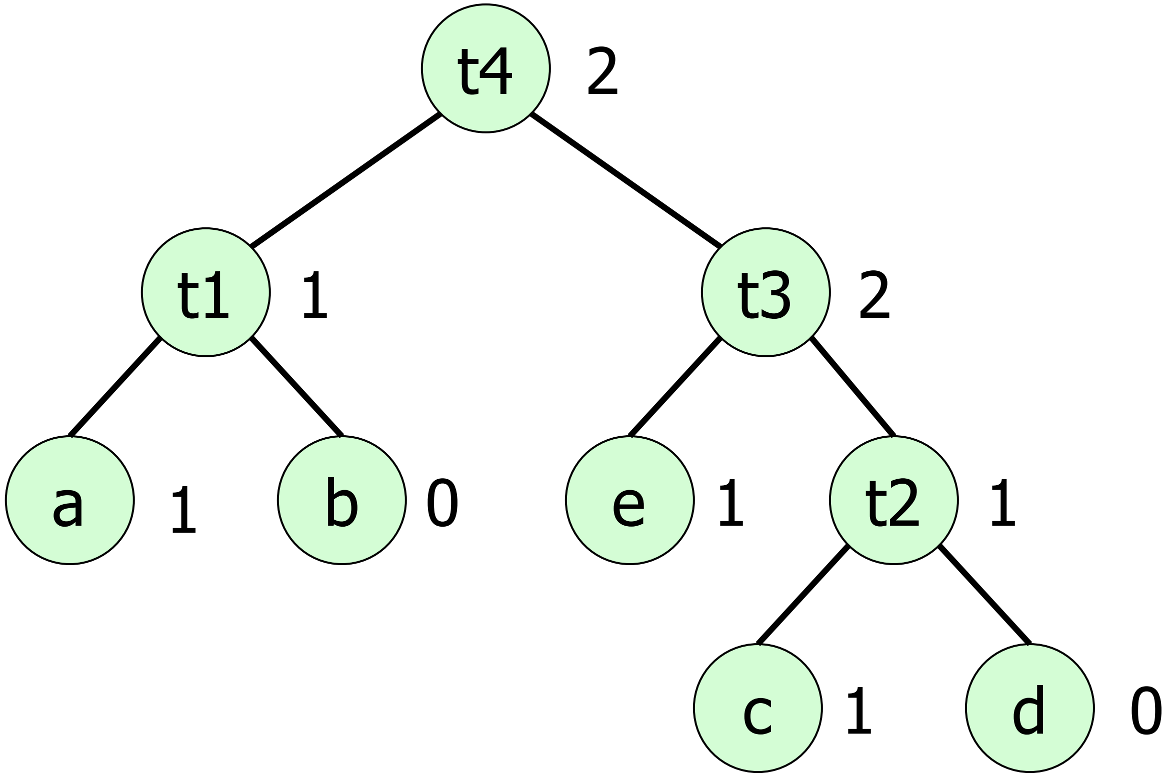
$$label(t3) > label(t1)$$

$$label(t3) + 1 - 1 = 2$$

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

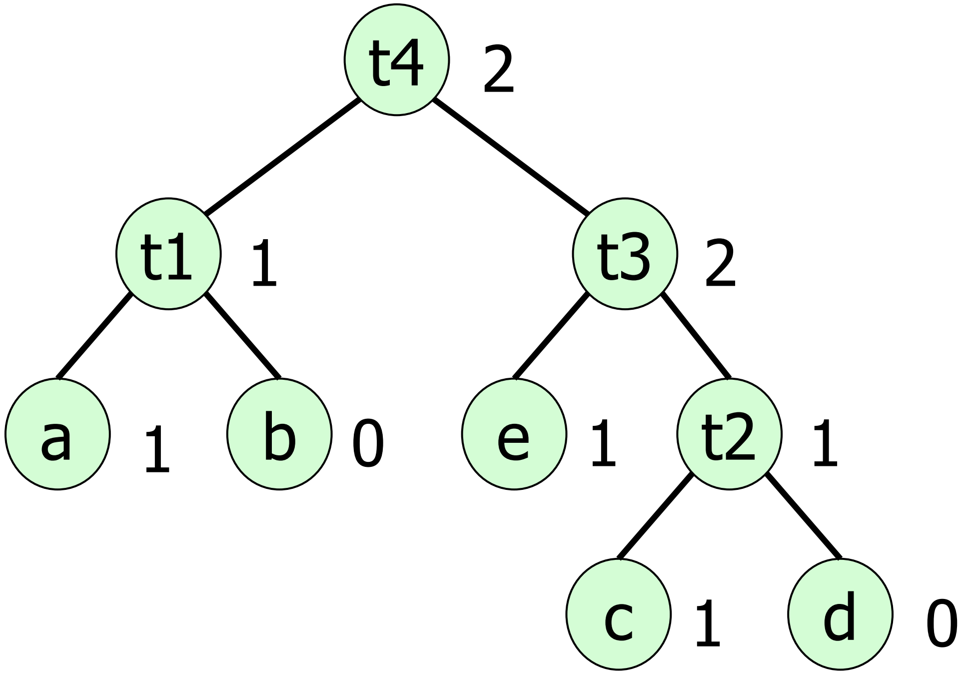
$$\text{Thus } label(t4) = 2$$

Example



Example

Now we can use the labeled tree to generate the code



MOV	e, R1
MOV	c, R0
ADD	d, R0
SUB	R0, R1
MOV	a, R0
ADD	b, R0
SUB	R1, R0