LL(*k*) and LR(*k*) grammars:

Who cares?

1. Most compiler generators (Yacc, ANTLR, etc.) use LR(*k*) or LL(*k*) grammar for syntax analysis.
2. ANTLR uses LL(*k*) grammar for lexical analysis.
3. Need to understand these grammars to construct compilers or any other parsers.

LL(k) and LR(k) grammars

Important classes of grammar. They are defined operationally:

- **LL(*k*)** grammar: one that can be used with an LL(k) parser.
- **LR(*k*)** grammar: one that can be used with an LR(k) parser.

LL(k) and LR(k) parsers will be described later, but:

- First **L** means: parser reads input from **L**eft to right.
- **L**/**R** means: parser produces a **L**eftmost/**R**ightmost derivation.
- ***k*** is the *lookahead*: parser can decide which production to use after looking at the next ***k*** terminal symbols. (If grammar is unambiguous, there is only ever one usable production.)

What is a derivation?

Grammar:

$$E \rightarrow M$$
$$E \rightarrow E + M$$
$$M \rightarrow F$$
$$M \rightarrow M * F$$
$$F \rightarrow x$$
$$F \rightarrow y$$
$$F \rightarrow (E)$$

Input:

$$(x*y)+x$$

Leftmost derivation:	Rightmost derivation:
$\underline{E}$	$\underline{E}$
$\underline{E}+M$	$E+\underline{M}$
$\underline{M}+M$	$E+\underline{F}$
$\underline{F}+M$	$E+x$
$(\underline{E})+M$	$\underline{M}+x$
$(\underline{M})+M$	$\underline{F}+x$
$(\underline{M}*F)+M$	$(\underline{E})+x$
$(\underline{F}*F)+M$	$(\underline{M})+x$
$(x*\underline{F})+M$	$(M*\underline{F})+x$
$(x*y)+\underline{M}$	$(\underline{M}*y)+x$
$(x*y)+\underline{F}$	$(\underline{F}*y)+x$
$(x*y)+x$	$(x*y)+x$

There may be only one parse tree (grammar is unambiguous) but more than one derivation!

What is lookahead?

Grammar is LL(k) if it is possible to decide production during a leftmost derivation by looking k symbols ahead.

- One symbol is enough when expanding F.
- How many symbols are needed when expanding E?

This grammar is not LL(k) for any k.

Reasons for non-LL(k):

- Left recursion (as in definitions of E and M).
- More than one production beginning with same symbol(s).

$$
\begin{aligned}
E &\rightarrow M \\
E &\rightarrow E + M \\
M &\rightarrow F \\
M &\rightarrow M * F \\
F &\rightarrow x \\
F &\rightarrow y \\
F &\rightarrow (E)
\end{aligned}
$$

Transforming a grammar to LL(k)

1. Left factoring

If productions begin with same terminal symbols:

$$A \to a\ B$$
$$A \to a\ C$$

Factor out common symbols:

$$A \to a\ A'$$
$$A' \to B$$
$$A' \to C$$

Complications:

1. If productions begin with nonterminal symbols,

   $A \to B$

   $A \to a\ C$

   $B \to a\ D$

   need to substitute, to see the terminals that each production begins with:

   $A \to a\ D$

   $A \to a\ C$

   $B \to a\ D$

Complications:

2. If production has an empty string on right:

$A \rightarrow a + A$

$A \rightarrow$

$C \rightarrow (A)$

parser can choose the empty production for A if input symbol is one that *follows* A:

Choose	if symbol is
$A \rightarrow a + A$	a
$A \rightarrow$	)

2. Removing left recursion

If production begins with same terminal symbol as on left:

$$A \rightarrow A\ a$$
$$A \rightarrow b$$

Replace by:

$$A \rightarrow b\ A'$$
$$A' \rightarrow a\ A'$$
$$A' \rightarrow$$

Complications:

1. Left recursion may be mutual:

$$A \to B\ a$$
$$A \to b$$
$$B \to A$$

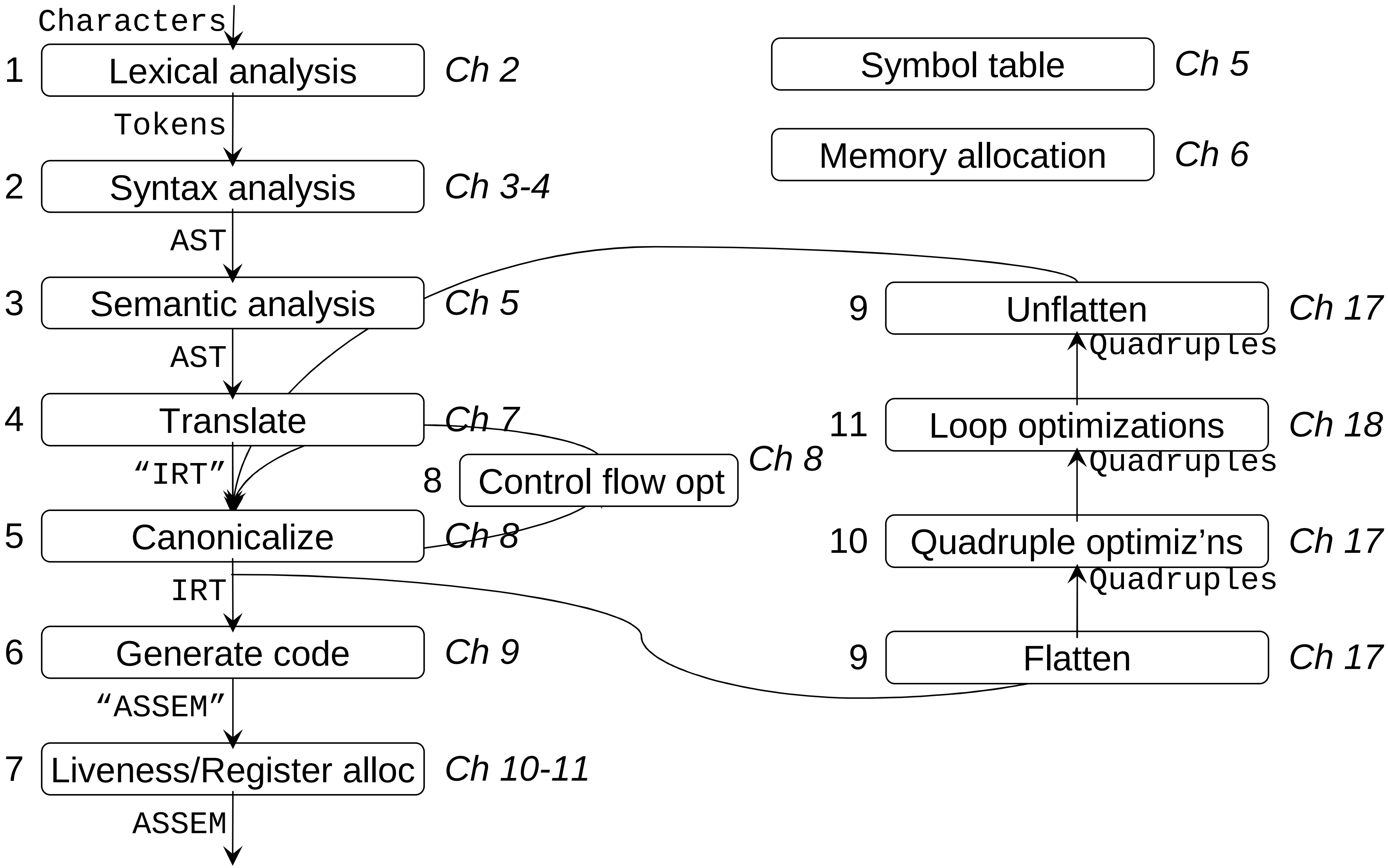
2. Left recursion may be "hidden":

$$A \to C\ A$$
$$A \to b$$
$$C \to D$$
$$C \to$$

Outline

1. Compiler phases used in this unit.

2. Chapters of Appel book.


Characters
1 Lexical analysis Ch 2
Tokens
2 Syntax analysis Ch 3-4
AST
3 Semantic analysis Ch 5
AST
4 Translate Ch 7
“IRT”
5 Canonicalize Ch 8
IRT
6 Generate code Ch 9
“ASSEM”
7 Liveness/Register alloc Ch 10-11
ASSEM
8 Control flow opt Ch 8
Symbol table Ch 5
Memory allocation Ch 6
9 Unflatten Ch 17
Quadruples
11 Loop optimizations Ch 18
Quadruples
10 Quadruple optimiz’ns Ch 17
Quadruples
9 Flatten Ch 17

Example program

Source program:

```
area = (top – bottom) * (top – bottom);
```

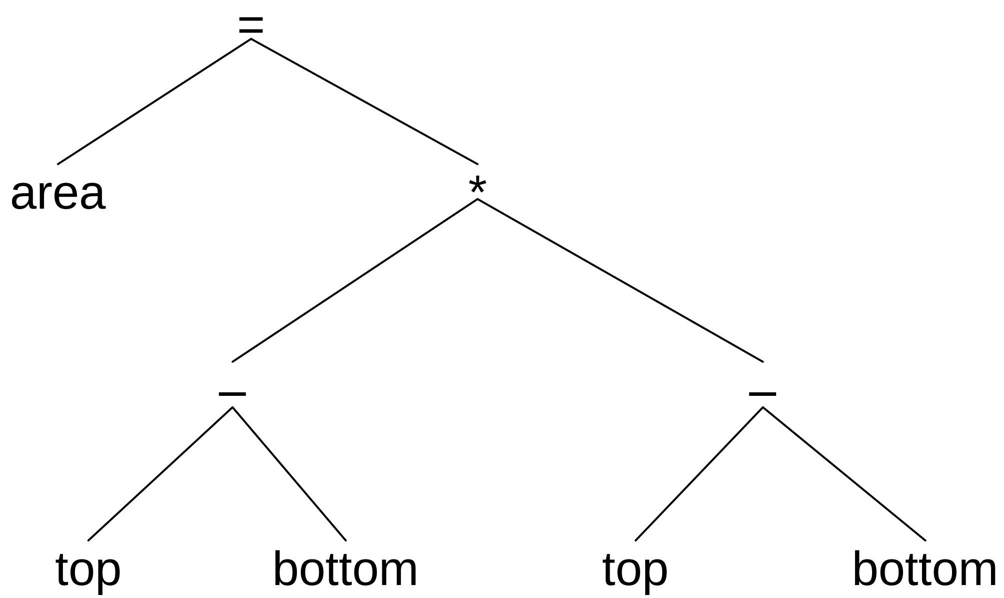
Output from stage 1 (lexical analysis)

Tokens:

```
(IDENTIFIER,"area") (EQUAL) (OPENPAREN) (IDENTIFIER,"top") (MINUS)
(IDENTIFIER,"bottom") (CLOSEPAREN) (TIMES) (OPENPAREN) (IDENTIFIER,"top")
(MINUS) (IDENTIFIER,"bottom") (CLOSEPAREN) (SEMICOLON)
```

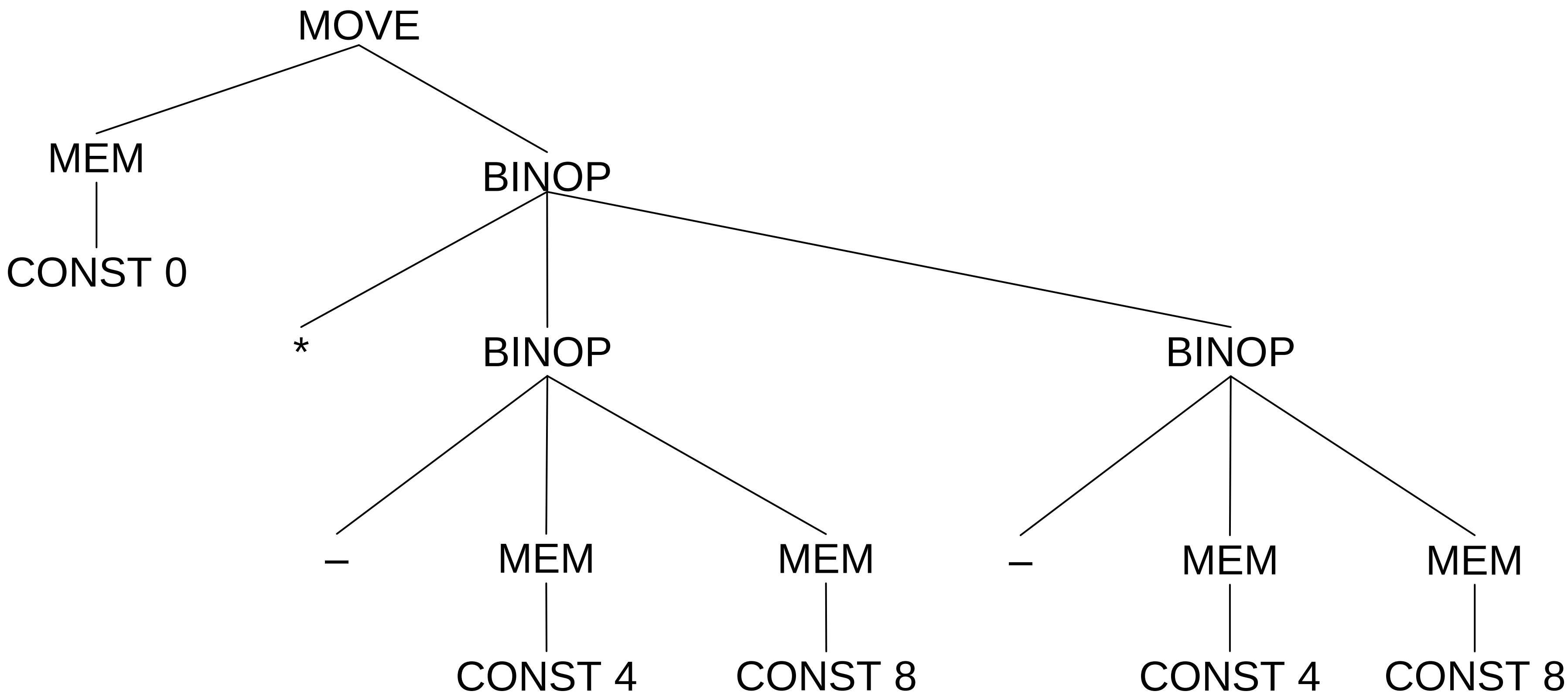
Output from stage 2 (syntax analysis)

Abstract Syntax Tree (AST):



Output from stage 4 (translation)

Intermediate Representation Tree (IRT):



Output from stage 9 (flattening)

Quadruples:

```
t1 = M[4]
t2 = M[8]
t3 = t1 – t2
t4 = M[4]
t5 = M[8]
t6 = t4 – t5
t7 = t3 * t6
M[0] = t7
```

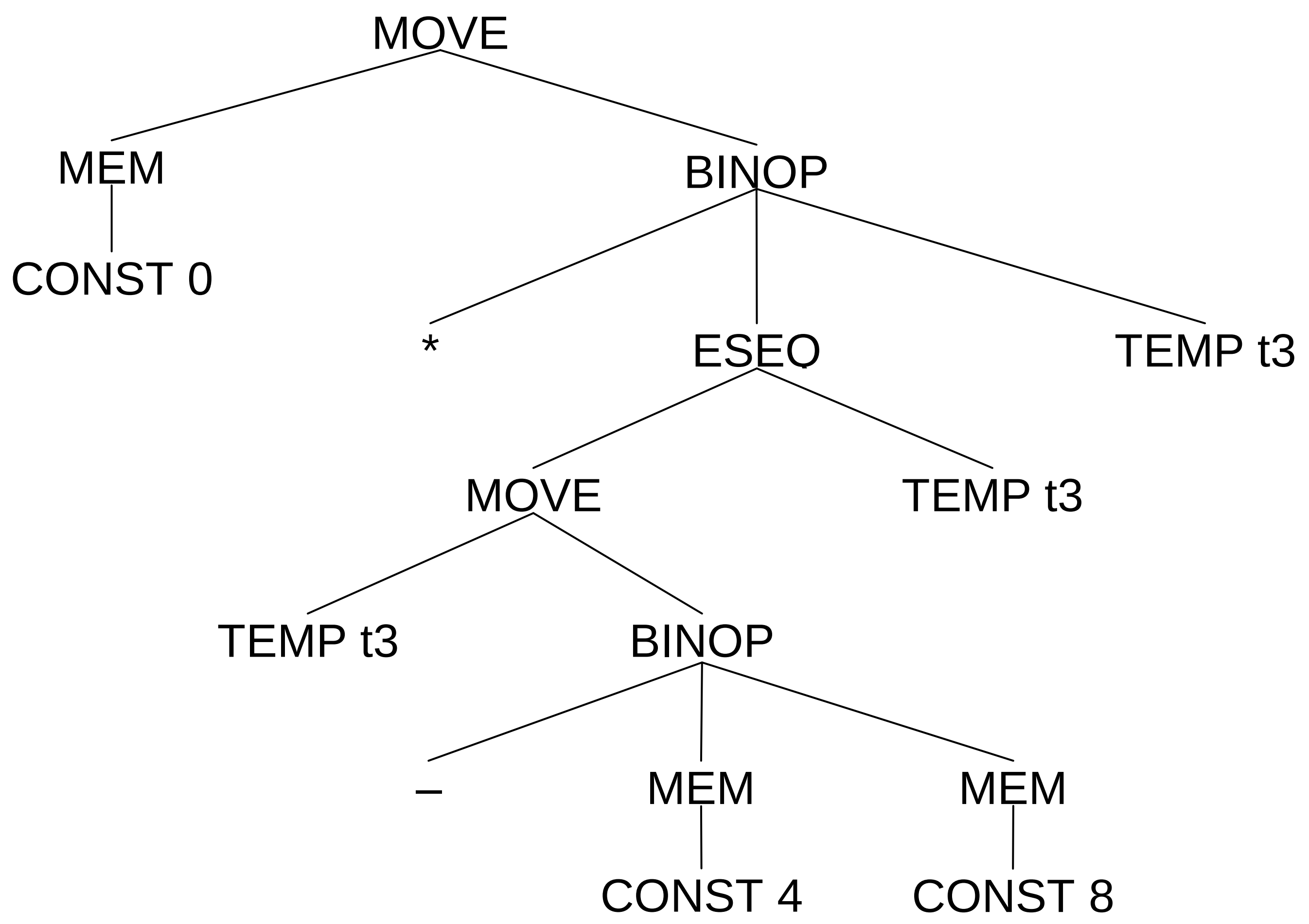
Output from stage 10 (optimization)

Quadruples:

```
t1 = M[4]
t2 = M[8]
t3 = t1 – t2
t7 = t3 * t3
M[0] = t7
```

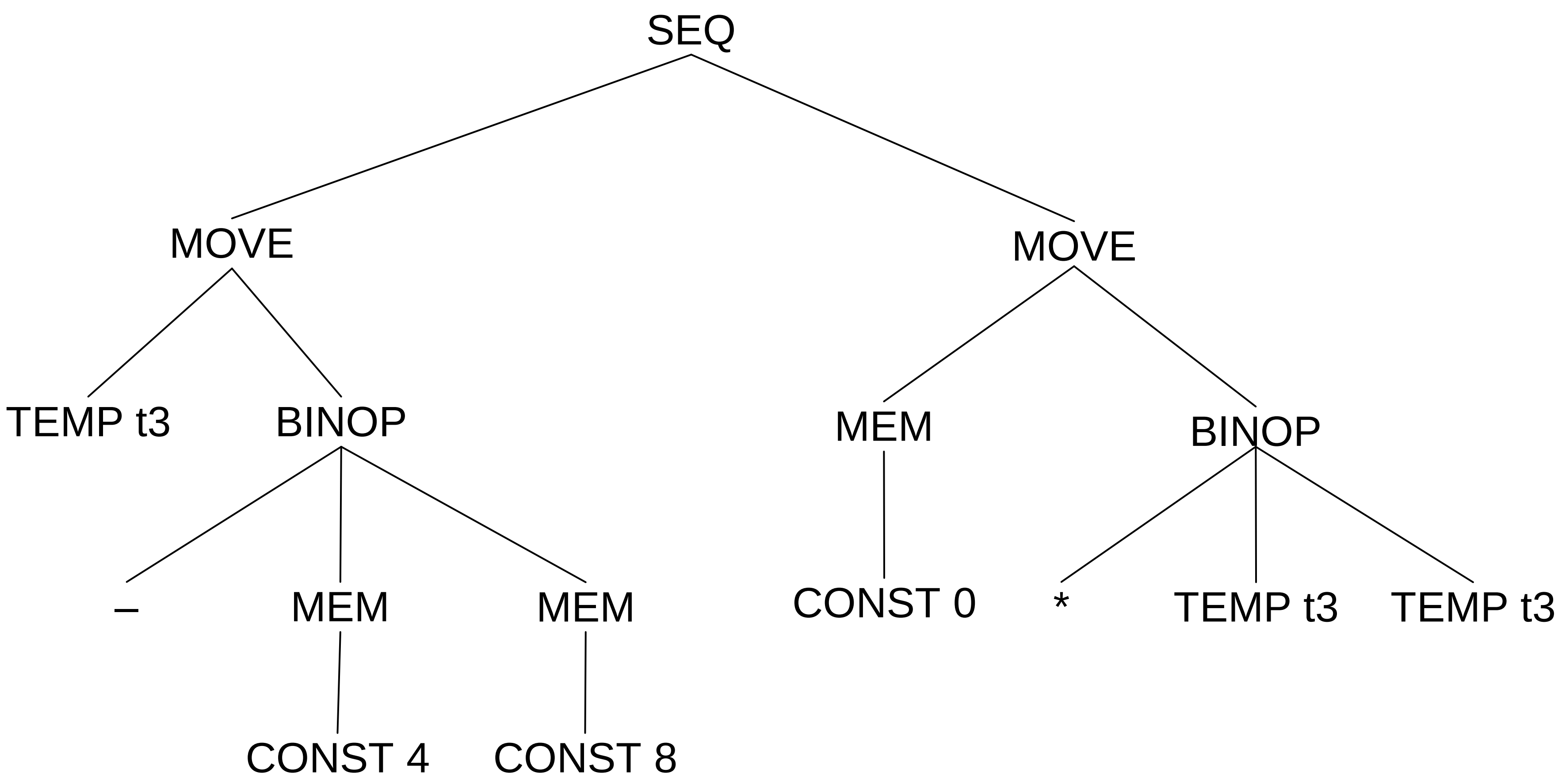
Output from stage 9 (unflattening)

Intermediate Representation Tree (IRT) (not in canonical form):



Output from stage 5 (canonicalization)

Intermediate Representation Tree (IRT):



Output from stage 6 (code generation)

Assembly code (using unlimited number of registers):

```
LOAD:  R1 ← M[4]
LOAD:  R2 ← M[8]
SUB:   R3 ← R1 - R2
ADD:   R4 ← R3 + R0
MUL:   R5 ← R4 * R4
STORE: M[0] ← R5
```

Output from stage 7 (register allocation)

Assembly code:

```
LOAD:  R1 ← M[4]
LOAD:  R2 ← M[8]
SUB:   R1 ← R1 - R2
MUL:   R1 ← R1 * R1
STORE: M[0] ← R1
```