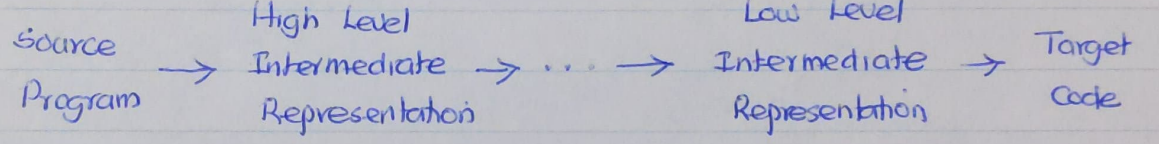
INTERMEDIATE CODE GENERATION

- In a compiler, the front-end translates a source program into an intermediate representation from which the back-end generates target code.

Intermediate Languages:

- Syntax trees, postfix notation, and three-address code are different forms of intermediate representations.
- In the process of translating a program in a given source language into code for a target machine, a compiler may construct a sequence of intermediate representations as shown below:



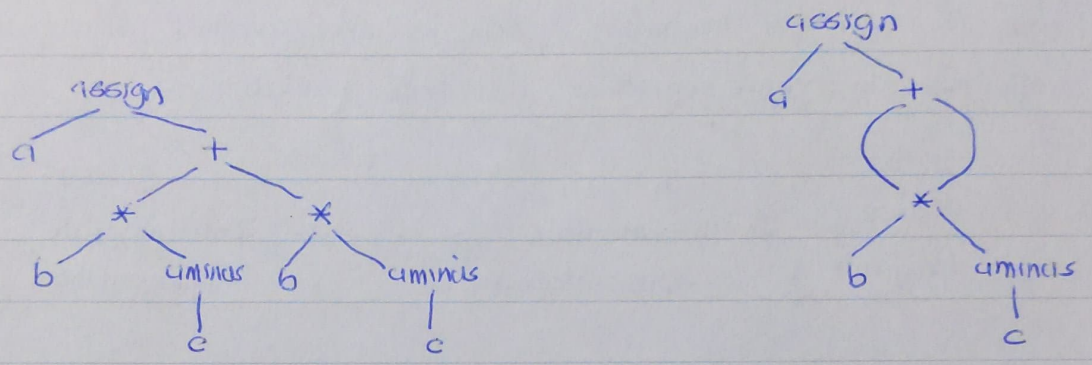
- High-level representations are close to the source language and low-level representations are close to the target machine.
  - Syntax trees are high-level; they depict the natural hierarchical structure of the source program and are well suited to tasks like static type checking.
- A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection.
- Three-address code can range from high-level to low-level, depending on the choice of operators.
  - For expressions, the difference between syntax trees and three-address code are superficial.
  - For loop constructs a syntax tree represents the components of the construct, whereas three-address code contains labels and jump statements to represent the flow of control, as in machine language.

Variants of Syntax Trees

- Nodes in a syntax tree represent constructs in the source program; the children of a node represent the meaningful components of a construct.
- A directed acyclic graph (DAG) for an expression identifies the common subexpressions of the expression.
- A syntax tree and DAG for the assignment statement

  a = b * -c + b * -c

  is given below:

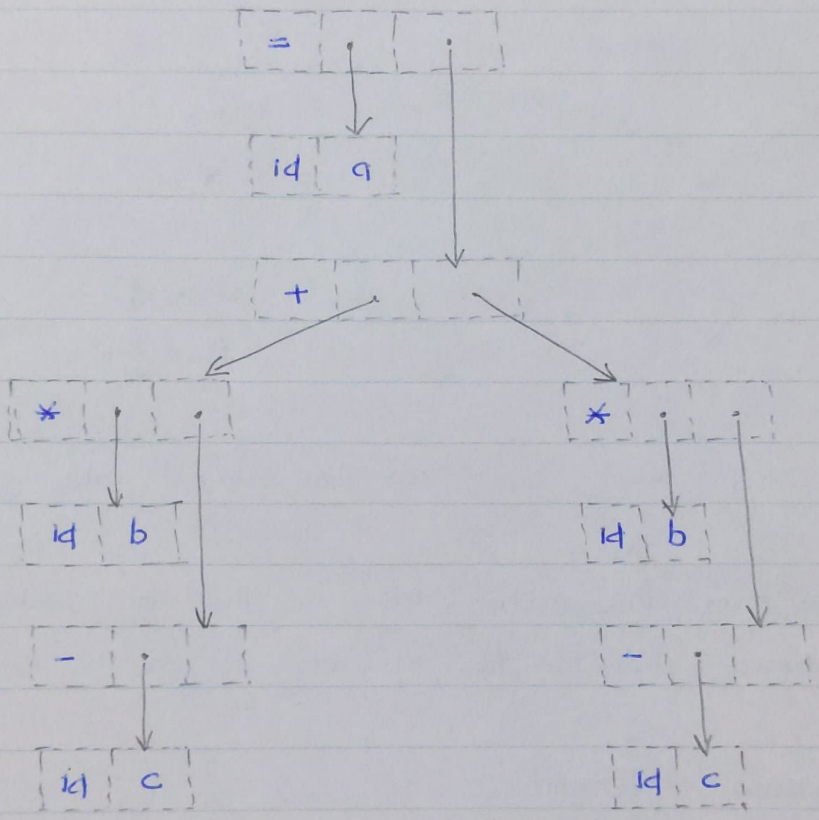


(a) Syntax tree (b) DAG

- Syntax trees for assignment statements are produced by the syntax-directed definition given below:

PRODUCTION	SEMANTIC RULES
$S \rightarrow id = E$	S.node = new Node ('=', Leaf (id, id.entry), E.node)
$E \rightarrow E_1 + T$	E.node = new Node ('+', E_1.node, T.node)
$E \rightarrow E_1 - T$	E.node = new Node ('-', E_1.node, T.node)
$E \rightarrow T$	E.node = T.node
$T \rightarrow (E)$	T.node = E.node
$T \rightarrow id$	T.node = new Leaf (id, id.entry)
$T \rightarrow num$	T.node = new Leaf (num, num.val)

- The same syntax-directed definition will produce a DAG if the function 'Node (op, left, right)' return a pointer to an existing node whenever possible, instead of constructing new nodes.
- A representation of the syntax tree for the assignment statement given above appears below:



Three Address Code

- In three-address code, there is atmost one operator on the right side of an instruction; that is no built-up arithmetic expressions are permitted.
  - A source language expression like $x + y * z$ might be translated into the sequence of three-address instructions

$$t_1 = y * z$$
$$t_2 = x + t_1$$

where, t_1 and t_2 are compiler generated temporary names.