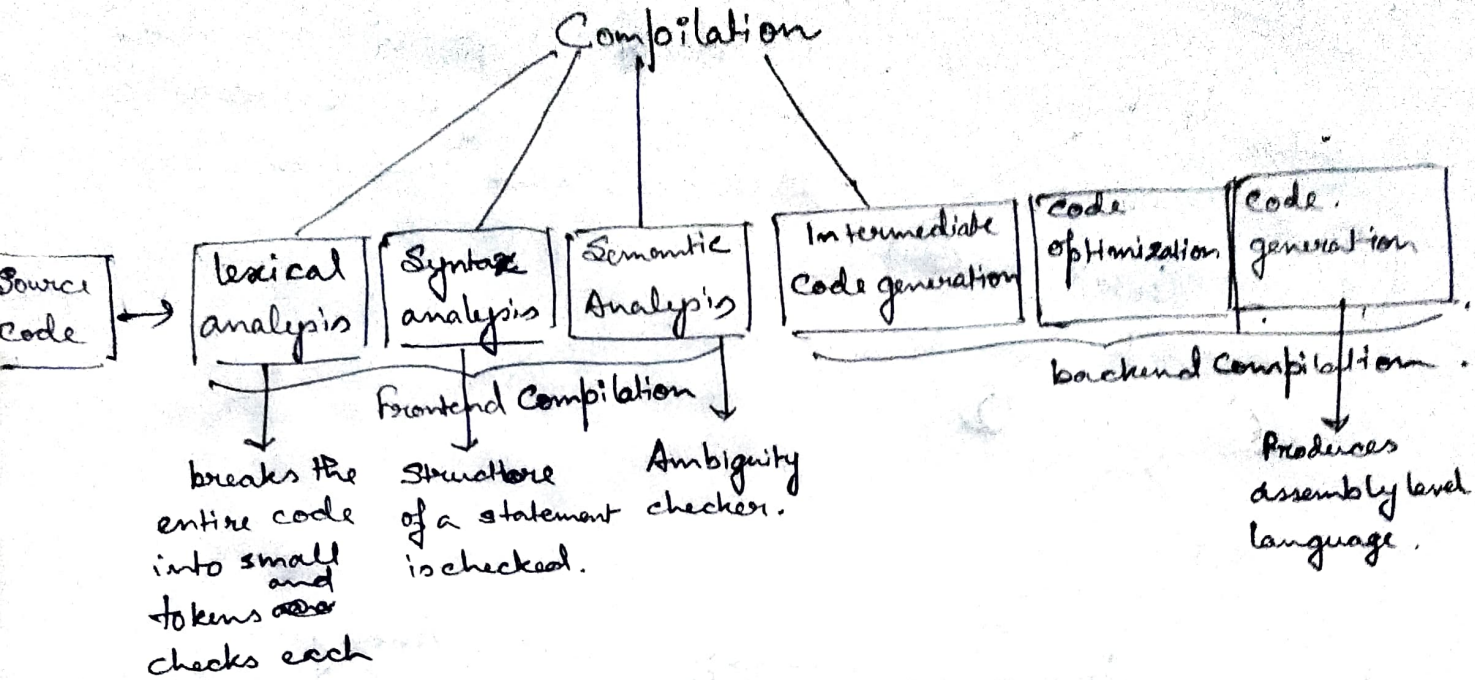
## Compiler Design :)




Ambiguity: If a local and global variable has same identifier name we give preferance to the scope.

## Intermediate Code Generation :)

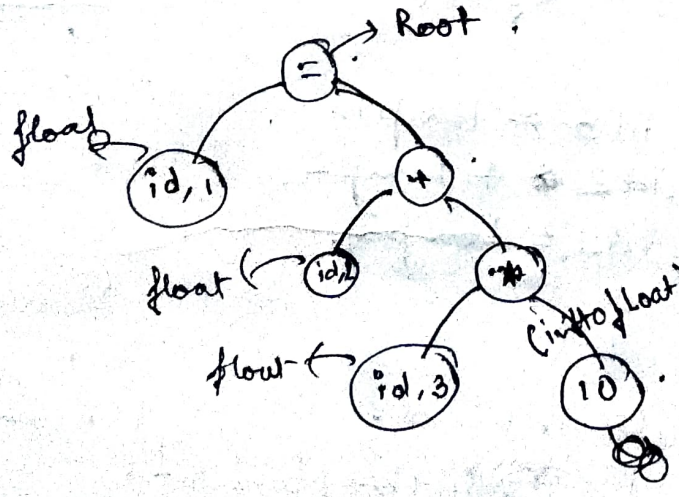
a = b + c * 10. ; → lexical Analyser.
breaks down entire stream to valid word (token) called lexims.

↓

⟨id,1⟩ ⟨=⟩ ⟨id,2⟩ ⟨+⟩ - - - ⟨10⟩ ⟨;⟩

token: ~~(id, value)~~ ⟨id, value⟩

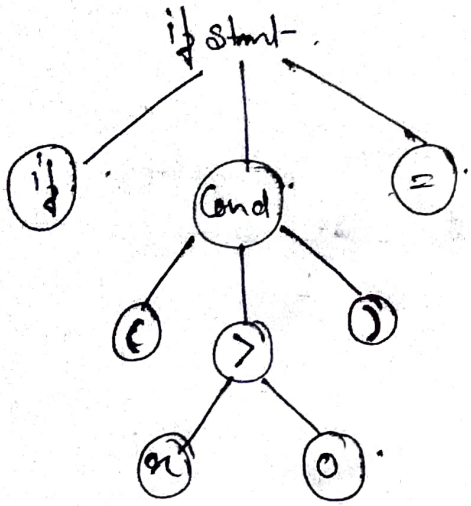
⟨id,1⟩⟨=⟩⟨id,2⟩⟨+⟩⟨id,3⟩⟨*⟩⟨10⟩ → Syntax Analyser.
Generates a parse tree.




Operator becomes root.
left-hand expression → left child.
Right hand expression → right child.
left-child → must be a variable.
right child → any expression.

Typecasting issues (implicit) is taken care at this stage only.

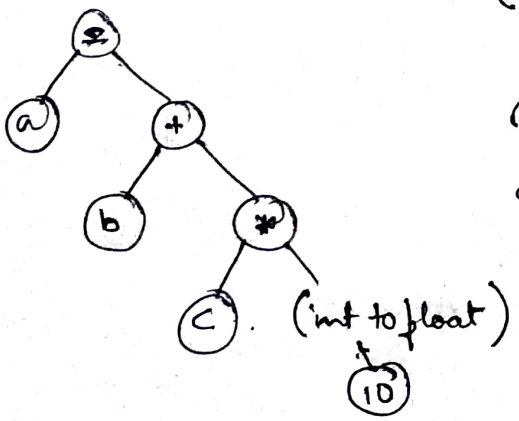
if (x > 0) y = 5;



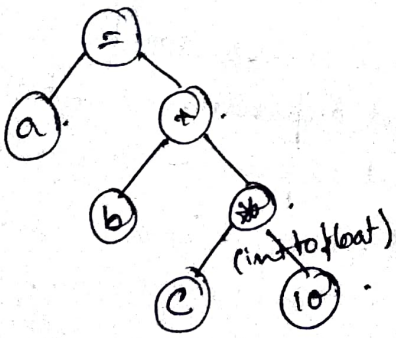

Incase, our lexical strim has braces, intermediate nodes are drawn, like "condition", "expression".

Ambiguous parse tree → Symmantic Analysis.
(~~must~~ gives unambiguous parse tree).

Typecast (implicite) is taken care at this stage.




Intermediate Code generation phase: (Three Address Code)
- Uses several intermediate representation format.




temp1 := (int-to-float) 10.
(register physically).
temp2 := id3 * temp1.
temp3 := id2 + temp2.
~~temp4~~ id1 := temp3.

In this stage resources of the computer is used.
We can use maximum of 3 ~~temporary variable~~ operands
Generation & Optimization if deligated to 2 different-
states, reduces the compilation time to great extent.

ode Optimization → Reduces the number of resources.

temp1 := id3 * (int-to-float) 10.

id1 := id2 + temp1.

Code generation : High level instructions are mapped to assembly level code (lower level code).

Everytime a variable is encountered, load operation is used to load it to the register, to be used by the processor.

LDF. (loads the floating point-number).

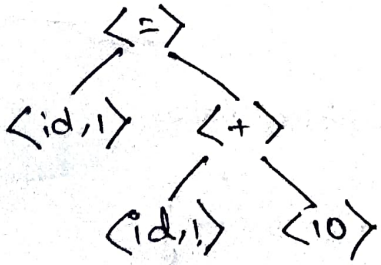
```
LDF   R1 , id3.
MULF  R1 , R1 , 10.0
LDF   R2 , id2
ADDF  R2 , R2 , R1
STF   id1 , R2
```

intel X86 ISA based code.

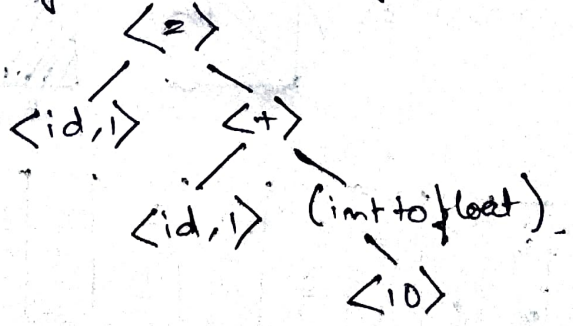
b. x := x + 10.

lexical analyser. → <id,1> <=> <id,1> <+> <10>

Syntactical analyser.




Symantical Analyser




Intermediate code.

temp1 := (int to float)(10).

temp2 := id1 + temp1.

id1 := temp2.

Optimizer:

id1 := id1 + (int-to-float)(10).

Code generation

LDF R1, id1.

ADDF R1, R1, 10.0.

STF id1, R1

## Image Processing :)

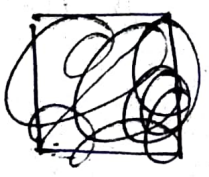
Apply Contrast-stretching techniques on 3bit-level image of size 4×4.

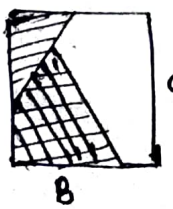
| | | | |
|---|---|---|---|
| 2 | 1 | 2 | 1 |
| 4 | 5 | 5 | 6 |
| 3 | 2 | 1 | 4 |
| 6 | 2 | 1 | 6 |

4×4

Procedure by which generate a contrast between two ~~pixel having cols~~ close value. region where the pixels are having close values.

Contrast :






| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | → pixel value |
|---|---|---|---|---|---|---|---|---|---|
| ni | 0 | 4 | 4 | 1 | 2 | 2 | 3 | 0 | → frequency. |

$$\frac{(L_{max} - L_{min})(m - m_{min})}{}$$

Formula:

$$l = \frac{(l_{max} - l_{min})(m - m_{min})}{m_{max} - m_{min}} + l_{min}.$$

m: all intensity value available in 3bit grayscale images.

$l_{min}$: lower limit (0) $l_{max}$: Upper limit (7) of gray level image

$m_{max}$: highest m (6) $m_{min}$: lowest m (0)

$$\therefore l = \frac{(7-0)(m-1)}{6-1} + 0$$

$$\boxed{l = \frac{7(m-1)}{5}}$$

* Within 3bit gray level image -ve values won't be considered hence replaced by 0.
* If value of $m > 7$ consider $m = 7$ because range is 0 to 7

For $m = 0$,

$$l = \frac{7(0-1)}{5} = -\frac{7}{5} = -1.4 \simeq 0.$$

For $m = 1$,

$$l = \frac{7(1-1)}{5} = 0$$

$m = 2$, $L = \frac{7*(2-1)}{5} = \frac{7}{5} = 1.4 \simeq 1$

$m = 3$, $L = \frac{7(3-1)}{5} = \frac{14}{5} = 2.8 \simeq 3$

$m = 4$, $L = \frac{7(4-1)}{5} = \frac{21}{5} = 4.2 \simeq 4$

$m = 5$, $L = \frac{7(5-1)}{5} = \frac{28}{5} = 5.6 \simeq 6$

$m = 6$, $L = \frac{7(6-1)}{5} = \frac{35}{5} = 7.$

$m = 7$, $L = \frac{7(7-1)}{5} = \frac{42}{5} = 8.4 \simeq 7$

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| mi | 4 | 4 | 0 | 1 | 2 | 0 | 2 | 3. |

| | | | |
|---|---|---|---|
| 1 | 0. | 1 | 0. |
| 4 | 6 | 6. | 7. |
| 3. | 1 | 0 | 4 |
| 7. | 1 | 0. | 7. |

# COMPILER DESIGN :)

Lexical Analysis : Formal languages & Automata Theory → Finite Automata (DFA, NFA)
→ Regular Languages / Expres.
→ Grammer (Regular / Context free)

Languages : Finite Set of Symbols (Alphabets).
↓
Sequence of symbols (String) → Infinite Set.
↓ subset.
Language.

Length of String = no. of position occupied.

$\Sigma = \{0, 1\}$ no. of character is 2

01101 position occupied is 5 (hence the length)

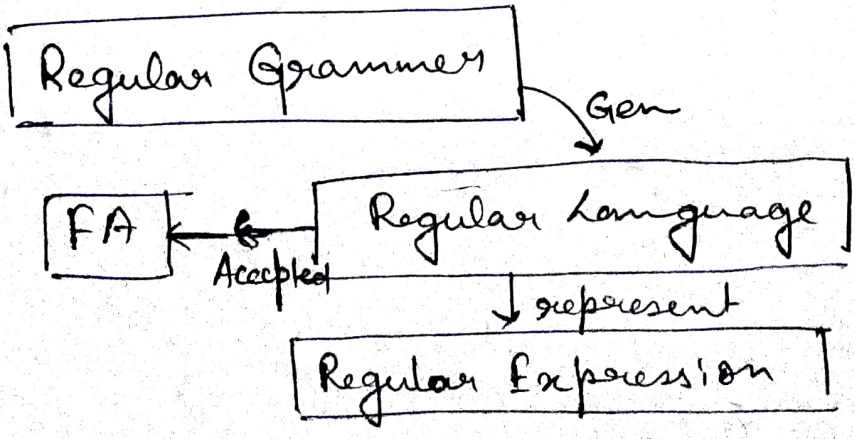
$\Sigma^K = \{$ Set of strings with string's of length K $\}$.

$\Sigma^0 = \{\epsilon\}$. $\Sigma^2 = \{00, 01, 10, 11\}$.

$\Sigma^1 = \{0, 1\}$. $\Sigma^3 = \{\ldots\}$.

$\Sigma^* = \{$ Union of all these different set $\}$ (Set of all strings)

$= \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$



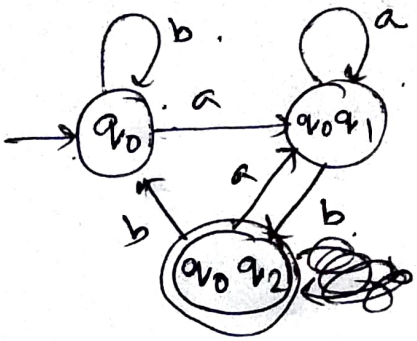
## Transition table:

NFA.

| State | a | b |
|---|---|---|
| $\rightarrow q_0$ | $q_0, q_1$ | $q_0$ |
| $q_1$ | $\phi$ | $q_2$ |
| $*q_2$ | $\phi$ | $\phi$ |

DFA.

| State | a | b |
|---|---|---|
| $\rightarrow q_0$ | $\{q_0, q_1\}$ | $q_0$ |
| $\{q_0, q_1\}$ | $\{q_0, q_1\}$ | $\{q_0, q_2\}$ |
| $*\{q_0, q_2\}$ | $\{q_0, q_1\}$ | $q_0$ |




## Regular Expression:

Union → + (a+b)

Concatination → . (a.b)

Kleen Star → * (Closure)

$\Sigma = \{a, b\}$.

| language | Reg Ex: |
|---|---|
| $\{a\}$ | a |
| $\{a, b\}$ | (a+b) |
| $\{ab\}$ | (a.b) |
| $\{\epsilon, a, aa \ldots\}$ | a* |

Q. Find the regular expression of a language over alphabet $\Sigma = \{a, b\}$ where strings are length 2.

$L = \{aa, ab, ba, bb\}$.

Reg Ex = aa + ab + ba + bb

= a(a+b) + b(a+b)

= (a+b).(a+b).

Q. Set of all even length strings?

$L = \{\epsilon, aa, ab, ba, bb, aaaa, abba\}$.

→ $((a+b).(a+b))^*$

Q. Set of all strings with exactly 2 a's

→ $b^*ab^*ab^*$

Q. Set of all even a's.

→ $(b^*ab^*ab^*)^* + b^*$

Q. Set of all strings that begins and ends with two diff$^n$ symbols.

→ $a(a^*b^*)b + b(a^*b^*)a$ $\quad a(a+b)^*b + b(a+b)^*a$.

$(a^*b^*)(ab+ba)$

Grammer:

$\vdash \langle V, T, P, S \rangle$.

V → Set of all non-terminals $\{A, B\}$ → Variables.

T → Set of terminal $\{a, b\}$ → $\Sigma$ alphabet

P → Production rule [Head → Body] Combination of Symbols

S → Starting Symbols.

eg: $V = \{S\}$ $\quad P = \{S \to SS+, S \to SS-, S \to a\}$.

$T = \{+, -, a\}$ $\quad S = S$.

Q. i) $(a+b)^*$ : $S \to aS \mid bS \mid \epsilon$.

ii) $a(a+b)^*b$: $S \to aAb$

$A \to aA \mid bA \mid \epsilon$

iii) $a^n b^n \mid n \geq 1$:

$S \to aSb \mid ab$

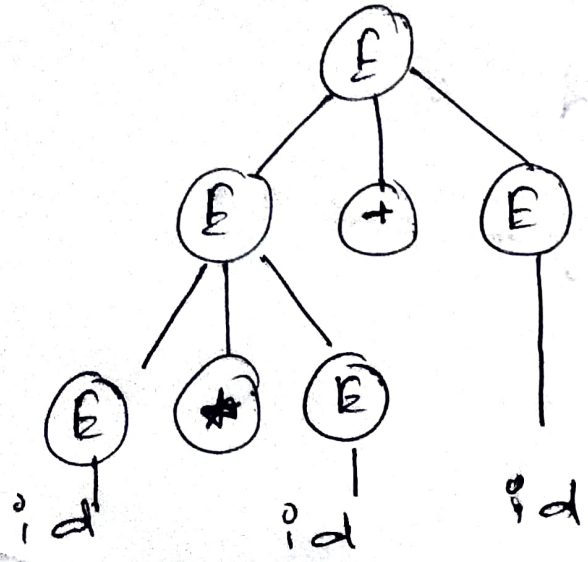
iv) Set of all pallindromes over $\Sigma = (a, b)$

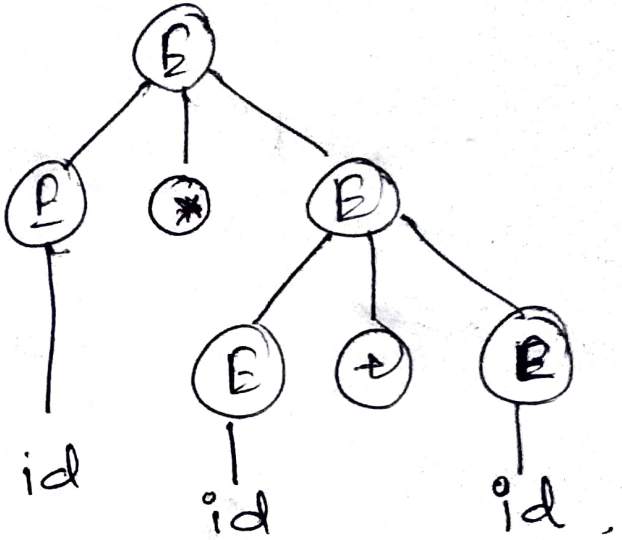
~~$S \to AB \mid \in$~~

~~$A$~~

$S \to aSa \mid bSb \mid a \mid b \mid \in$

## Ambiguous Grammar:




## Syntax Analysis:

↳ Parsing: i. Top down - LL(1)

ii. Bottom up - LR(0) : SLR

## Intermediate Code Generation:

Input: Unambiguous Parse Tree (from Semantic Analysis)

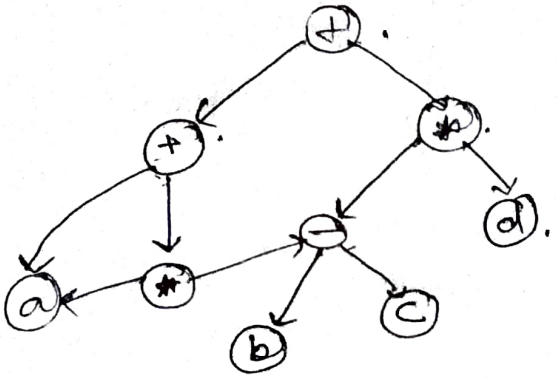
Output: i) Similar to Assembly level.
ii) Three Address Code / D.A.G
↓
Directed Acyclic Graph.

## Directed Acyclic Graph:

→ Algebraic Structure, collection of edges and vertices

~~Directed~~

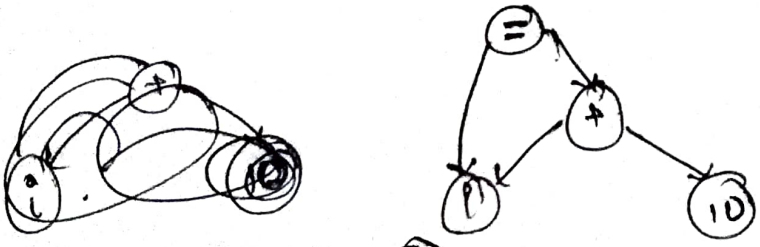
a + a * (b - c) + (b - c) * d.



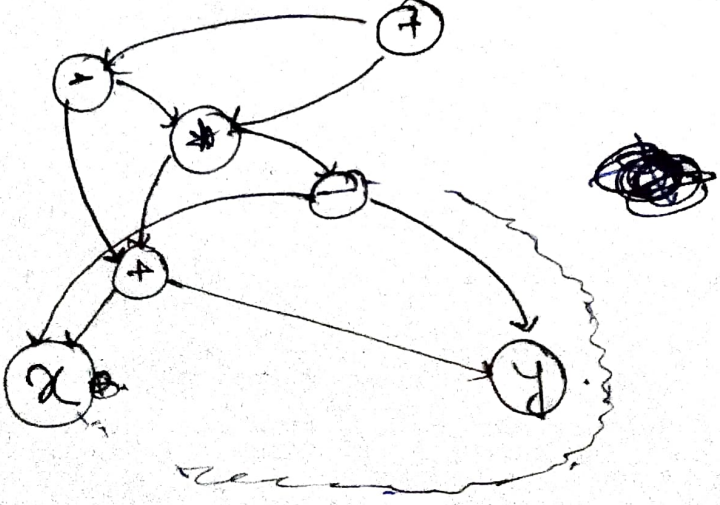

Q. i) i = i + 10

ii) ((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))

i)

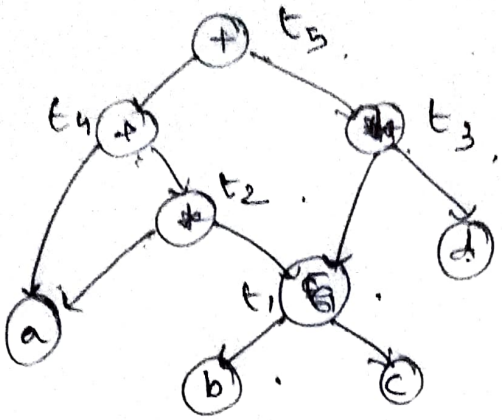



ii)



Three Address Code Representation.

$a + a * (b - c) + (b - c) * d$




$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = t_1 * d$

$t_4 = a + t_2$

$t_5 = t_4 + t_3$

Types of instructions :>

- Binary : $x = y + z$.
- Unary : $x = -y$
- Assignment : $x = y$
- Conditional.

Data structures used to store instruction :-

i) Quadripples.

ii) Triples.

iii) Indirect - Tripples.

Quadripple DS.

| Operator | arg1 | arg2 | result |
|---|---|---|---|
| minus | c | - | $t_1$ |
| * | b | $t_1$ | $t_2$ |
| + | $t_2$ | $t_2$ | a |

$a = b * -c + b * -c$

↑ unary minus.

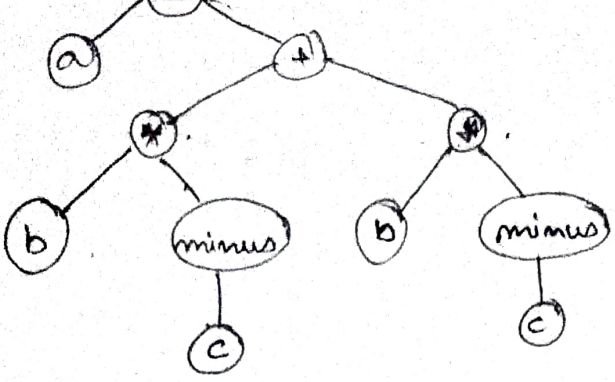
$t_1 = \text{minus } c$

$t_2 = b * t_1$

$a = t_2 + t_2$

Optimized.

Optimized Output.




Intermediate code is the simple conversion of the syntax tree.

P.S: It is not optimized.

Three address Code:

$t_1 = minus\ c$

$t_2 = b * t_1$

$t_3 = minus\ c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

| op | arg1 | arg2 | result |
|---|---|---|---|
| minus | c | | $t_1$ |
| * | b | $t_1$ | $t_2$ |
| minus | c | | $t_3$ |
| * | b | $t_3$ | $t_4$ |
| + | $t_2$ | $t_4$ | $t_5$ |
| = | $t_5$ | | a |

Intermediate Code generation Output.

Triples :)

| | op | arg1 | arg2 |
|---|---|---|---|
| (0) | minus | c | |
| (1) | * | b | (0) |
| (2) | minus | c | |
| (3) | * | b | (2) |
| (4) | + | (1) | (3) |
| (5) | = | a | (4) |

Here we have index value of each operation.

We donot have result column.

In triples we try to reduce the ~~no of operation~~ size of the code.

## Indirect Triples:

↳ Array is created, that holds the address of the instruction. or elements are referred to actual instructi

| (0) | (1) | (2) | (3) | (4) | (5) |
|---|---|---|---|---|---|

Makes the orginal triple relocatable:

We only upload the Indirect triple, so that it reduces the memory space to be used in a processor.

Shortnote: LEX, YACC, Activation Record, Symbol Table