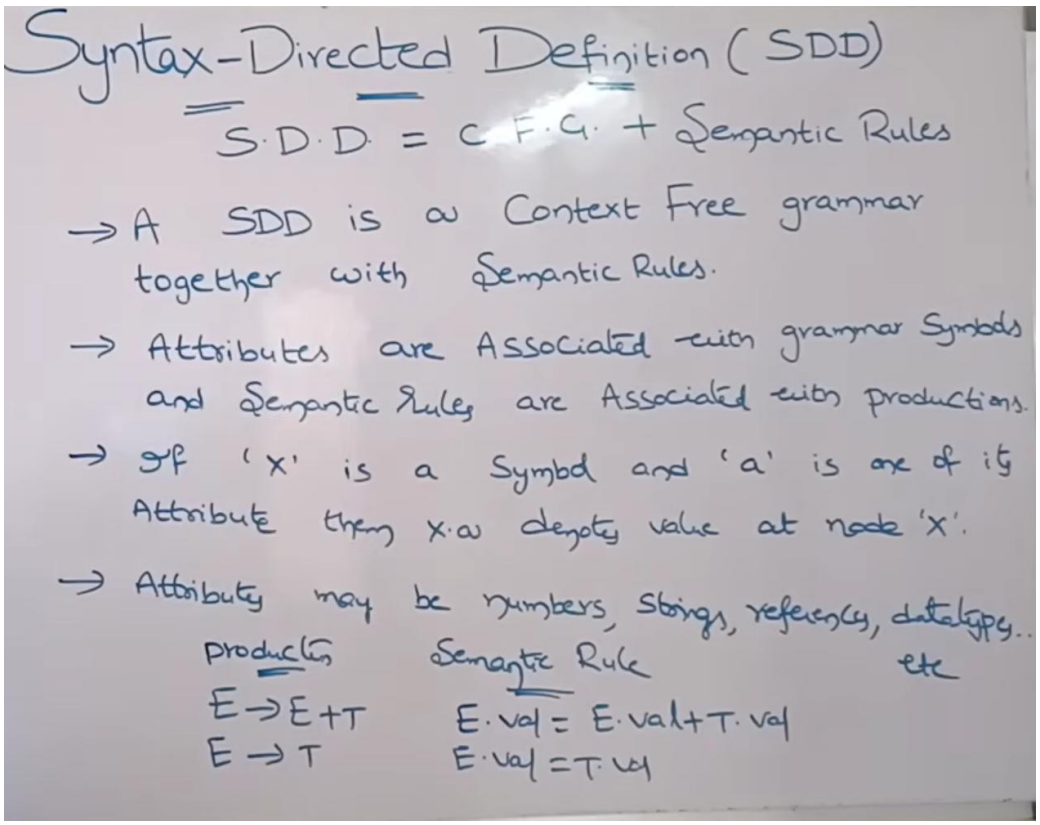
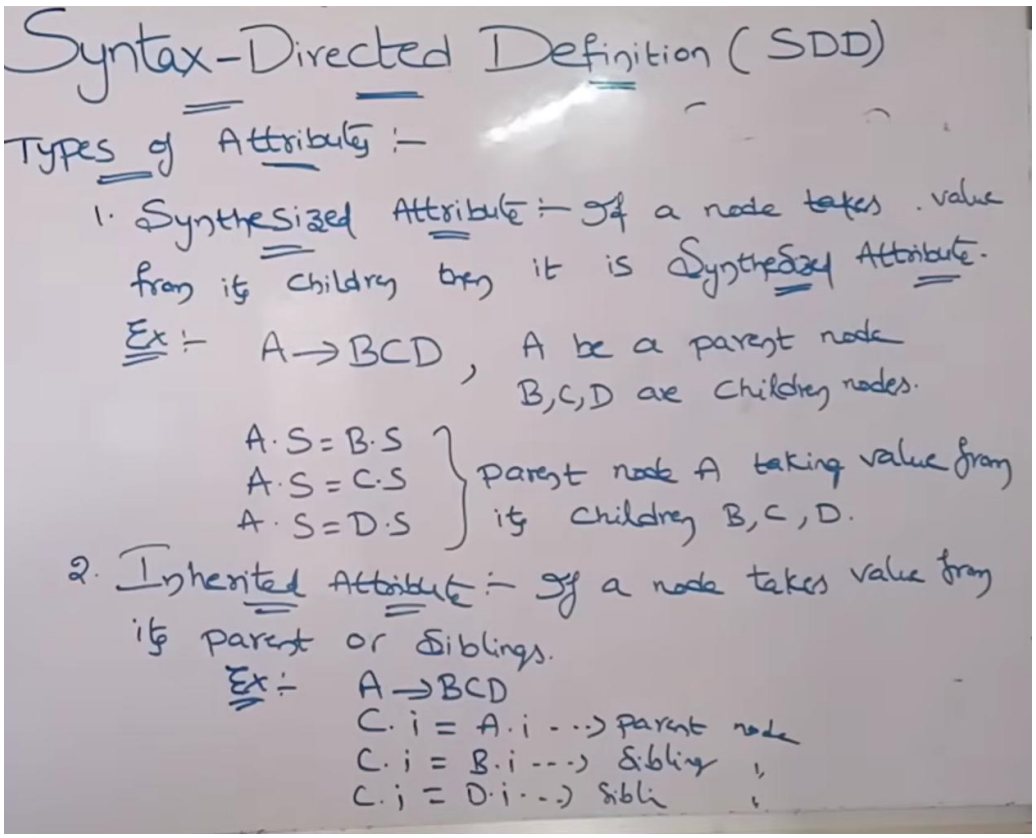
Playlist Link:

https://m.youtube.com/playlist?list=PLXj4XH7LcRfC9pGMWuM6UWE3V4YZ9TZzM





Syntax-Directed Definition (SDD)

Types of SDD:-

1. S-Attributed SDD (or) S-Attributed Definitions (or) S-Attributed grammar
2. L- " " " L- " " " L- " "

S-Attributed SDD

1. A SDD that uses only Synthesized Attributes is called as S-Attributed SDD.

   Ex:- $A \rightarrow BCD$
   $A.S = B.S$
   $A.S = C.S$
   $A.S = D.S$

2. Semantic Actions are always placed at right End of the production. It is also called as "postfix SDD".
3. Attributes are Evaluated with Bottom-up parsing.

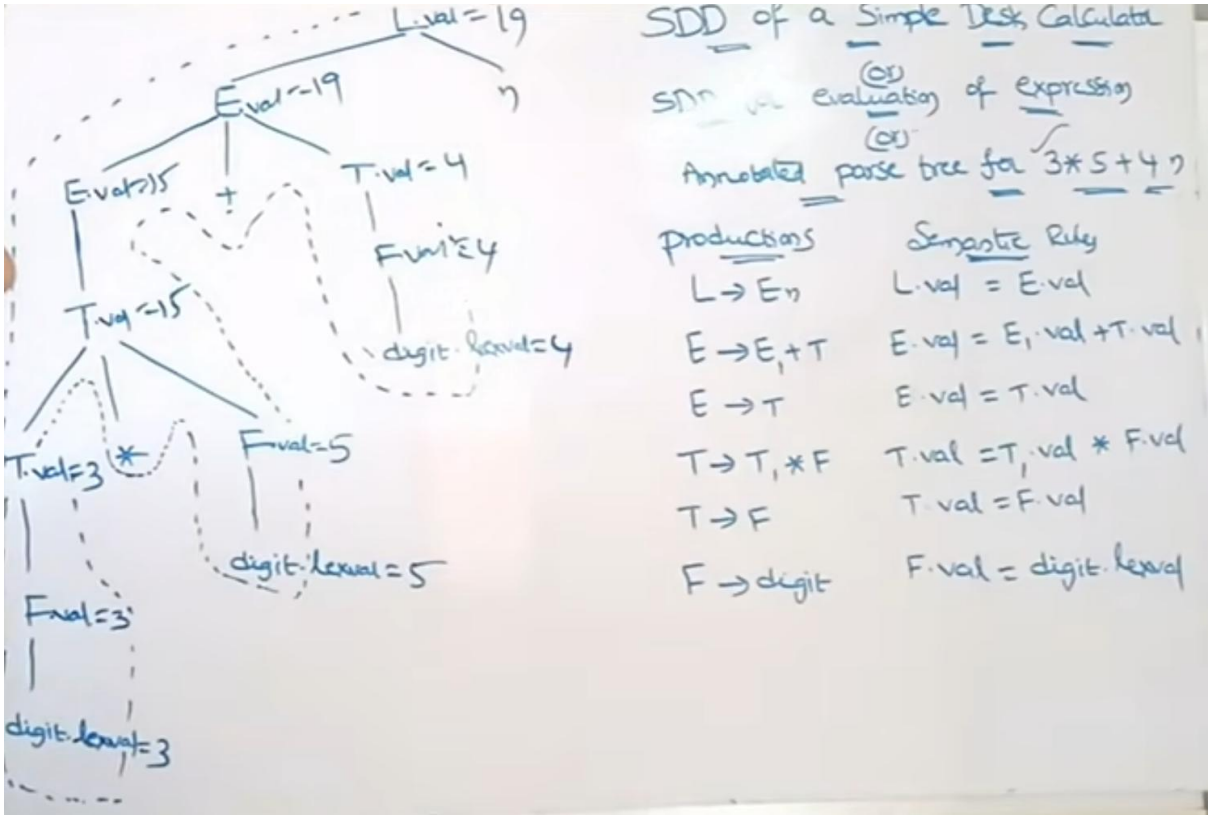
L-Attributed SDD

1. A SDD that uses both Synthesized & Inherited Attributes is called as L-Attributed SDD but each inherited Attribute is restricted to inherit from parent or left Sibling only.

   Ex:- $A \rightarrow XYZ$ { $Y.S = A.S$, $Y.S = X.S$, $Y.S = Z.S$ }

2. Semantic Actions are placed anywhere on R.H.S.
3. Attributes are Evaluated by traversing parse tree depth first, left to right order.

SDD of a Simple Desk Calculator:



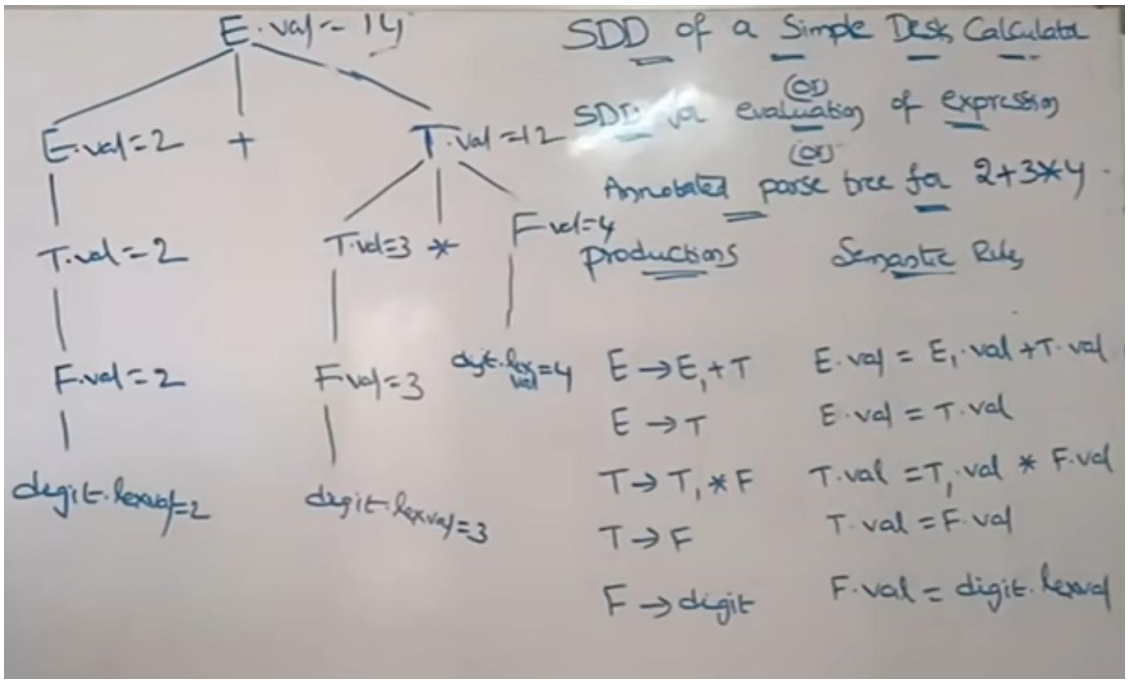
SDD of a Simple Desk Calculator

SDD for evaluation of expressions (or) Annotated parse tree for 3*5+4n

Productions	Semantic Rules
$L \rightarrow En$	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow digit$	$F.val = digit.lexval$

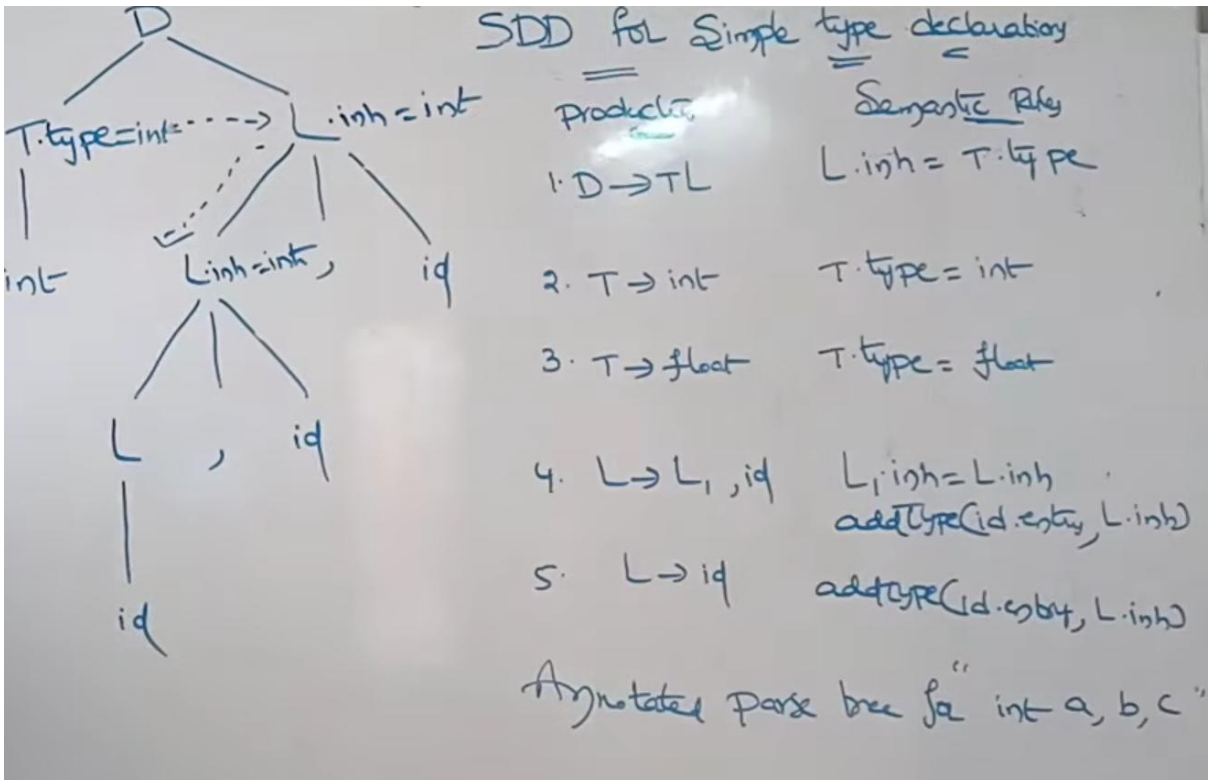
E1 is same as E, we write E1 to differentiate E and E1 on left and right

Another example:



SDT and Annotated Parse Tree:

Ex: int a, b, c



Ex: int a[2][3]

Construct yourself

Productions:

A→ B C

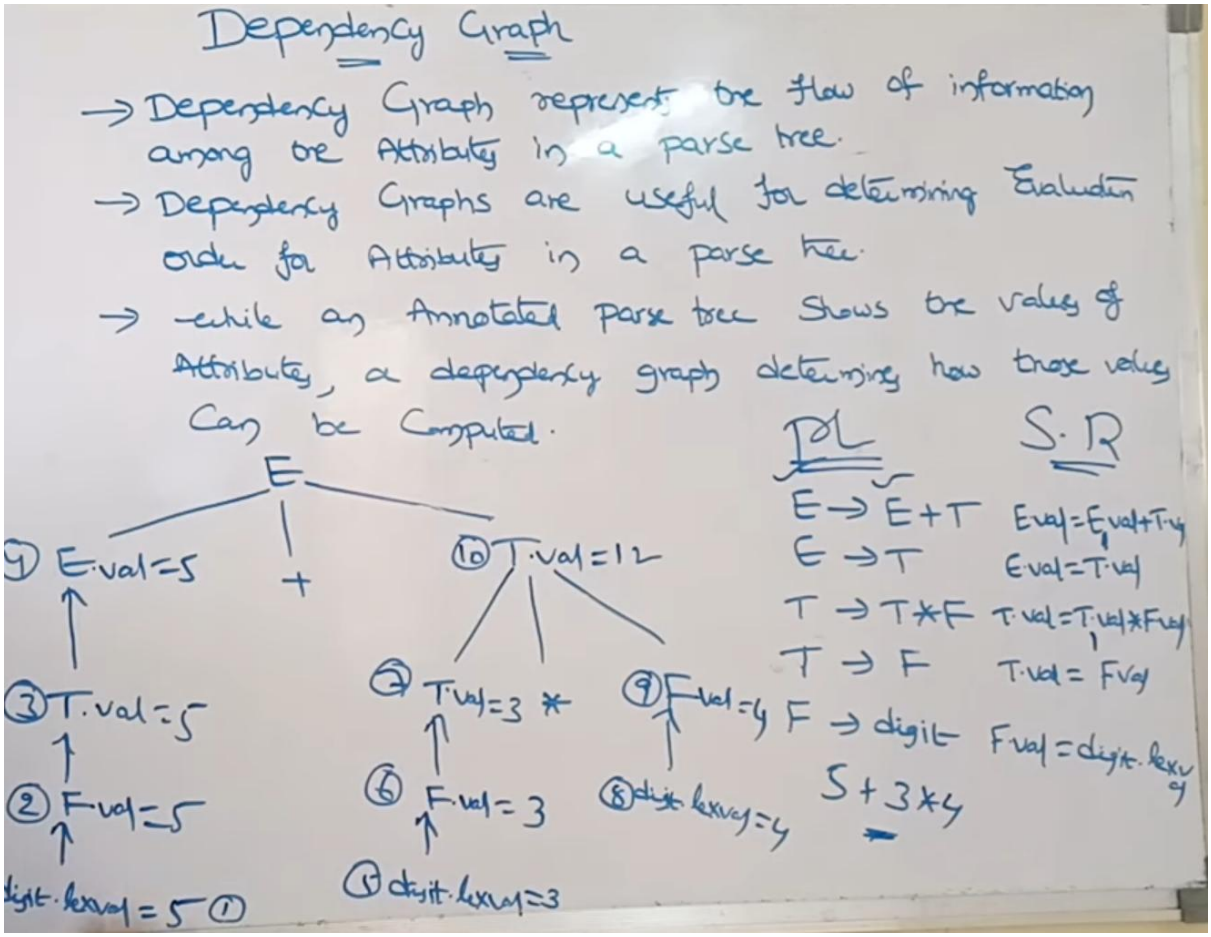
B→ int

B→ float

C→ C1[num]

C→ id

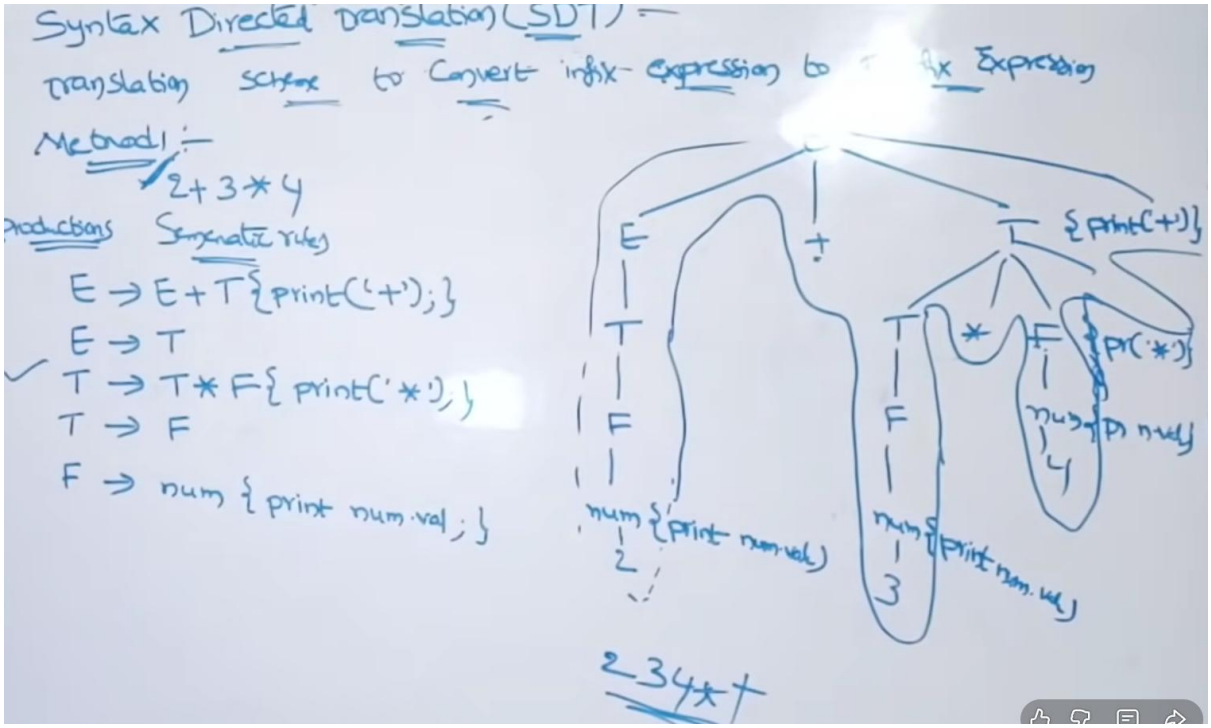
Dependency Graph:



Syntax Directed Translation

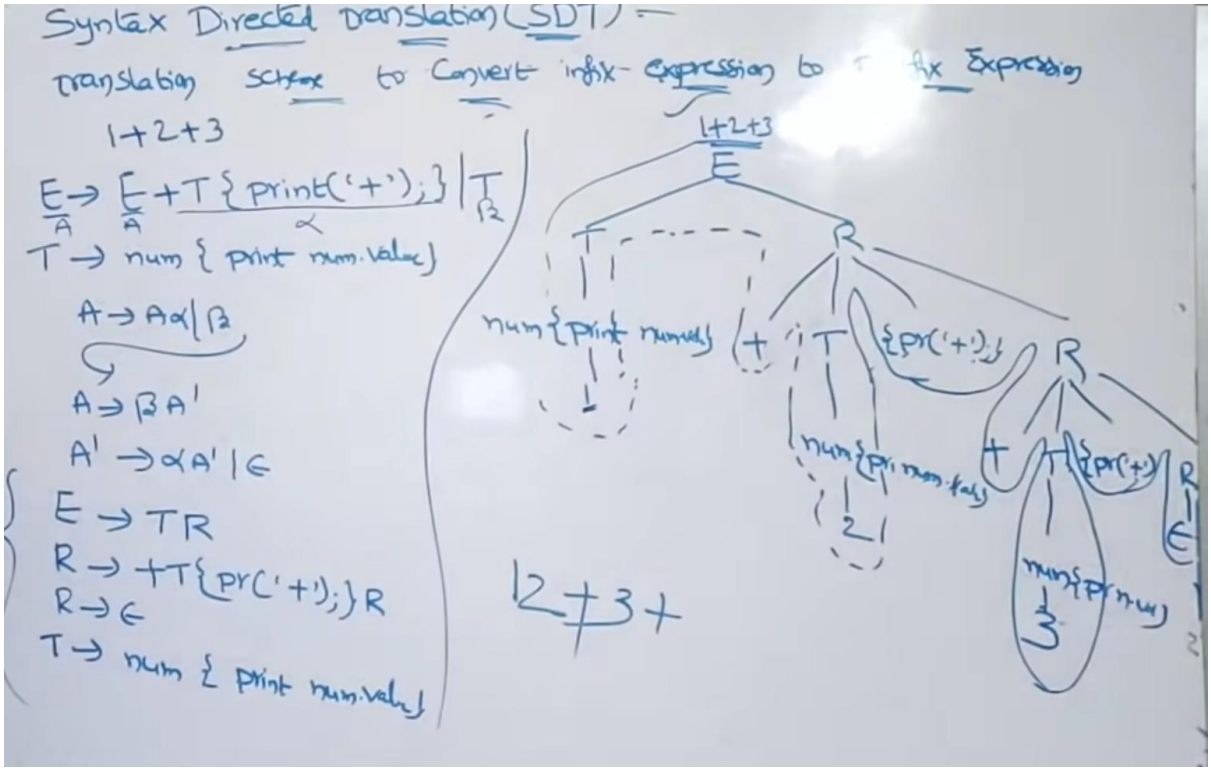
Infix to Postfix:

Ex: 2+3*4

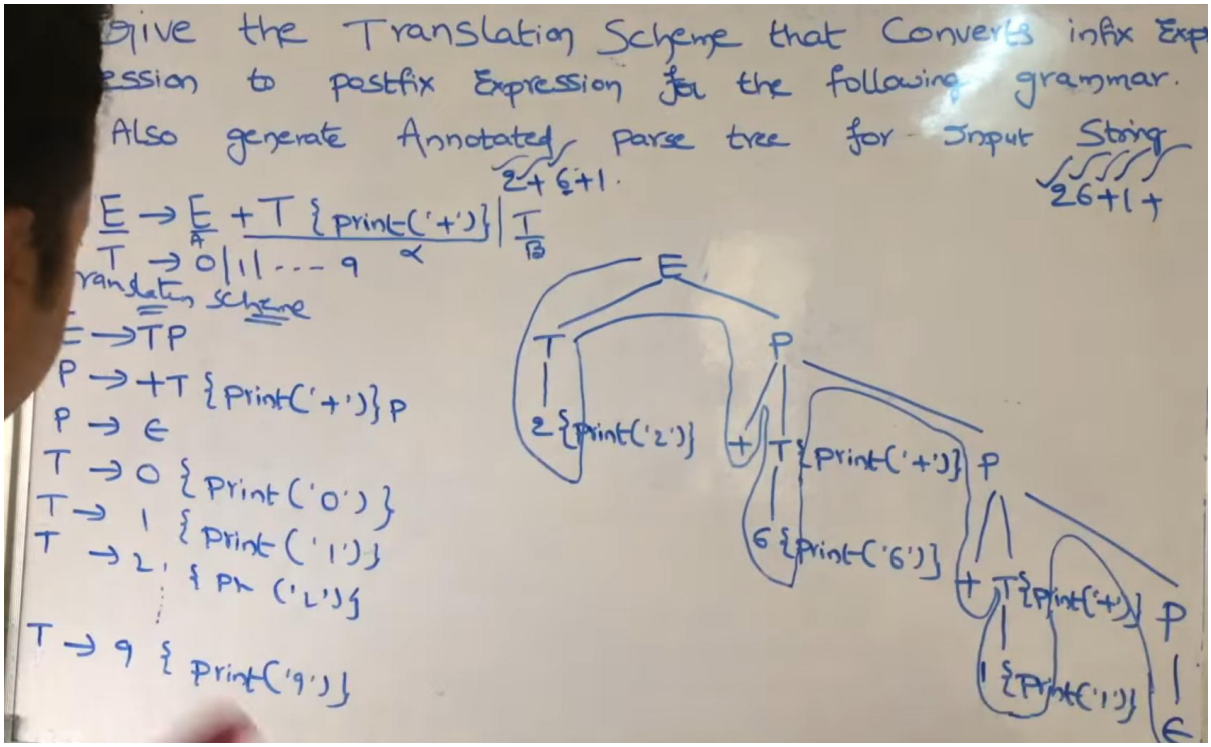


Ex: 1+2+3

**Eliminate Left Recursion here



Ex: 1+2+3 (Draw the productions with nums)

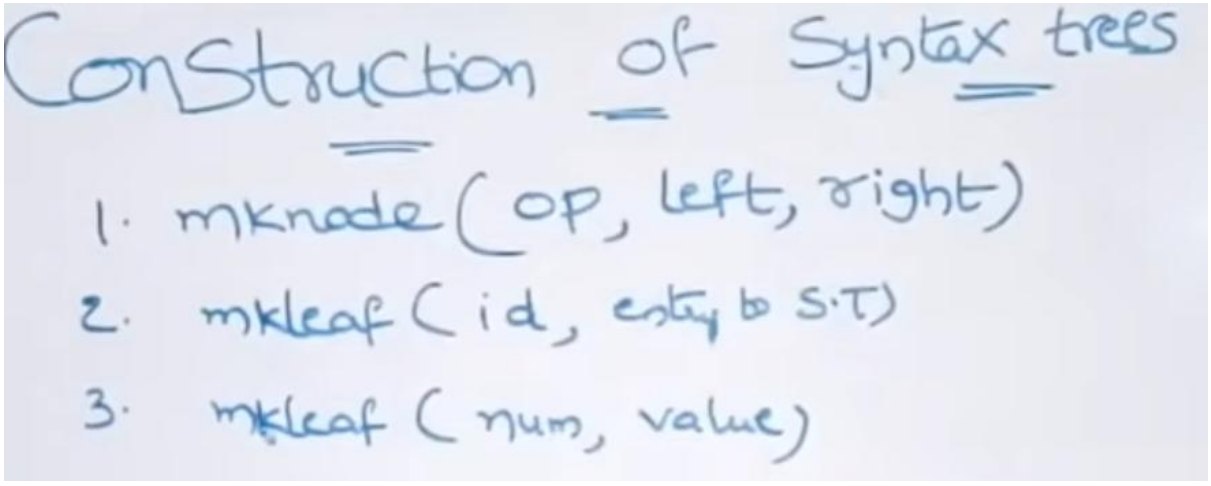


For Infix to Prefix, write the print statements in the start of production

E -> {print(‘+’)} E1 + T

Syntax Trees:

Functions used here



Mknode -> node is always an Operator, made with left and right child address

Mkleaf -> leaf is always a terminal (symbol), with an entry to symbol table (S.T) address/pointer.

Mkleaf -> number, and its value

Examples:

Ex: x*y-5+z

Steps:

1. Eval each operation by operation (x*y), then ((x*y)-5), then ((x*y-5)+z)
2. Mark them as some function pointers (p0, p1, p2, etc)
3. After a node is created, store that in a pointer, and use that pointer as params
4. Construct Syntax Tree with productions and Semantic Rules

Construction of Syntax trees

Ex:- x*y-5+2

Steps for construction of Syntax tree for x*y-5+2

symbol	operation
x	p1 = mkleaf(id, entry-x)
y	p2 = mkleaf(id, entry-y)
*	p3 = mknode(*, p1, p2)
5	p4 = mkleaf(num, 5)
-	p5 = mknode(-, p3, p4)
2	p6 = mkleaf(id, entry-2)
+	p7 = mknode(+, p5, p6)

SDD for x*y-5+2

Prod	Semantic R
$E \to E_1 + T$	E.node = mknode(+, E_1.node, T.node)
$E \to E_1 - T$	E.node = mknode(-, E_1.node, T.node)
$E \to E_1 * T$	E.node = mknode(*, E_1.node, T.node)
$E \to T$	E.node = T.node
$T \to id$	T.node = mkleaf(id, id.entry)
$T \to num$	T.node = mkleaf(num, num.val)

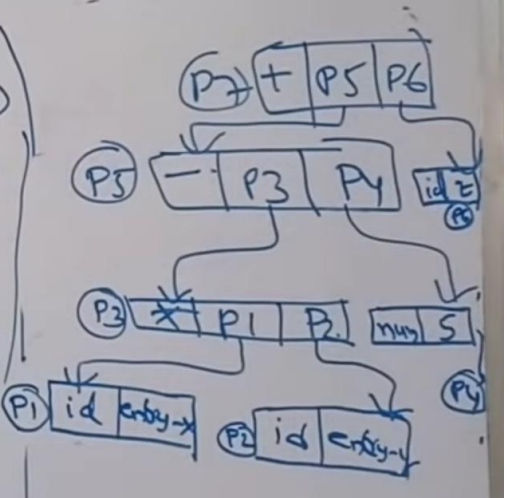
x*y-5+2
a-4+c

Construction of Syntax trees

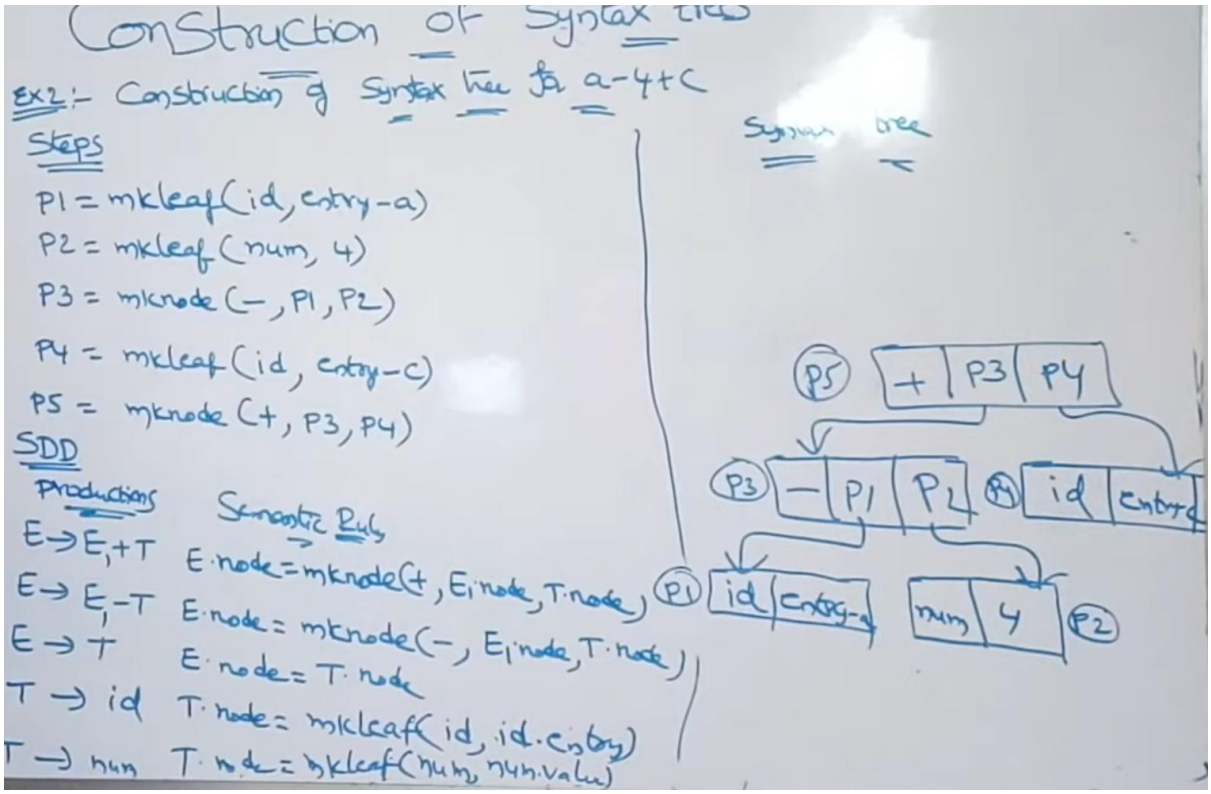
Ex:- x*y-5+2

Steps for construction of Syntax tree for x*y-5+2

symbol	operation
x	p1 = mkleaf(id, entry-x)
y	p2 = mkleaf(id, entry-y)
*	p3 = mknode(*, p1, p2)
5	p4 = mkleaf(num, 5)
-	p5 = mknode(-, p3, p4)
2	p6 = mkleaf(id, entry-2)
+	p7 = mknode(+, p5, p6)



Ex: a – 4 + c



Forms of Intermediate Code:

3 Address Notation:

Uses at most 3 variables in one Step and only 1 operator

Ex: t1 = a + b, t1 = -c

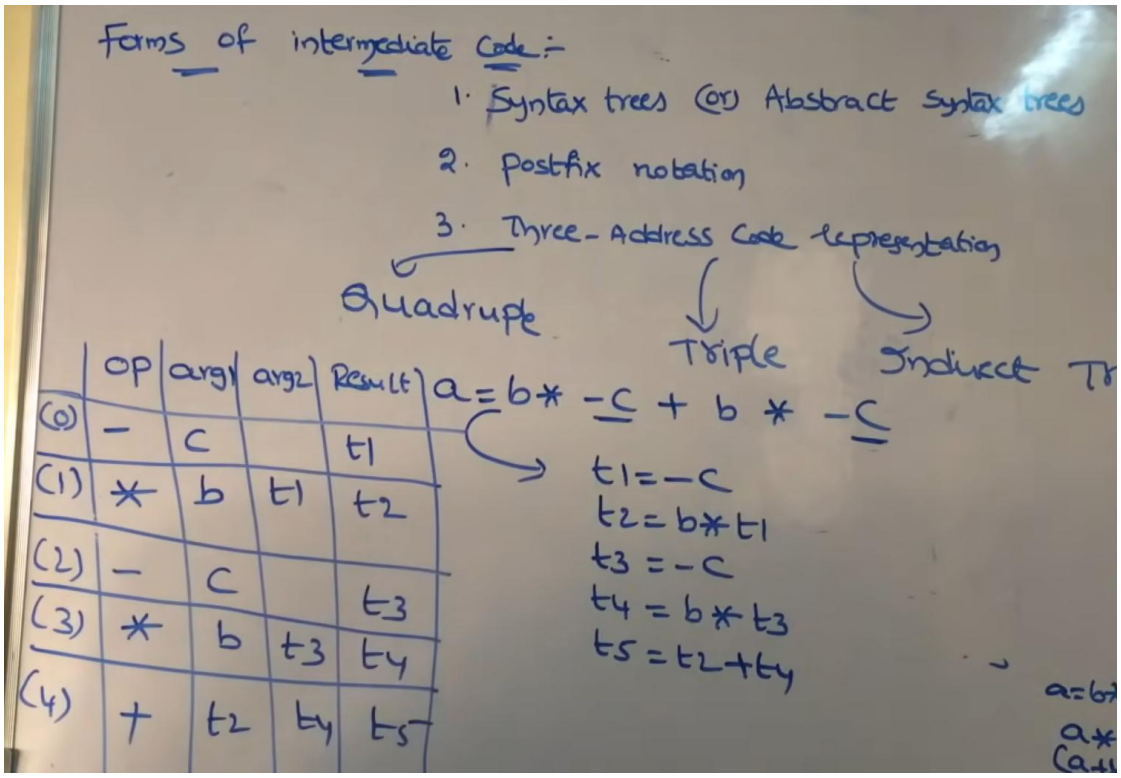
Types:

1. Quadruple (stores as ---
   Step Address -> (operand, arg1, arg2, result))
2. Triple (stores as ---
   Step Address - > (operand, arg1, arg2))
   (but uses the results from the step address space instead of result column)
3. Indirect Triple (same as Triple, but the Step Address are pointers pointing to Step Address of Triple)

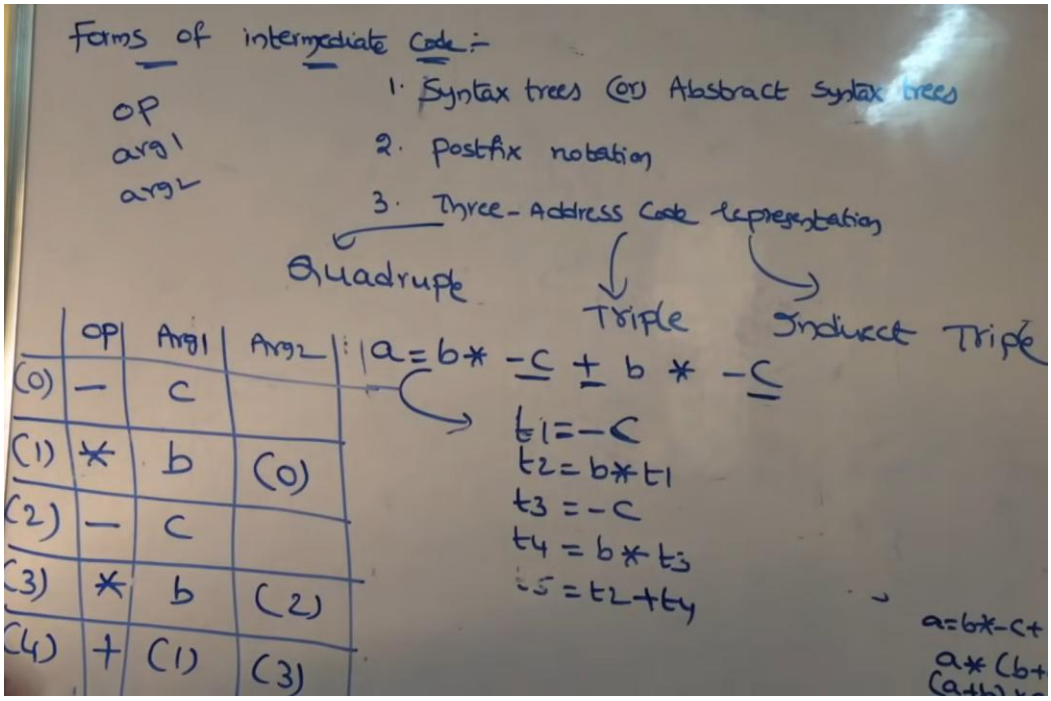
Ex: a = b * -c + b * -c

ALWAYS USE BODMAS WHILE WRITING THESE 3-ADDRESS FORMS (Left-Right Precedence when only +/-, or */'/'

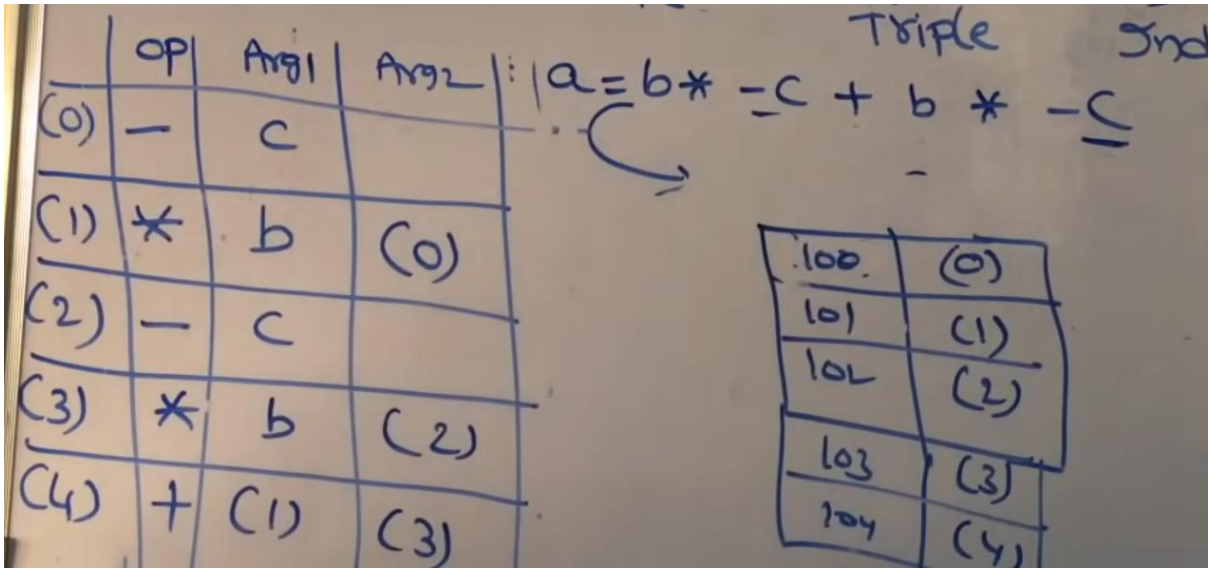
Quadruple Notation:



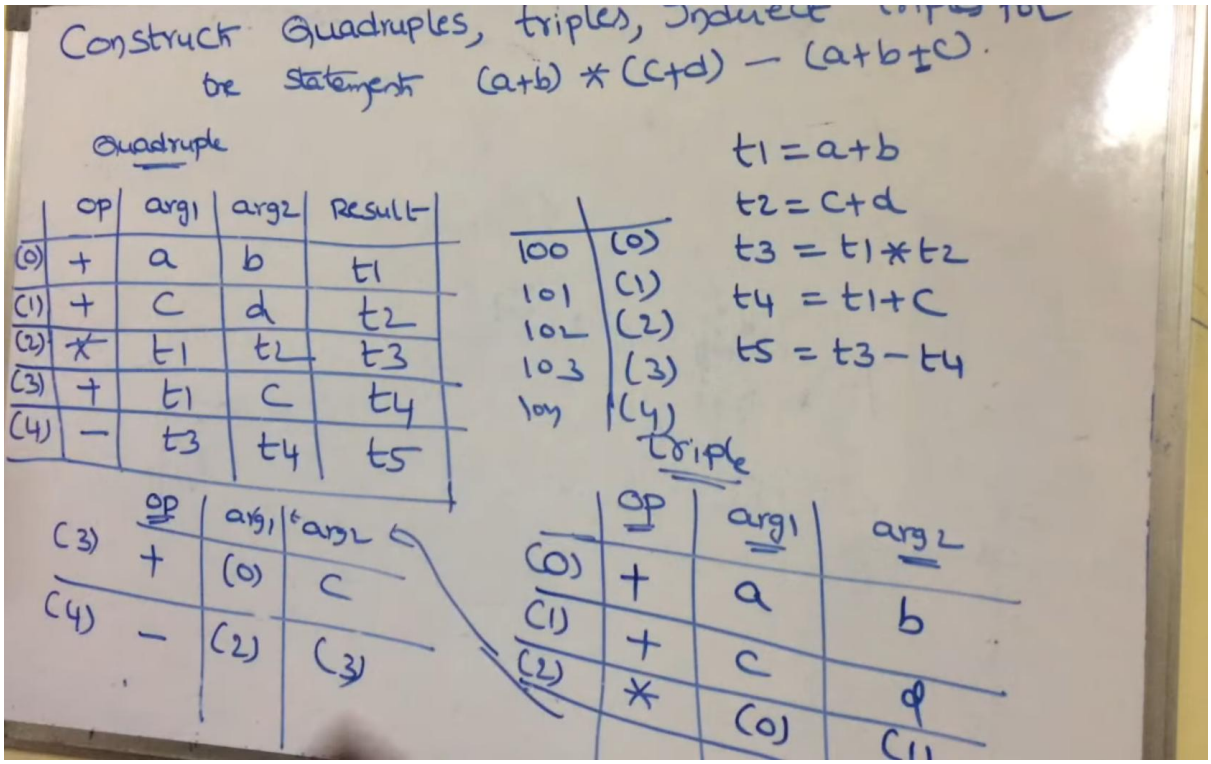
Triple:



Indirect Triple:



Ex: (a+b) * (c+d) – (a+b+c)

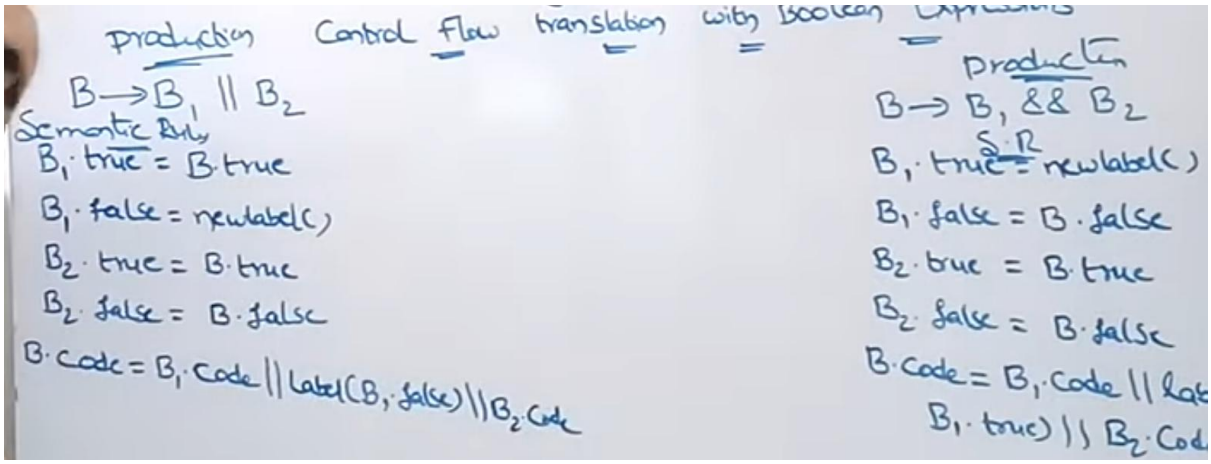


SDD Translation of 3 Address Code for Boolean Expressions

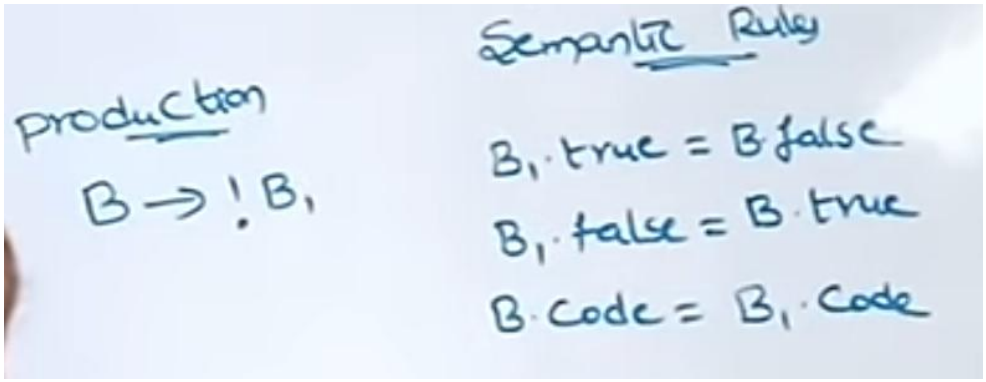
Rules:

1. Evaluate the True and False cases for each Boolean on RHS
2. If the final value is undecided, generate a newLabel()
3. After all True False for all Boolean operands is done, now write the final Code for the expression.
4. First write from LHS to RHS, always write Bx.code and next if it has some newLabels || it with Label(Bx.True/False)

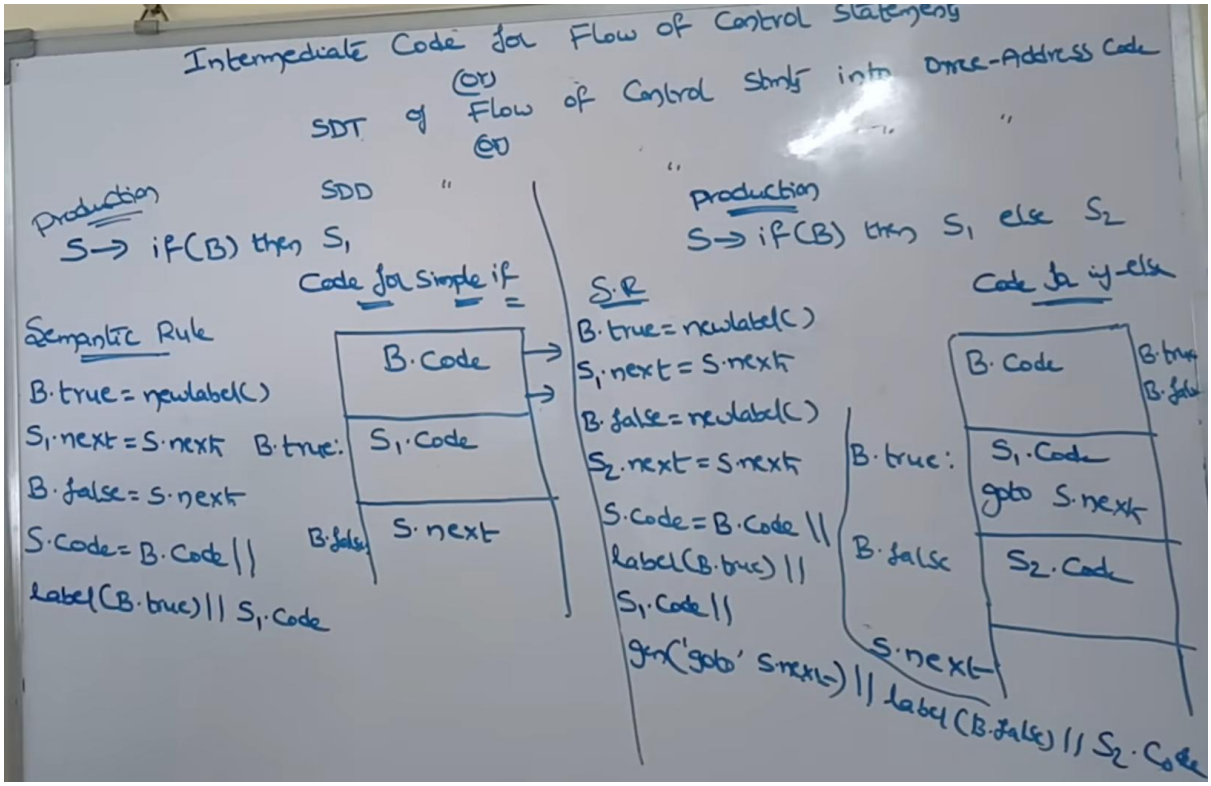
Ex: B = B1 || B2 and B = B1 && B2



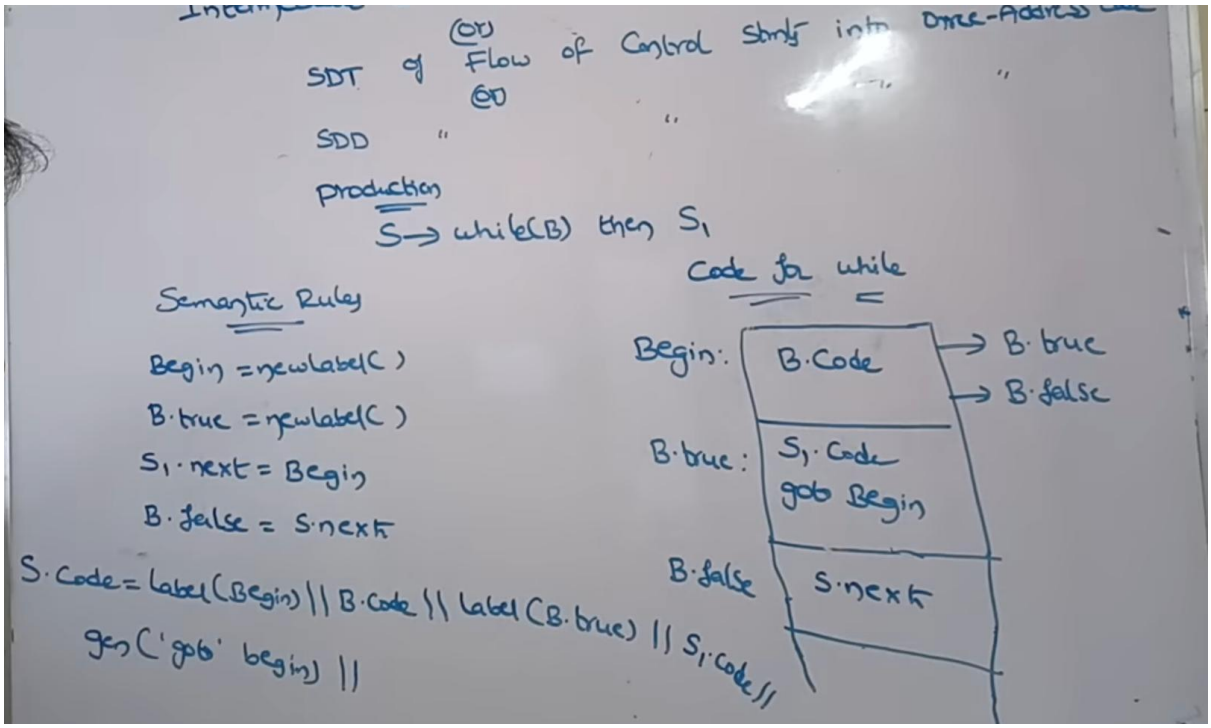
Ex: B = !B1 (no need for Label(B1.) since there is no newLabel()



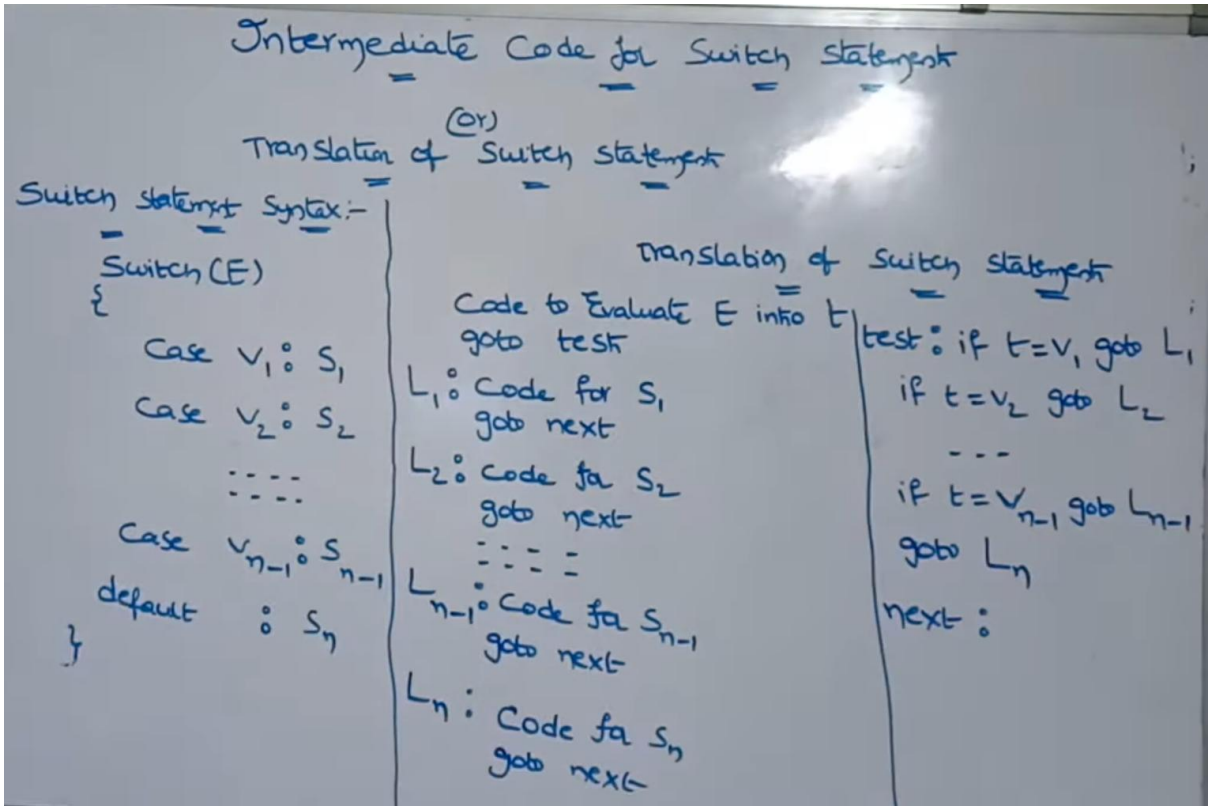
Intermediate Code (IC) for Simple If and If Else Statements:



IC for while loop:



IC for switch Statement:



BackPatching

Process of evaluating by keeping blank spaces for unknown expressions and then filling it by back tracking.

Steps:

1. Go Left to Right, and for each process, check True and False case,
2. Here we leave blank spaces, since we don’t know the address of the final Statement or the next executable statement.

   Ex: for x<100, if its True, we don’t have address of True yet so blank.

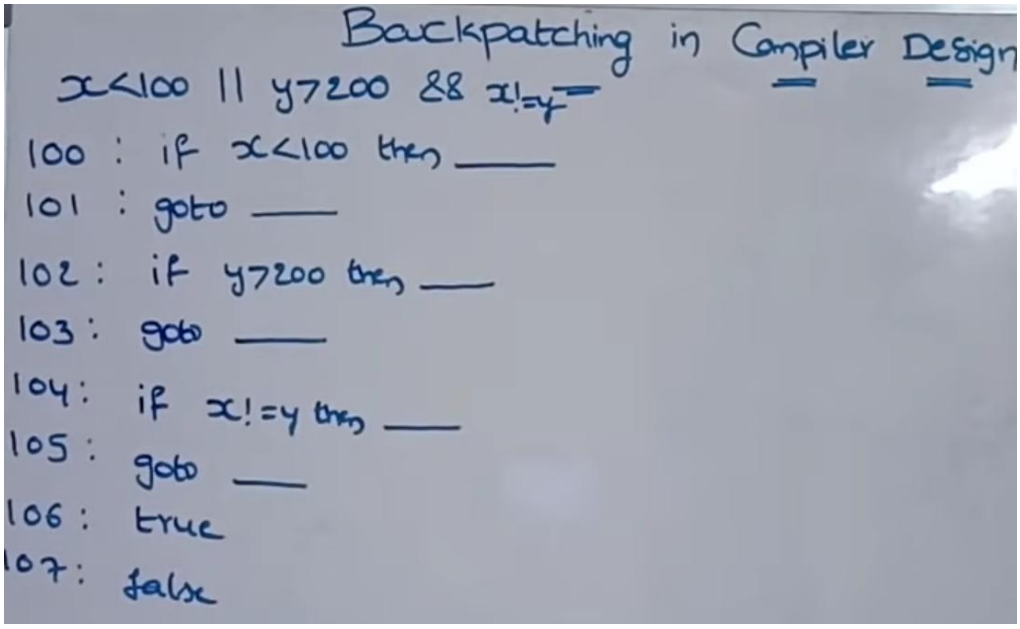
   If its False, we goto the next expression y>200, but the address is unknown

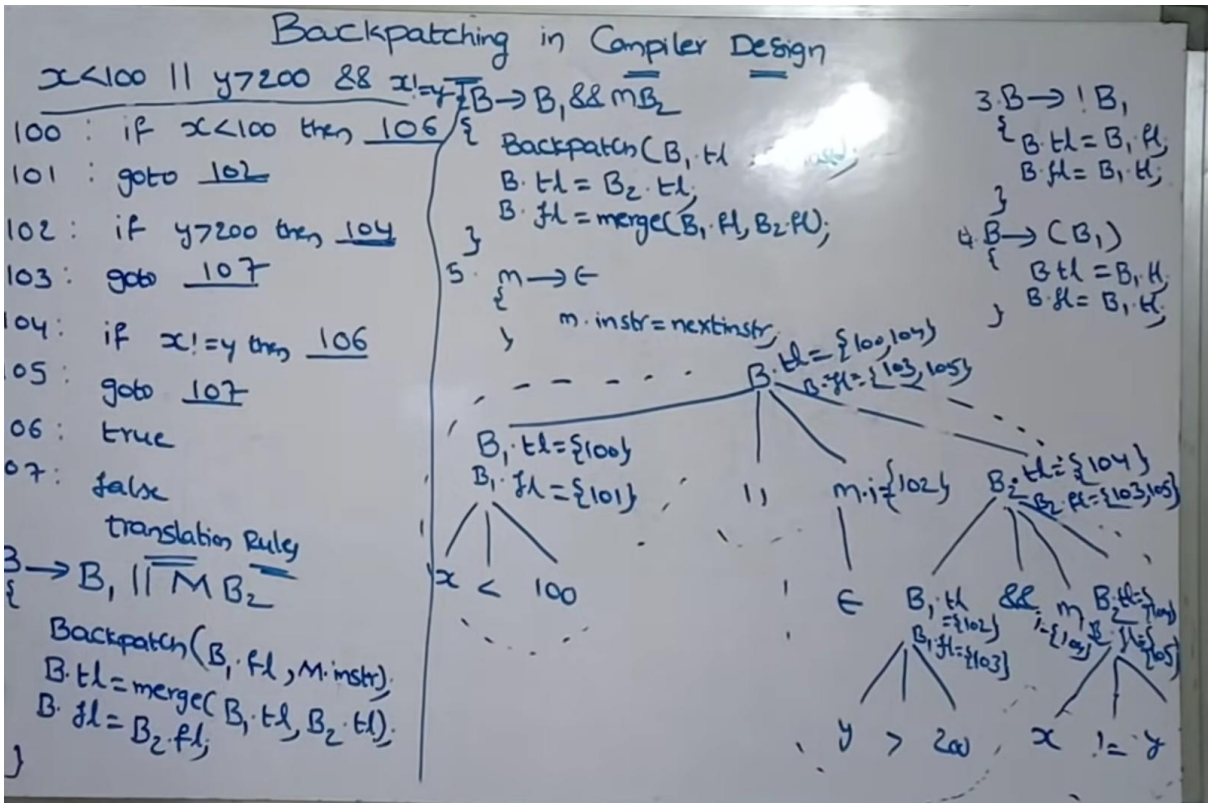
3. After all expressions are done, you write addresses for True and False, and then back track.
4. Write Translation Rules for productions of all Boolean Expressions in Backpatch terms. B -> B1 || M B2, B ->B1 && B2, B -> !B1, B->B1, M-> e
5. Construct Annotated Parse Tree using these Productions.

Ex: x<100 || y>200 && x!=y

Backpatch() is called when u don’t know the address of the next instruction.

Merge() is called for two or more possible cases.





Main Issues of Code Generator

1. **Input to the Code Generator**
    - **The Issue:** The generator must be "front-end agnostic." If the Intermediate Representation (IR) is too high-level, the generator lacks the detail needed for hardware optimization. If it is too low-level, the generator becomes tied to one specific language.
    - **Why it's a challenge:** The generator must handle various IR forms (like DAGs or Quadruples) while maintaining all semantic information from the source, ensuring no logic is lost during the translation to machine code.
2. **Target Program Output**
    - **The Issue:** The "issue" is the trade-off between **portability** and **performance**.
    - **Why it's a challenge:** Generating **Absolute Code** is easy but inflexible (the program can only load at one memory address). Generating **Relocatable Code** is more complex because the generator must produce extra information for the Linker to resolve external references.
3. **Instruction Selection**
    - **The Issue: Redundancy and Complexity.** Most hardware architectures provide multiple ways to do the same thing.
    - **Why it's a challenge:** If the generator is naive, it produces "heavy" code. For example, using a MULT instruction for x * 2 is an issue because a SHIFT LEFT instruction is significantly faster. The generator must constantly weigh the cost (cycles) of different instruction sequences.
4. **Register Allocation and Assignment**
    - **The Issue: Limited Resources.** Registers are the fastest storage, but there are rarely enough of them for all variables.
    - **Why it's a challenge:** This is an NP-complete problem. The "issue" is **Register Spilling**: if you run out of registers, you must move data to the slow RAM (memory). Deciding *which* variable to "kick out" of a register requires complex algorithms to avoid a massive performance hit.
5. **Choice of Evaluation Order**
    - **The Issue: Dependency Management.** Some instructions depend on others, and the order dictates how many registers are needed at once.
    - **Why it's a challenge:** By picking a poor evaluation order, the generator might force a "register spill" that could have been

avoided. The goal is to find an order that minimizes temporary storage while maintaining the original logic of the code.

6. Memory Management

- **The Issue: Mapping Abstraction to Reality.** Source code uses variable names (symbols), but hardware uses byte offsets.
- **Why it's a challenge:** The generator must manage the **Runtime Stack** and **Heap**. It has to decide exactly where a variable lives (Activation Records). If it fails to manage these addresses correctly, the program will suffer from buffer overflows or memory corruption.

Different types of instructions used in a simple target machine model

1. Detailed Target Machine Instructions

A simple target machine model (often a Load/Store architecture) assumes that most operations are performed in registers. The instructions are categorized as follows:

- **Load and Store:**
  - LD r, m: Loads content of memory location m into register r.
  - ST m, r: Stores content of register r into memory location m.
  - **Addressing Modes:** Includes absolute (e.g., 100), register (R1), register-indirect (e.g., *R0), and indexed (e.g., 100(R0)), literal
- **ALU Operations:**
  - OP dst, src1, src2: Performs an operation (ADD, SUB, MUL, DIV, OR, AND) on two source registers and stores the result in a destination register.
  - Example: ADD R1, R2, R3 implements $R1 = R2 + R3$.
- **Branching (Control Flow):**
  - **Unconditional:** BR L forces execution to jump to label L.
  - **Conditional:** BLTZ r, L (Branch if Less Than Zero). These rely on testing the value in a register or a status flag.

Briefly explain each of them

a. Common sub-expression elimination
b. Dead code elimination

a. Common Sub-expression Elimination (CSE)

CSE aims to reduce redundant computations by identifying expressions that appear multiple times where the values of the operands do not change between occurrences.

- **Local CSE:** Occurs within a "Basic Block." If E was previously computed, and the variables in E have not been assigned new values, the second occurrence of E is replaced by the saved value.
- **Global CSE:** Occurs across different basic blocks within a procedure, requiring "Data-Flow Analysis" to ensure the value is valid across all paths.
- **Benefit:** Reduces CPU cycles spent on redundant arithmetic.

b. Dead Code Elimination (DCE)

Dead code is divided into two categories:

1. **Unreachable Code:** Code that can never be executed (e.g., code following a return statement or inside an if(false) block).
2. **Useless Code:** Code that executes but computes a value that is never "live"—meaning it is never read by any subsequent instruction before being overwritten or before the program ends.

- **The Process:** Compilers use "Liveness Analysis" to determine if a variable's value is needed. If a statement like x = y + z is found and x is never used again, the instruction is deleted.
- **Benefit:** Decreases memory footprint and improves instruction cache utilization.

Explain the code generation algorithm. Generate target code for a given arithmetic expression.

X = a/(b+c)–d*(e+f)

A code generation algorithm must efficiently translate intermediate code into machine instructions while managing limited resources like registers. The most common approach involves using a **descriptor-based algorithm** that tracks the state of registers and memory.

1. The Code Generation Algorithm

The algorithm processes a sequence of three-address statements and performs the following for each operation ($x = y \text{ op } z$):

- **Register Selection (getReg):** It first determines where to perform the calculation. It looks for a register that currently holds y, an empty register, or a register whose current value is no longer needed (to avoid "spilling" to memory).
- **Register Descriptors:** It maintains a map of what is currently stored in each register (e.g., R_1 contains variable a).
- **Address Descriptors:** It tracks the location(s) where the current value of a variable can be found (e.g., the value of x is in both R_1 and memory location M_x).
- **Instruction Generation:** It issues a Load if y is not in a register, then generates the Operator instruction using z, and updates the descriptors to show that the result x is now in the chosen register.

Generating Target Code for the Expression

Expression: X = a / (b + c) - d * (e + f)

First, we convert the expression into **Three-Address Code (TAC)** to simplify the generation process:

1. t_1 = b + c
2. t_2 = a / t_1
3. t_3 = e + f
4. t_4 = d * t_3
5. X = t_2 - t_4

Target Code (Assembly):

Using a standard machine model where R_n represents registers:

Three-Address Code	**Machine Instructions**	**Description**

t_1 = b + c	LD R1, b
	
	ADD R1, R1, c
		Load b into R_1, add c. R_1 now holds t_1.

$t_2 = a / t_1$	LD R2, a DIV R2, R2, R1	Load a into R_2, divide by R_1. R_2 now holds t_2.

$t_3 = e + f$	LD R3, e
	
	ADD R3, R3, f
		Load e into R_3, add f. R_3 now holds t_3.

$t_4 = d * t_3$	LD R4, d
	
	MUL R4, R4, R3
		Load d into R_4, multiply by R_3. R_4 now holds t_4.

$X = t_2 - t_4$	SUB R2, R2, R4 ST X, R2	Subtract R_4 from R_2. Store final result in memory X.

Explain code generation for simplified procedure calls

In compiler design, generating code for procedure calls involves managing the transition between the **caller** (the function making the call) and the **callee** (the function being called). This is handled via a **calling sequence**, which coordinates the allocation of memory on the runtime stack.

1. The Runtime Stack and Activation Records

Each time a procedure is called, a block of memory called an **Activation Record (or Stack Frame)** is pushed onto the stack. This record stores all information necessary for that specific execution of the procedure.

- **Parameters:** Values passed by the caller.
- **Return Address:** The location in the code where execution resumes after the call.
- **Saved Machine Status:** The state of registers before the call.
- **Local Data:** Variables declared within the procedure.
- **Temporaries:** Intermediate values calculated during expression evaluation.

---

2. The Calling Sequence (Step-by-Step)

The code generator must emit instructions for both the caller and the callee to ensure data is preserved.

A. Actions by the Caller

1. **Evaluate Arguments:** Calculate the values of the actual parameters.
2. **Store Arguments:** Place these values into the activation record of the callee or into specific registers.
3. **Save Return Address:** Save the current Program Counter (PC) so the callee knows where to return.
4. **Jump to Callee:** Execute a branch instruction to the start of the procedure code.

B. Actions by the Callee

1. **Save Registers:** Save any registers that the caller was using (callee-save convention).
2. **Initialize Local Data:** Allocate space for local variables on the stack.
3. **Execute Body:** Run the actual code of the procedure.

C. Returning from the Call

1. **Place Return Value:** Put the result in a designated register (e.g., R_0).
2. **Restore Status:** Reload the saved registers and "pop" the activation record by adjusting the stack pointer.
3. **Branch to Return Address:** Jump back to the caller’s saved address.

---

3. Example Code Generation

Consider a simple call: result = sum(a, b);

Three-Address Code:

1. param a
2. param b
3. t1 = call sum, 2
4. result = t1

Generated Assembly (Target Machine):

Code snippet

```
# Caller side
PUSH a           ; Push first parameter onto stack
PUSH b           ; Push second parameter onto stack
CALL sum         ; Save return address and jump
ADD SP, SP, 8    ; Clean up stack (2 params * 4 bytes)
ST result, R0    ; Store returned value from R0 to 'result'


# Callee side (sum)
sum:
  PUSH BP        ; Save base pointer
  MOV BP, SP     ; Set new base pointer for this frame
  LD R1, 8(BP)   ; Load param 'a' (offset from BP)
  LD R2, 12(BP)  ; Load param 'b'
  ADD R0, R1, R2 ; Perform addition, result in R0
  MOV SP, BP     ; Restore stack pointer
  POP BP         ; Restore base pointer
  RET            ; Return to caller
```

What is Three address code? Construct the basic blocks for the following program segment:

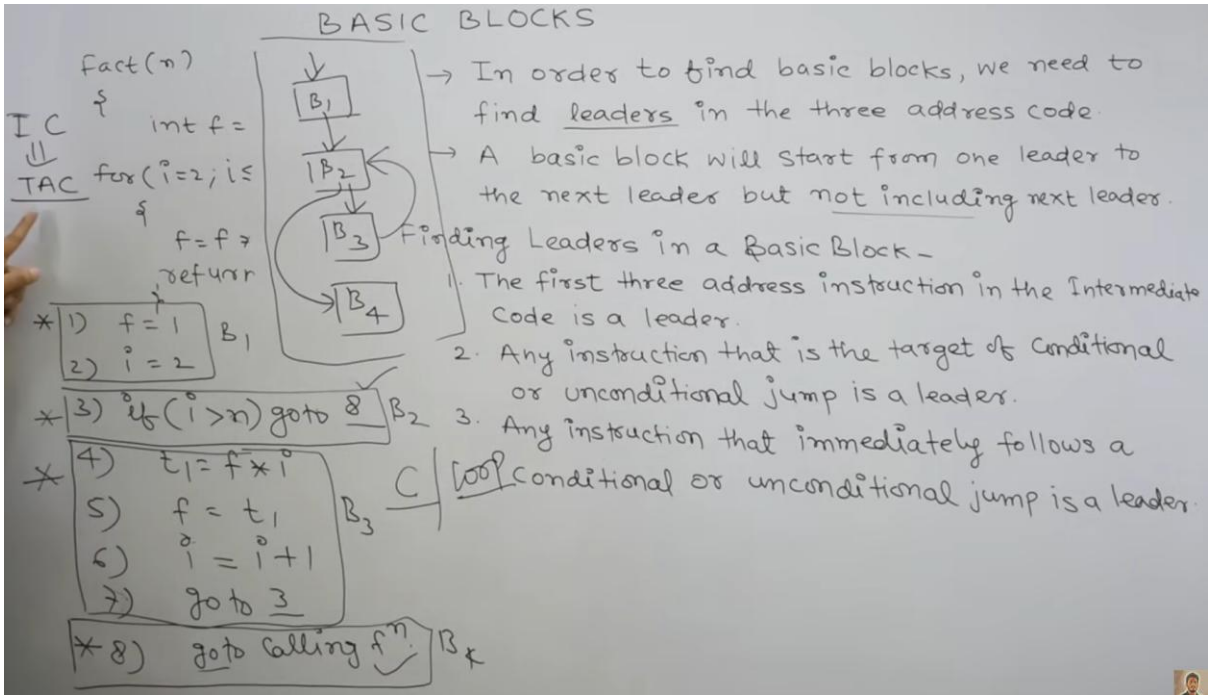
```
rev=0;
while(num>=0) {
	dig= num%10;
	rev=rev*10 +dig;
}
```

Three-Address Code (TAC)

Three-Address Code is an intermediate representation used by compilers where each instruction has at most three operands: two for the input values and one for the result. It typically follows the format:

$x = y \text{ op } z$

It is a linearized representation of a syntax tree. It bridges the gap between high-level languages and low-level machine code by breaking down complex expressions into simple, manageable steps using **temporary variables** ($t_1, t_2, \dots$).



Step 1: Converting to Three-Address Code

Before identifying basic blocks, we must translate the program segment into TAC:

1. rev = 0
2. if num < 0 goto (10) (Exit condition for while)
3. t1 = num % 10
4. dig = t1
5. t2 = rev * 10
6. t3 = t2 + dig
7. rev = t3
8. goto (2) (Loop back to condition)
9. (10) Next Statement

Step 2: Constructing Basic Blocks

A **Basic Block** is a sequence of instructions that enters at the beginning and leaves at the end without any jumps or branches in between (except at the last instruction).

Leaders Identification:

To find the blocks, we identify the "leaders":

- The first instruction is a leader.
- Any instruction that is the target of a jump (e.g., the while condition) is a leader.
- Any instruction immediately following a jump is a leader.

The Basic Blocks:

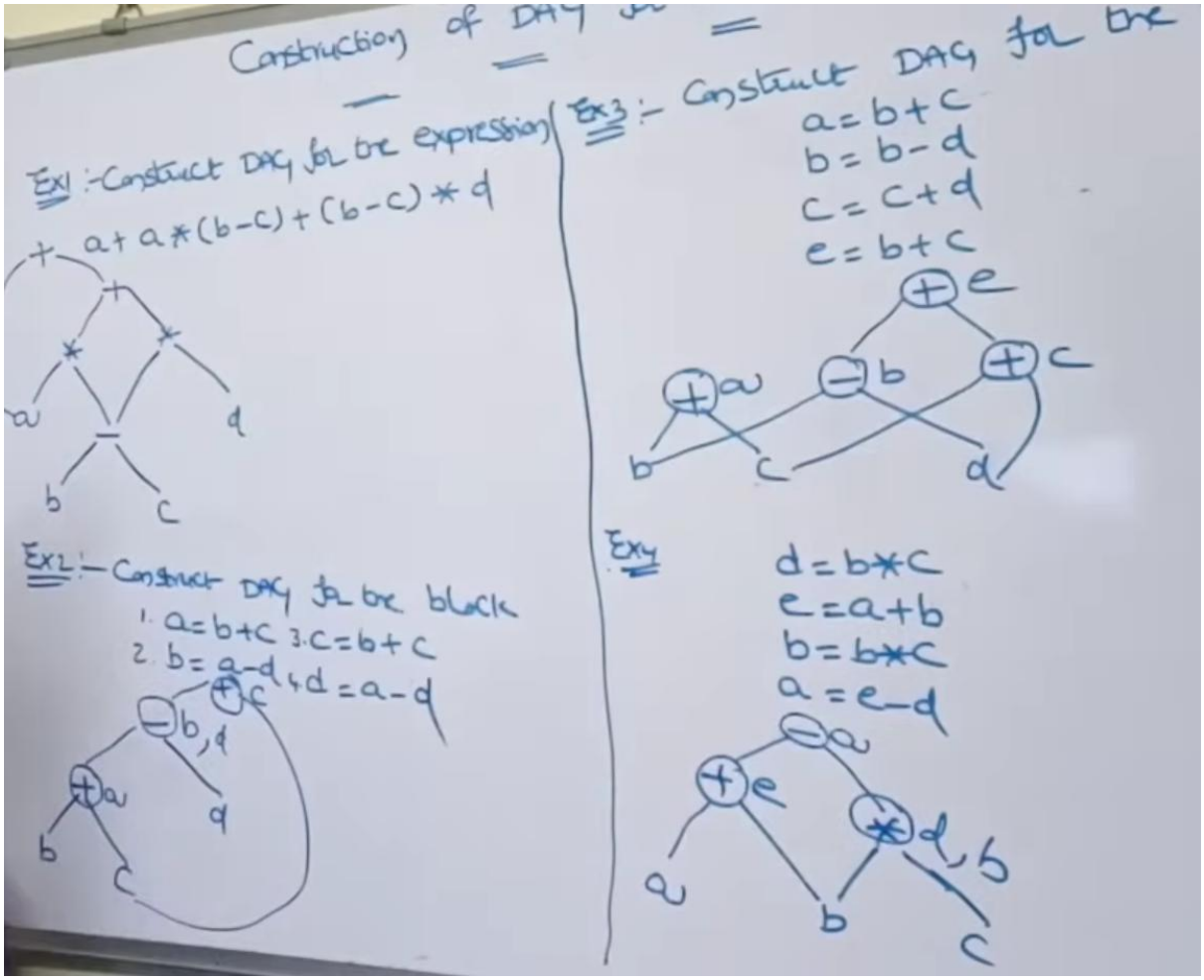
Block	Instructions	Description
B1	(1) rev = 0	**Entry Block:** Initializes the variable.
B2	(2) if num < 0 goto (10)	**Header Block:** Contains the loop condition.
B3	(3) t1 = num % 10	**Loop Body:** Executes the logic and jumps back to the header.

Block	Instructions	Description
	(4) dig = t1 (5) t2 = rev * 10 (6) t3 = t2 + dig (7) rev = t3 (8) goto (2)	
B4	(10) Next Statement	**Exit Block:** Execution continues here after loop termination.

Control Flow Graph (CFG) Relationship

- **B1** flows into **B2**.
- **B2** has two edges: one to **B3** (if true/loop continues) and one to **B4** (if false/exit).
- **B3** has a "back-edge" returning to **B2**.

Explain the construction of a Directed Acyclic Graph (DAG) for a basic block.



A **Directed Acyclic Graph (DAG)** is a specialized data structure used to represent the structure of a basic block. It helps the compiler identify common sub-expressions, determine which values are computed more than once, and see which variables are used outside the block.

Rules for DAG Construction

- **Leaves:** Represent initial values of variables or constants (labels).
- **Internal Nodes:** Represent operators (e.g., +, *, %).
- **Nodes can have multiple labels:** If multiple variables hold the same value, they are attached to the same node.

DAG Construction for Block B3

From the previous example, the loop body (Block B3) contains:

1. t1 = num % 10
2. dig = t1
3. t2 = rev * 10

4. t3 = t2 + dig
5. rev = t3

Step-by-Step Construction:

1. **Node 1 (%):** Create a leaf for the current value of num and a leaf for the constant 10. Create an internal node % connecting them. Label this node t1 and dig.
2. **Node 2 (*):** Create a leaf for the current value of rev and reuse the constant leaf 10. Create an internal node *. Label this node t2.
3. **Node 3 (+):** Connect the result of **Node 2** (t2) and **Node 1** (dig) with a + operator node.
4. **Update Labels:** Label the final + node as t3 and rev.

Why is the DAG useful for this block?

- **Optimization:** If the code had another line like x = num % 10, the DAG would not create a new node; it would simply add the label x to Node 1.
- **Dead Code Elimination:** If t2 or t3 are never used again, the compiler can see they are just intermediate steps to reach the final value of rev.
- **Instruction Ordering:** The DAG clearly shows that num % 10 and rev * 10 are independent and can be calculated in any order, but the final + must wait for both.

Explain type checking and its importance in a compiler.

Type checking is a core component of the semantic analysis phase of a compiler.[1] It is the process of verifying that the data types of operands in a program are consistent with the operations being performed on them according to the language's formal grammar.[2]

1. How Type Checking Works

The compiler uses a **Type System**, which is a set of rules that assign types to various parts of the program (variables, expressions, functions).[3]

- **Type Synthesis:** The compiler builds up the type of an expression from the types of its sub-expressions. For example, if f is a function that returns an int, the expression $f() + 10$ is synthesized as an int.
- **Type Inference:** The compiler automatically determines the type of a variable based on its usage or initialization (common in languages like Python, Swift, or C++ with auto).[4]

---

2. Static vs. Dynamic Type Checking

- **Static Type Checking:** Performed at **compile-time**. It catches errors before the program ever runs.[5] (e.g., C, C++, Java).
- **Dynamic Type Checking:** Performed at **runtime**.[6] The compiler generates code that checks types while the program is executing. (e.g., Python, JavaScript).

---

3. Importance of Type Checking

Type checking is critical for several technical and functional reasons:

- **Error Detection:** It prevents "type crashes" where the computer tries to perform nonsensical operations, such as adding a string to an integer ("hello" + 5) or calling a variable that isn't a function.
- **Security:** By ensuring that a variable only contains the type of data it was intended to hold, type checking prevents certain memory exploits, such as buffer overflows or unauthorized memory access.
- **Efficiency (Code Optimization):** If the compiler knows the exact type of a variable (e.g., a 32-bit integer), it can select the most efficient machine instruction. Without type knowledge, the compiler would have to generate slower, more generic code to handle various possibilities.
- **Memory Management:** Different types require different amounts of memory (e.g., a char vs. a double). Type checking allows the compiler to allocate the exact amount of space needed on the stack or heap.

---

4. Type Conversions

The type checker also manages how types interact:

- **Implicit Conversion (Coercion):** The compiler automatically changes a type (e.g., converting an int to a float during addition).[7]
- **Explicit Conversion (Casting):** The programmer manually forces a type change (e.g., (int)3.14).[8]

Explain the backpatching scheme for Control Flow Statements with suitable examples. Discuss the three dunctions that are used for manipulating list of jumps in the process of translation.

Backpatching is a technique used by compilers to generate code for boolean expressions and control-flow statements in a single pass.

In a standard one-pass compiler, when a jump instruction (like goto) is generated, the target address might not be known yet because the target label hasn't been reached. Backpatching solves this by leaving the jump address blank and "patching" it later once the address is determined.

Three Functions for List Manipulation

To implement backpatching, we use three primary functions to manage lists of instruction addresses:

1. **makelist(i)**: Creates a new list containing only the index i (an address of a jump instruction). It returns a pointer to this list.
2. **merge(p1, p2)**: Concatenates two lists pointed to by p1 and p2. It returns a pointer to the combined list. This is useful when multiple conditions lead to the same destination.
3. **backpatch(p, i)**: This is the core function. It takes a list p and fills in the address i as the target for every instruction index stored in the list.

Backpatching in Control Flow (Example)

Consider the translation of an if-then-else statement:

if (B) S1 else S2

In this scheme, the boolean expression B has two lists:

- **B.truelist**: Instructions that jump if B is true.
- **B.falselist**: Instructions that jump if B is false.

Example Walkthrough:

Suppose we have the code: if (x < 100) y = 1; else y = 2;

1. **Generate Jump for B:** The compiler generates:
    - 100: if x < 100 goto _
    - 101: goto _
2. **List Creation:** * B.truelist = makelist(100)
    - B.falselist = makelist(101)
3. **Handle S1 (True Path):** When the compiler reaches y = 1, it knows this starts at address 102. It calls **backpatch(B.truelist, 102)**.
4. **Handle S2 (False Path):** After S1, there is usually a jump to skip the else block. Suppose y = 2 starts at address 104. The compiler calls **backpatch(B.falselist, 104)**.

Importance of Backpatching

- **Single-Pass Efficiency:** It avoids the need to build a complete Abstract Syntax Tree (AST) before generating code, saving memory and time.
- **Jump Optimization:** It cleanly handles complex nested logic (like if(A || B)) by merging lists of jumps that all lead to the same "true" or "false" block.