5. With the help of an example, explain the different representation of three address code. Generate 3 address code for the Boolean expression — if A<B OR C<D AND P<S.

In compiler design, **Three-Address Code (TAC)** is an intermediate representation used in compilers. It is called "three-address" because each instruction typically involves at most three addresses (or operands), which could be variables, constants, or labels.

There are different representations of Three-Address Code, which can vary based on the structure of the instructions. Some common forms include:

1. **Quadruples (4-tuple)**:
   - A tuple consisting of an operator, two operands, and a result (e.g., `(operator, operand1, operand2, result)`).
   - Example: `(+, a, b, t1)` means `t1 = a + b`.
2. **Triples (3-tuple)**:
   - A tuple consisting of an operator and two operands, but without a separate result field (e.g., `(operator, operand1, operand2)`).
   - Example: `(+, a, b)` means `a + b`.
3. **Indirect Triples**:
   - A variation of triples where the result is stored indirectly using a reference to the instruction rather than a direct variable.
   - Example: `t1 = a + b` and storing `t1` indirectly using a reference.

---

## Example: Boolean Expression TAC

Given the Boolean expression:

```
A < B OR C < D AND P < S
```

We need to generate the Three-Address Code for this expression. We'll break it down into smaller steps and generate the corresponding TAC.

## Step-by-Step Breakdown:

1. **Evaluate `A < B`**:
   - `t1 = A < B`
2. **Evaluate `C < D`**:
   - `t2 = C < D`
3. **Evaluate `P < S`**:
   - `t3 = P < S`
4. **Evaluate `t2 AND t3` (i.e., `C < D AND P < S`)**:
   - `t4 = t2 AND t3`
5. **Evaluate `t1 OR t4` (i.e., `A < B OR (C < D AND P < S)`)**:
   - `t5 = t1 OR t4`

---

## Final Three-Address Code:

```
t1 = A < B         // Compare A with B
t2 = C < D         // Compare C with D
t3 = P < S         // Compare P with S
t4 = t2 AND t3     // AND operation between t2 and t3
t5 = t1 OR t4      // OR operation between t1 and t4
```

## Quadruple Representation:

```
( <, A, B, t1 )
( <, C, D, t2 )
( <, P, S, t3 )
( AND, t2, t3, t4 )
( OR, t1, t4, t5 )
```

## Triple Representation:

```
( <, A, B )
( <, C, D )
( <, P, S )
( AND, 1, 2 )
( OR, 0, 3 )
```

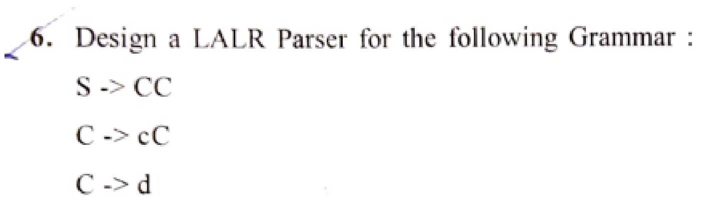
## Indirect Triple Representation:

- Each instruction stores the result as a reference to the previous instruction.

This is how the Boolean expression can be represented in Three-Address Code using different formats.

---

---



6. Design a LALR Parser for the following Grammar :

   S -> CC

   C -> cC

   C -> d

## LALR Parser Design for the Grammar:

Given the grammar:

1. **S → CC**
2. **C → cC**
3. **C → d**

## Step 1: Construct the Canonical Collection of LR(1) Items

To construct the LALR parser, we first need to create the **LR(1) items** from the given grammar. We do this by creating **item sets** that contain the possible parsing configurations of the grammar.

**Starting with the augmented grammar:**

Let the augmented grammar be:

1. **S' → S**
2. **S → CC**
3. **C → cC**
4. **C → d**

Now, we will create the **LR(1) items** for each production.

**Initial Item Set ($I_0$)**

We begin with the start symbol **S'** and place a dot at the beginning of the production rule:

- **$I_0$**:

```
S' → •S
S → •CC
C → •cC
C → •d
```

**Step 2: Create Item Sets by Moving the Dot**

We now move the dot through the items to generate the next item sets.

- **Move the dot after S (from $I_0$)**:

```
I₁:
S' → S•
```

- **Move the dot after C (from $I_0$)**:

```
I₂:
S → C•C
```

- **Move the dot after c (from C → cC)**:

```
I₃:
C → c•C
```

- **Move the dot after d (from C → d)**:

```
I₄:
C → d•
```

**Step 3: Construct the Remaining Item Sets**

We continue by moving the dot based on the non-terminal and terminal symbols. Here are the remaining items:

- **Move the dot after C (from S → CC)**:

  ```
  I₅:
  S → C C•
  ```

- **Move the dot after C (from C → cC)**:

  ```
  I₆:
  C → c C•
  ```

- **Move the dot after C (from C → cC)**:

  ```
  I₇:
  C → •cC
  ```

**Step 4: Construct the LR(1) Parsing Table**

Now that we have all the item sets, we can create the **LR(1) parsing table** with **actions** (shift/reduce) and **goto** entries.

- **Actions**:
  - **Shift**: Move to the next state.
  - **Reduce**: Apply a production rule to reduce the stack.
  - **Accept**: Accept the input string.
- **Goto**: This tells us where to go after processing a non-terminal symbol.

For simplicity, we will not build the full table here, but I will provide an outline of what the **action** and **goto** tables will look like:

| State | Action (on terminal input) | Goto (on non-terminal input) |
|---|---|---|
| 0 | Shift on `c` → 3, Shift on `d` → 4 | Goto on `S` → 1, Goto on `C` → 2 |
| 1 | Accept | |
| 2 | Shift on `c` → 3, Shift on `d` → 4 | Goto on `C` → 5 |
| 3 | Shift on `c` → 6, Shift on `d` → 4 | |
| 4 | Reduce C → d | |
| 5 | Reduce S → CC | |
| 6 | Reduce C → cC | |

## Step 5: Final LALR Parser

The final **LALR Parser** uses the above parsing table to parse the input string. The steps are:

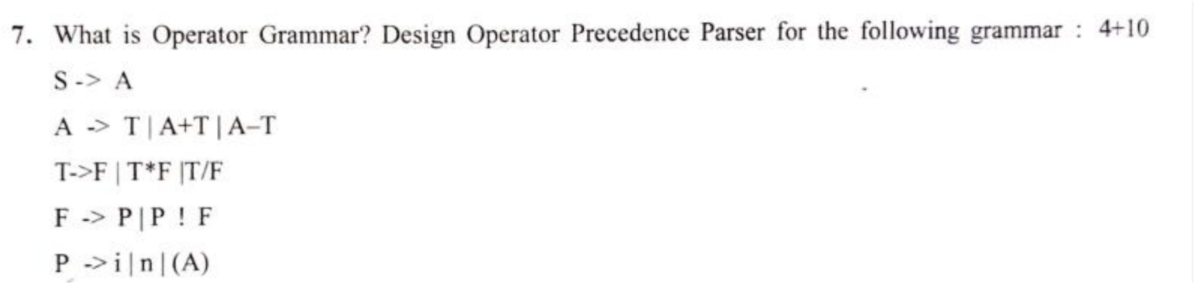
1. Start at state 0.
2. For each input symbol, check the **action** and either **shift** to the next state or **reduce** according to the rules.

3. Use the **goto** to move between states when parsing non-terminals.
4. Accept the input if the parse is successful, or handle errors if the action is not defined.

This completes the design of an **LALR parser** for the given grammar. The parsing table and states are simplified here, but in an actual implementation, these would be generated automatically by an LALR parser generator like **Yacc** or **Bison**.

---

---



7. What is Operator Grammar? Design Operator Precedence Parser for the following grammar : 4+10

S -> A

A -> T | A+T | A–T

T->F | T*F |T/F

F -> P | P ! F

P -> i | n | (A)

## Operator Grammar:

**Operator Grammar** is a type of context-free grammar where the productions are designed specifically to handle operators and precedence relationships between them. This kind of grammar is used to define the structure of expressions involving operators such as `+`, `-`, `*`, `/`, etc. The primary purpose of an operator grammar is to describe the precedence and associativity rules of operators in expressions.

In **Operator Precedence Parsing**, the grammar is defined such that each operator has a certain precedence (order of evaluation) and associativity (left or right). The parser will use this information to correctly order the operations when parsing expressions.

## Grammar Given:

- **S → A**
- **A → T | A + T | A - T**
- **T → F | T * F | T / F**
- **F → P | P ! F**
- **P → i | n | (A)**

### Step 1: Assign Operator Precedence and Associativity

The grammar involves several operators, such as `+`, `-`, `*`, `/`, and `!`, each of which needs to be assigned a precedence.

We assume the following operator precedence:

- `!` has the highest precedence (since it is a unary operator).
- `*` and `/` have the next level of precedence (higher than `+` and `-`).
- `+` and `-` have the lowest precedence.

Furthermore, we assume the following **associativity**:

- **Unary operators** like `!` are **right-associative**.
- **Binary operators** like `+`, `-`, `*`, and `/` are **left-associative**.

## Step 2: Define the Operator Precedence Relations

The precedence relations can be defined as:

- `!` > `*`, `/`, `+`, `-` (highest precedence).
- `*`, `/` > `+`, `-` (multiplication and division have higher precedence than addition and subtraction).
- `+`, `-` have equal precedence (lowest).

## Step 3: Construct the Operator Precedence Parser

To construct an **Operator Precedence Parser**, we need to define the following:

1. **Operator Precedence Table**: This table specifies the relative precedence between operators.
2. **Parsing Stack**: This stack keeps track of operators and operands while parsing the input expression.
3. **Action Table**: Defines whether we need to **shift** (push the current symbol onto the stack) or **reduce** (apply a production rule) based on operator precedence.

## Operator Precedence Table:

We define the precedence relations in the following table:

| Operator | + | - | * | / | ! | i | n | ( | ) |
|---|---|---|---|---|---|---|---|---|---|
| + | = | = | < | < | < | < | < | < | > |
| - | = | = | < | < | < | < | < | < | > |
| * | > | > | = | = | < | < | < | < | > |
| / | > | > | = | = | < | < | < | < | > |
| ! | > | > | > | > | = | < | < | < | > |
| i | > | > | > | > | > | = | = | < | > |
| n | > | > | > | > | > | = | = | < | > |
| ( | < | < | < | < | < | < | < | < | = |
| ) | > | > | > | > | > | > | > | | |

- The `=` symbol represents equality: Operators of equal precedence can be processed in the same way.
- The `<` and `>` symbols represent precedence relations: `<` means the left operator has lower precedence, and `>` means the right operator has lower precedence.

## Step 4: Algorithm for the Operator Precedence Parser

The algorithm for the Operator Precedence Parser is as follows:

1. **Initialize** the stack with an initial symbol (usually `$` for the end of input).
2. **Move through the input** expression, comparing the top of the stack with the current symbol using the precedence table.
   - If the precedence relation is `>`, **reduce** (apply a production rule).
   - If the precedence relation is `<`, **shift** (push the current symbol onto the stack).
   - If the precedence relation is `=`, **accept** or **reduce** based on the rule.
3. **Repeat** until the entire input expression is parsed.

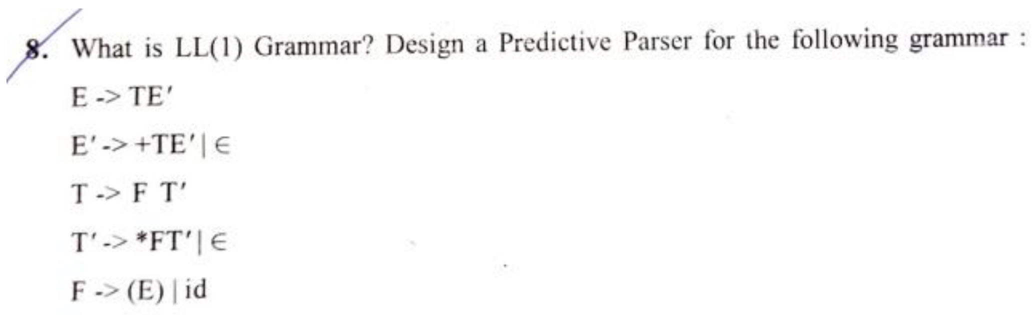
## Example of Parsing the Expression: `i + n * i`

Let's parse the expression `i + n * i` using the operator precedence parsing algorithm.

1. **Initial Stack**: `$`
   - Input: `i + n * i`
   - Shift `i` to the stack.
2. **Stack**: `i`
   - Input: `+ n * i`
   - Shift `+` to the stack.
3. **Stack**: `i +`
   - Input: `n * i`
   - Since `+` has lower precedence than `*`, shift `n` to the stack.
4. **Stack**: `i + n`
   - Input: `* i`
   - Shift `*` to the stack.
5. **Stack**: `i + n *`
   - Input: `i`
   - Shift `i` to the stack.
6. **Stack**: `i + n * i`
   - Input: (empty)
   - Reduce by `T → F` for the last `i`.
7. **Stack**: `i + n * F`
   - Input: (empty)
   - Reduce by `T → T * F` for `n * i`.
8. **Stack**: `i + T`
   - Input: (empty)
   - Reduce by `A → A + T` for `i + n * i`.
9. **Stack**: `A`
   - Input: (empty)
   - Accept the input.

## Conclusion:

An **Operator Precedence Parser** is designed by defining the precedence and associativity rules of operators in the grammar, constructing an operator precedence table, and then applying the shift/reduce algorithm to parse the input expression. This parser is particularly useful for mathematical expressions and can handle complex operator precedence and associativity rules.



8. What is LL(1) Grammar? Design a Predictive Parser for the following grammar :

E -> TE′

E′ -> +TE′ | ∈

T -> F T′

T′ -> *FT′ | ∈

F -> (E) | id

## LL(1) Grammar:

**LL(1) Grammar** is a type of context-free grammar that can be parsed using a **predictive parser** with **one lookahead symbol**. "LL" stands for:

- **L**: Left-to-right scanning of the input string.
- **L**: Leftmost derivation of the string.
- **1**: Using 1 lookahead symbol to decide which production to use.

For a grammar to be LL(1), it must satisfy the following conditions:

1. **No Left Recursion**: The grammar must not have any left recursion (i.e., a production where a non-terminal on the left-hand side can eventually lead back to itself).
2. **Uniqueness of First Sets**: The FIRST sets of the right-hand sides of each production for a non-terminal must be disjoint. If two productions for the same non-terminal have a common terminal in their FIRST sets, the grammar is not LL(1).
3. **Nullability Handling**: If one of the productions for a non-terminal can derive the empty string (ε), the FIRST set of the non-terminal must not overlap with the FOLLOW set of the non-terminal.

## Grammar Given:

```
E  → TE'
E' → +TE' | ε
T  → FT'
T' → *FT' | ε
F  → (E) | id
```

## Step 1: Compute FIRST and FOLLOW Sets

We first compute the **FIRST** and **FOLLOW** sets for the grammar.

**FIRST Sets:**

- **FIRST(E) = FIRST(T) = FIRST(F) = { '(', 'id' }**
- **FIRST(E') = { '+', ε }** (since `E' → +TE'` and `E' → ε`)
- **FIRST(T') = { '*', ε }** (since `T' → *FT'` and `T' → ε`)
- **FIRST(F) = { '(', 'id' }**

**FOLLOW Sets:**

- **FOLLOW(E) = { '$', ')' }** (since `E` is the start symbol and it is followed by `)` in `F → (E)` and `$` as the end of input).
- **FOLLOW(E') = { '$', ')' }** (since `E'` appears at the end of the production `E → TE'` and is followed by `)` in `F → (E)`).
- **FOLLOW(T) = { '+', '$', ')' }** (since `T` is followed by `+` in `E' → +TE'` and `)` in `F → (E)`).
- **FOLLOW(T') = { '+', '$', ')' }** (since `T'` is followed by `+` in `E' → +TE'` and `)` in `F → (E)`).
- **FOLLOW(F) = { '+', '*', '$', ')' }** (since `F` is followed by `*` in `T' → *FT'`, and `+`, `*`, and `)` in other productions).

## Step 2: Construct Predictive Parsing Table

The predictive parsing table is a table that tells us which production to apply based on the current non-terminal and the lookahead symbol.

| Non-terminal | **id** | **(** | **+** | ***** | **)** | **$** |
|---|---|---|---|---|---|---|
| **E** | `T E'` | `T E'` | | | | |
| **E'** | | | `+ T E'` | | `ε` | `ε` |
| **T** | `F T'` | `F T'` | | | | |
| **T'** | | | `ε` | `* F T'` | `ε` | `ε` |
| **F** | `id` | `( E )` | | | | |

**Explanation of the Table:**

- **For `E`**:
  - When the lookahead symbol is `id` or `(`, we apply the production `E → T E'`.
- **For `E'`**:
  - When the lookahead symbol is `+`, we apply the production `E' → + T E'`.
  - When the lookahead symbol is `)` or `$` (end of input), we apply the production `E' → ε` (empty string).
- **For `T`**:
  - When the lookahead symbol is `id` or `(`, we apply the production `T → F T'`.
- **For `T'`**:
  - When the lookahead symbol is `*`, we apply the production `T' → * F T'`.
  - When the lookahead symbol is `+`, `)`, or `$`, we apply `T' → ε`.
- **For `F`**:
  - When the lookahead symbol is `id`, we apply the production `F → id`.
  - When the lookahead symbol is `(`, we apply the production `F → ( E )`.

## Step 3: Predictive Parsing Algorithm

The **Predictive Parsing** algorithm works as follows:

1. **Initialize**:
   - Place the start symbol `E` on the stack.
   - The input string is given, and a pointer is set at the beginning of the string.

2. **Repeat**:
   - If the top of the stack is a terminal symbol, compare it with the current input symbol. If they match, pop the stack and move to the next input symbol. Otherwise, report a syntax error.
   - If the top of the stack is a non-terminal symbol, use the predictive parsing table to select the appropriate production rule based on the current input symbol (lookahead).
   - Replace the non-terminal symbol with the right-hand side of the production rule and continue parsing.
3. **Accept**:
   - If the stack is empty and all input symbols are consumed, the input string is successfully parsed.

## Example Parsing: `id + id * id`

Input: `id + id * id`

1. **Initialize**:
   - Stack: `E $`
   - Input: `id + id * id $`
2. **Apply `E → T E'`**:
   - Stack: `T E' $`
   - Input: `id + id * id $`
3. **Apply `T → F T'`**:
   - Stack: `F T' E' $`
   - Input: `id + id * id $`
4. **Apply `F → id`**:
   - Stack: `id T' E' $`
   - Input: `id + id * id $`
5. **Match `id` with input symbol**:
   - Stack: `T' E' $`
   - Input: `+ id * id $`
6. **Apply `T' → ε`**:
   - Stack: `E' $`
   - Input: `+ id * id $`
7. **Apply `E' → + T E'`**:
   - Stack: `+ T E' $`
   - Input: `+ id * id $`
8. **Match + with input symbol**:
   - Stack: `T E' $`
   - Input: `id * id $`
9. **Apply `T → F T'`**:
   - Stack: `F T' E' $`
   - Input: `id * id $`
10. **Apply `F → id`**:
    - Stack: `id T' E' $`
    - Input: `* id $`
11. **Match `id` with input symbol**:
    - Stack: `T' E' $`

   - Input: `* id $`
12. **Apply `T' → * F T'`**:
    - Stack: `* F T' E' $`
    - Input: `* id $`
13. **Match `*` with input symbol**:
    - Stack: `F T' E' $`
    - Input: `id $`
14. **Apply `F → id`**:
    - Stack: `id T' E' $`
    - Input: `id $`
15. **Match `id` with input symbol**:
    - Stack: `T' E' $`
    - Input: `$`
16. **Apply `T' → ε`**:
    - Stack: `E' $`
    - Input: `$`
17. **Apply `E' → ε`**:
    - Stack: `$`
    - Input: `$`
18. **Accept**: The input string is successfully parsed.

## Conclusion:

This is a **Predictive Parser** for the given LL(1) grammar. It uses the **FIRST** and **FOLLOW** sets to construct a **parsing table** and applies a **top-down parsing** strategy to parse the input string.