YEAR 2022

(a) Viable prefix
A viable prefix is a prefix of a right sentential form that does not extend beyond the right end of the handle.
Example:
For grammar S → aAB, A → a, B → b,
In the sentential form aaBb, aaB is a viable prefix.

(**b) CFG for palindromes**
A palindrome reads the same forward and backward.

S → aSa | bSb | a | b | ε
(c) Grammar for unequal number of 0s and 1s

S → A0 | A1
A → 0A1 | 1A0 | ε
This ensures number of 0s ≠ number of 1s.

(d) Unit production
A unit production is a production of the form A → B where A and B are non-terminals.
Example:

A → B
B → a
Impact: Unit productions can be removed by replacing them with equivalent non-unit productions to simplify the grammar.

(e) Language generated by R = (aa)*(bb)*b

(aa)* = even number of a's (including 0)

(bb)* = even number of b's (including 0)

Final b makes total b's odd.
Language: { a^(2m) b^(2n+1) | m, n ≥ 0 }

(f) Conditions for rejecting an input string in a Turing Machine
A Turing machine rejects if:

It enters a reject state.

It halts without entering an accept state.

It loops indefinitely without accepting.

(g) **Handling unary operators in operator precedence relations**
Unary operators can be handled by:

Assigning highest precedence to unary operators.

Using separate precedence rules for unary and binary forms.

Modifying the grammar to distinguish unary from binary usage.

a) **Eliminate left recursion (direct and indirect) from the following grammer :**
S->L,S | S
L->Lb|Sa|c

Thanks — one quick thing: the original S -> L,S | S you wrote looks like it has a typo (the , and the bare S are ambiguous).
A very common textbook example is:

S → L S | ε
L → L b | S a | c

I'll solve for that version (if your grammar is different paste the exact productions and I'll redo it).

Goal: eliminate direct and indirect left recursion.

Step 1 — list productions

less
Copy
Edit
S → L S | ε
L → L b | S a | c
Step 2 — remove indirect recursion by substitution
Process nonterminals in order (S first, then L).
When processing L, it has a production starting with S (S a), so replace that S by its productions L S and ε:

L → L b | (L S) a | ε a | c
ε a becomes a, so:

less
Copy
Edit
L → L b | L S a | a | c
Step 3 — eliminate immediate left recursion in L
We have forms L → L α | β with

left-recursive alternatives: L b (α_1 = b) and L S a (α_2 = S a)

non-left alternatives: a, c (β_1, β_2)

Introduce new nonterminal L' and rewrite:

nginx
Copy

Edit
L → a L' | c L'
L' → b L' | S a L' | ε
S had no immediate left recursion, so S stays:

nginx
Copy
Edit
S → L S | ε
Final grammar (left-recursion eliminated):

nginx
Copy
Edit
S → L S | ε
L → a L' | c L'
L' → b L' | S a L' | ε
Remarks:

This grammar contains no immediate left recursion. The indirect recursion between S and L was handled by substituting S's productions into L before eliminating L's immediate left recursion.

If your original S production was different (for example S → L S | S or S → L,S | S literally), paste it exactly and I'll redo the elimination for that exact grammar.

Alright — let's remove direct and indirect left recursion for your exact grammar:

Given:

less
Copy
Edit
S → L S | S
L → L b | S a | c
Step 1 – Detect the recursion type
Indirect left recursion is possible between S and L because:

S → L S and L → S a form a cycle (S ⇒ L S ⇒ S a S).

L → L b has direct left recursion.

We need to eliminate indirect first, then direct.

Step 2 – Order non-terminals
Order: S, L.

We substitute productions where earlier non-terminals appear.

Step 3 – Remove indirect recursion from S
S → L S | S

But L → L b | S a | c starts with:

L (left recursion)

S (indirect)

c (terminal)

Substitute L in S → L S:

r
Copy
Edit
S → (L b | S a | c) S | S
So:

less
Copy
Edit
S → L b S | S a S | c S | S
Now S appears on RHS starting with itself in S a S and with direct S — that's direct left recursion.

Step 4 – Remove direct left recursion from S
Write S in the form:

nginx
Copy
Edit
S → S α | β
From above:

α's (starting with S): a S, ε (empty) if S alone.

β's (not starting with S): L b S, c S.

Thus:

less
Copy
Edit
S → L b S | c S | S a S | S
Direct left recursion removal:

nginx
Copy
Edit
S → L b S S' | c S S'
S' → a S S' | ε
Step 5 – Remove direct left recursion from L
L → L b | S a | c
Here:

Left recursive part: L b

Non-left recursive: S a, c

Eliminate:

nginx
Copy
Edit
L → S a L' | c L'
L' → b L' | ε
Final grammar without left recursion
nginx
Copy
Edit
S → L b S S' | c S S'
S' → a S S' | ε

L → S a L' | c L'
L' → b L' | ε
If you want, I can also fully expand S so it has no mutual recursion with L (for top-down parsing without left factoring). That would make it even cleaner.

(b) **Input buffering in lexical analysis — concept and role of sentinels**
Concept: Input buffering uses buffered blocks (typically two equal-size buffers) to read the source into memory in chunks so the scanner can read characters efficiently without a system call per character. The lexeme begin and forward pointers move through the buffers to identify tokens.
Two-buffer scheme: Fill buffer A and B alternately. When forward reaches the end of one buffer, the next buffer is refilled while scanning continues from the other buffer.
Sentinels help by: placing a special end-of-buffer marker (sentinel) at the end of each buffer so the scanner can detect buffer boundary without checking forward == end on every character. When a sentinel is found the scanner refills the next buffer and continues, avoiding expensive boundary checks for every character. Sentinels simplify code and slightly improve speed.

(c) **Top-down parsing without backtracking — when possible and example**
When possible: You can build a top-down parser with no backtracking (a predictive parser) iff the grammar is LL(1) (or LL(k) for some fixed k): for each nonterminal and lookahead token the parser can choose exactly one production. That requires the grammar to be free of left recursion and to be left-factored so FIRST/FOLLOW sets do not conflict.
Parser type: Predictive parser (table-driven LL(1)).
Example: Grammar for simple expressions (no left recursion, left-factored):

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id
This grammar is LL(1) so predictive parsing works without backtracking: at each step the parser inspects one lookahead token and picks a unique production.

(d) **DAG for a + a*(b - c) + (b - c)*d — identify common subexpressions**
Observe common subexpression: (b - c) occurs twice. Build DAG nodes for unique subexpressions and reuse (b-c) node.

Textual DAG structure (nodes and edges):

Node N1: a

Node N2: b

Node N3: c

Node N4: -(N2, N3) representing (b - c) ← reused

Node N5: *(N1, N4) representing a * (b - c)

Node N6: *(N4, d) representing (b - c) * d

Node N7: +(N1, N5) representing a + a*(b-c)

Node N8 (root): +(N7, N6) representing a + a*(b-c) + (b-c)*d

Comment: By using the DAG node N4 for both occurrences of (b-c) we avoid recomputation and expose optimization opportunities (common-subexpression elimination).

(e) **(i) 3-address code for x = *y; a = &x;**
We assume * is dereference and & is address-of. 3-address form uses temporaries for intermediate results.

(i) Three-address code (one possible form):

ini
Copy
Edit
t1 = * y // load value pointed by y
x = t1
a = & x // address of x
(you might simply write x = *y and a = &x as two 3-address statements; above shows an explicit temp)

(ii) Triples (each triple is (op, arg1, arg2) and indexed by result position):

scss
Copy
Edit
(0) (*, y, -) // result at [0] is *y
(1) (=, 0, x) // copy result[0] into x (or (=, [0], x))
(2) (&, x, -) // result at [2] is &x
(3) (=, 2, a) // copy result[2] into a
Indirect triples use pointers to triples instead of explicit result positions:

scss

Copy
Edit
(0) (*, y, -)
(1) (=, (0), x)
(2) (&, x, -)
(3) (=, (2), a)
(Here (0) means "use result of triple 0". Implementation details vary; the above shows the mapping.)

(f) **“There are some CFGs for which shift–reduce parsing cannot be used” — critical comment**
Short answer: Shift–reduce (LR) parsing methods can handle a very large class of grammars (LR(0), SLR(1), LALR(1), LR(1)), including many grammars that LL parsers can’t. However, not all CFGs are LR(k) for any fixed k: some grammars are inherently ambiguous or require unbounded lookahead/complex disambiguation, so deterministic shift–reduce parsers cannot parse them. Examples:

Ambiguous grammar: S → S S | a is ambiguous (multiple parse trees) and cannot be parsed by any deterministic LR(k) parser in a straightforward deterministic way.

Non-LR grammar: There exist unambiguous CFGs that are still not LR(k) for any fixed k (they require nonlocal context).

Conclusion: Shift-reduce algorithms are powerful and practical for a wide range of programming-language grammars (most language grammars are LALR(1) or LR(1) after small rewrites). But there exist CFGs (especially ambiguous or those requiring unbounded lookahead) for which shift–reduce parsing cannot be applied deterministically. In such cases one must use generalized parsers (GLR, Earley) or rewrite the grammar.

3. Convert the regular expression abb(a | b)* to DFA using the direct method and minimize it.

1) Regular expression (augmented)

Regex:

R=abb(a | b) ∗

Augment with end marker # for the direct method:

R′=(abb(a | b) ∗)#.

We number the occurrences (leaf positions):

1. a (pos 1)
2. b (pos 2)
3. b (pos 3)
4. a (pos 4) inside the (a|b)
5. b (pos 5) inside the (a|b)
6. # (pos 6) — end marker

---

2) Syntax-tree attributes (nullable, firstpos, lastpos)

(Compute for internal nodes; final `root.firstpos = {1}` and `root.lastpos = {6}`.)

Important computed values (you can show a small table in exam):

- Leaves: firstpos(i) = lastpos(i) = {i} for i=1..6. none are nullable.
- `or` node for (4|5): firstpos = {4,5}, lastpos = {4,5}, nullable = false
- star node `(4|5)*`: firstpos = {4,5}, lastpos = {4,5}, nullable = **true**
- concat of `a·b` then · `b` then · star then · # yields:
    - root.firstpos = {1}
    - some intermediate lastpos include {3} then {3,4,5} and finally {6}

(You do not need every intermediate value in the answer if space is limited — show the principle and key firstpos/lastpos.)

---

3) followpos table (key step of direct method)

Using rules:

- For every concat node `X = A · B`, for every `i ∈` lastpos(A) add firstpos(B) to followpos(i).
- For every star node `A*`, for every `i ∈` lastpos(A) add firstpos(A) to followpos(i).

Computed **followpos** (only non-empty entries shown):

```
cpp
CopyEdit
followpos(1) = {2}
followpos(2) = {3}
followpos(3) = {4,5,6}     // after the third b comes the (a|b)* or #
followpos(4) = {4,5,6}     // because 4,5 are inside the star -> can loop to
firstpos{4,5} and eventually #
followpos(5) = {4,5,6}
followpos(6) = ∅
```

(These are the crucial results used to form DFA transitions.)

---

4) Build DFA (states = sets of positions)

Start state = `firstpos(root) = {1}`.

Compute transitions on input symbols `a` and `b`:

- Let `posSym` = symbol at each position:

- 1 → a, 2 → b, 3 → b, 4 → a, 5 → b, 6 → #.

For state `S` and input `x`:
`move(S,x) = ∪ followpos(p)` for every position `p ∈ S` whose symbol = `x`.

Compute reachable states:

1. `A = {1}` (start)
    - on `a`: positions in A with symbol `a` = {1} → move = followpos(1) = {2}
    - on `b`: none → move = ∅
2. `B = {2}`
    - on `a`: none → ∅
    - on `b`: {3}
3. `C = {3}`
    - on `a`: none → ∅
    - on `b`: followpos(3) = {4,5,6}
4. `D = {4,5,6}` (contains position 6 = # ⇒ **accepting**)
    - on `a`: for pos4 (a) → followpos(4) = {4,5,6}; pos5,6 don't contribute `a` moves beyond these sets ⇒ D on `a` → D
    - on `b`: similarly D on `b` → D
5. Dead state `E = ∅`
    - on `a` or `b` → E

So the DFA states (rename for convenience):
A={1} (start), B={2}, C={3}, D={4,5,6} (accept), E=∅ (dead).

Transition table (final DFA)

```
mathematica
CopyEdit
State |  a   |  b   | Accept?
------+------|------|--------
  A   |  B   |  E   |   No
  B   |  E   |  C   |   No
  C   |  E   |  D   |   No
  D   |  D   |  D   |  Yes
  E   |  E   |  E   |   No
```

Graphically (compact):

- A --a--> B
- A --b--> dead
- B --b--> C

- C --b--> D
- B --a--> dead ; C --a--> dead
- D loops on a and b to itself
- dead loops to itself

Interpretation: to accept, input must start with `a b b` (A → B → C → D after `abb`) and once in D any further a/b stay in D (because `(a|b)*` allows any suffix). Dead state catches strings that do not match the required prefix.

---

5) Minimize DFA

Partition initial groups: Accepting `{D}` and Non-accepting `{A,B,C,E}`.

Refinement by transitions (standard table/refinement) yields the partitions:

- `{D}` (accepting sink)
- `{A}` (because A on `a` → B but others behave differently)
- `{B}`
- `{C}`
- `{E}` (dead)

All states are distinguishable; **no further merging possible**. So the DFA above is already minimal with **5 states**.

---

6) Final answer (concise exam write-up)

1. Show the syntax tree (annotate leaves 1..6) and write `followpos` table (as above). (≈ 3 marks)
2. Give start state `firstpos(root) = {1}` and construct DFA states by computing moves using `followpos` (show steps for A → B → C → D). (≈ 3 marks)
3. Present the DFA transition table and mark accepting state `D = {4,5,6}`. (≈ 2 marks)
4. Minimize with partition-refinement (show initial partition and refinement — result: 5 distinct partitions A,B,C,D,E) and conclude DFA is minimal. (≈ 2 marks)

---

7) Short note (intuition)

This DFA accepts exactly strings that begin with the prefix `abb` followed by **any** string of `a` and `b` (including empty). That matches the regex `abb(a|b)*`. The sink `E` collects prefixes that can't lead to the required `abb` prefix.

4. Construct a Turing machine for the language $\{a^n b^n c^n\}$ and show that a string in that language can be derived from this machine. Write down the intermediate steps. 5+5

1. Idea / informal algorithm (brief)
    1. Repeatedly find the leftmost unmarked `a`, mark it (change to `X`).
    2. Go right to find the leftmost unmarked `b`, mark it (change to `Y`).
    3. Continue right to find the leftmost unmarked `c`, mark it (change to `Z`).
    4. Return to the leftmost tape position and repeat until there are no unmarked `a`'s.
    5. Then verify there are no remaining unmarked `b` or `c`. If all symbols are `X, Y, Z` (and blanks), **accept**; otherwise **reject**.

(This approach ensures one `a`, one `b`, one `c` are consumed per cycle, so counts must match.)

---

2. Formal TM description (states, tape alphabet, transitions)

Tape alphabet: {a,b,c,X,Y,Z,␣} where ␣ is blank.

States:

- `q0` — start, search for leftmost unmarked `a`.
- `q1` — after marking `a` (to `X`), scan right to find `b`.
- `q2` — after marking `b` (to `Y`), scan right to find `c`.
- `q3` — after marking `c` (to `Z`), move left back to left end (to next `a`).
- `q4` — final check: verify no unmarked symbols remain; accept if OK.
- `q_accept`, `q_reject`.

Transition function (important/representative rules). Write δ(state, symbol) = (new-state, write, move).

```
pgsql
CopyEdit
(1) Start / find a
δ(q0, a) = (q1, X, R)       // mark leftmost a -> find b
δ(q0, X) = (q0, X, R)       // skip marked X
δ(q0, Y) = (q0, Y, R)
δ(q0, Z) = (q0, Z, R)
```

```
δ(q0, b) = (q_reject, b, S) // found b before any a => bad
δ(q0, c) = (q_reject, c, S)
δ(q0, ␣) = (q4, ␣, R)       // no more a: go to final check

(2) Find b after marking a
δ(q1, a) = (q1, a, R)       // skip remaining a's (shouldn't see unmarked a
before b ideally)
δ(q1, X) = (q1, X, R)
δ(q1, b) = (q2, Y, R)       // mark b -> find c
δ(q1, Y) = (q1, Y, R)
δ(q1, c) = (q_reject, c, S) // b missing, found c instead => reject
δ(q1, ␣) = (q_reject, ␣, S) // reached blank - missing symbols

(3) Find c after marking b
δ(q2, b) = (q2, b, R)
δ(q2, Y) = (q2, Y, R)
δ(q2, c) = (q3, Z, L)       // mark c then go left to return
δ(q2, ␣) = (q_reject, ␣, S) // no c found => reject

(4) Return left to next a
δ(q3, a) = (q3, a, L)
δ(q3, b) = (q3, b, L)
δ(q3, c) = (q3, c, L)
δ(q3, X) = (q3, X, L)
δ(q3, Y) = (q3, Y, L)
δ(q3, Z) = (q3, Z, L)
δ(q3, ␣) = (q0, ␣, R)       // at left blank -> start next cycle

(5) Final check (no unmarked a/b/c)
δ(q4, X) = (q4, X, R)
δ(q4, Y) = (q4, Y, R)
δ(q4, Z) = (q4, Z, R)
δ(q4, ␣) = (q_accept, ␣, S)  // all symbols marked -> accept
δ(q4, a) = (q_reject, a, S)  // leftover unmarked symbol -> reject
δ(q4, b) = (q_reject, b, S)
δ(q4, c) = (q_reject, c, S)
```

(You may compress states or adjust moves, but this set is standard and correct.)

---

3. State diagram (sketch)

q0 (start) → q1 → q2 → q3 → q0 (loop per cycle). q0→q4 if no `a` left. q4→q_accept if only X,Y,Z and blank; otherwise q_reject.

---

4. Worked example (intermediate steps) — input `aabbcc` (show 5 marks)

We show configurations. Use convention: tape contents with the current state written just before the symbol under the head. Initially head at leftmost symbol. Blank symbol ␣ shown as _ for clarity.

Initial tape (with blanks on both ends):
`_ a a b b c c _ _ ...` and head at the first `a`. Start in `q0`.

List configurations (compressing blanks at ends):

1. `q0 a a b b c c`
   δ(q0, a) → mark `a`→`X`, move R, go to q1:
2. `X q1 a b b c c` (after marking first `a`)

Scan to b and mark it:
3. `X a q1 b b c c` (skip second `a`)
4. `X a Y q2 b c c` (found `b`, mark → `Y`, go to q2)

Scan to c and mark it:
5. `X a Y b q2 c c` (skip b)
6. `X a Y b Z q3 c` (mark first c as `Z`, go left (q3))

Return left to left end:
7. `X a Y b q3 Z c`
8. `X a Y q3 b Z c`
9. `X a q3 Y b Z c`
10. `X q3 a Y b Z c`
11. `q3 X a Y b Z c`
12. Reached left blank: `_ q0 X a Y b Z c` → change to q0 and move right

Start next cycle: find leftmost unmarked `a` (the second `a`):
13. `X q0 a Y b Z c`
δ(q0, a) → mark it: `X X q1 Y b Z c`

Find its b and c:
14. `X X Y q2 b Z c` (mark second `b` to `Y`)
15. `X X Y b q2 Z c`
16. `X X Y b Z q3 c` (mark second `c` to `Z` and go left)

Return to left blank again:
17. `... _ q0 X X Y Y Z Z ...`

Now in q0 the next symbol is `X` then `X`... scanning:
18. `q0 X X Y Y Z Z` → δ(q0,X)=q0 move right until blank
19. After skipping all marked symbols we reach blank: `_ q4 X X Y Y Z Z`
(q0 sees blank → q4 final check)

Final check:
20. q4 scans symbols: sees X,Y,Z only → continues to blank
21. At blank δ(q4, blank) → `q_accept`.

Thus `aabbcc` is accepted. All intermediate steps (marks) shown.

---

5. Rejecting examples / notes

- Input `aabcc` or `aaabbb` will be rejected because at some pass `q1` or `q2` will fail to find the required symbol.
- TM runs in O(n2) time essentially (each cycle scans across tape).

5. (a) Write the algorithm for shift-reduce parsing.

(b) Consider the following grammar :

$$S \rightarrow aABe$$
$$A \rightarrow Abc|b$$
$$B \rightarrow d$$

Using shift-reduce parser algorithm, parse the input string *abbcde*.

(c) Explain in detail the different conflicts that arise in shift reduce parsing.

(a) Shift–Reduce parsing algorithm (4 marks)

Idea: Bottom-up parser that attempts to construct a rightmost derivation in reverse by repeatedly **shifting** the next input symbol onto a stack or **reducing** a handle on top of the stack to a nonterminal, until start symbol is obtained.

Data: stack (initially $), input buffer with $ at end, parsing table/grammar.

Actions:

- **Shift**: push next input symbol onto stack and advance input pointer.
- **Reduce**: if top of stack matches right side β of a production `A → β`, pop `|β|` symbols and push `A`.
- **Accept**: if stack is `$S` and input is `$`.
- **Error**: if no action applicable.

Pseudo-code (simplified):

```
perl
CopyEdit
stack = [$]
input = w$
repeat:
  a = next input symbol
  if there exists production A → β such that top(stack) ends with β:
    // reduce (prefer reductions when grammar/table says so)
    pop |β| symbols
    push A
    record action "reduce A→β"
  else if a != $:
    push a
```

```
    advance input
    record action "shift a"
  else if stack = [$,S]:
    accept
  else:
    error (reject)
until accept or error
```

Notes: Practical parsers use parse tables (LR(0)/SLR(1)/LALR(1)/LR(1)) to decide **shift** vs **reduce** deterministically.

(b) Parse the string `abbcde` using shift–reduce (3 marks)

Grammar (as used):

```
less
CopyEdit
S → a A B e
A → A b c | b
B → d
```

Input: `a b b c d e $` (we append `$` as end marker). Show stack (left = bottom), remaining input, and action.

Step	Stack (bottom → top)	Input remaining	Action
0	$	a b b c d e $	start
1	$ a	b b c d e $	**shift** a
2	$ a b	b c d e $	**shift** b
3	$ a A	b c d e $	**reduce** A → b
4	$ a A b	c d e $	**shift** b
5	$ a A b c	d e $	**shift** c
6	$ a A	d e $	**reduce** A → A b c
7	$ a A d	e $	**shift** d
8	$ a A B	e $	**reduce** B → d
9	$ a A B e	$	**shift** e
10	$ S	$	**reduce** S → a A B e
11	$ S	$	**accept**

Conclusion: the string `abbcde` is **accepted**; full rightmost derivation (reverse reductions) matches grammar.

(c) Conflicts in shift–reduce parsing (3 marks)

Two principal conflicts encountered when building parse tables or deciding parser action:

1. Shift–Reduce conflict

- **Definition:** At some parser state with lookahead `a`, the parser table would allow both shifting (push `a`) and reducing by some production `A → β`.

- **Typical cause:** Ambiguous grammar or grammar that requires lookahead > k for LR(k). Classic example: **dangling-else**:

  ```
  mathematica
  CopyEdit
  S → if E then S | if E then S else S | other
  ```

  On seeing `else` as lookahead, parser must decide whether to **reduce** the inner `if` (then pair) or **shift** `else` to attach it to the nearer `if`.

- **Resolution:** Use precedence rules (prefer shift or specific associativity), rewrite grammar (left-factor), or use more powerful parsing algorithm (LR(1)/GLR).

2. Reduce–Reduce conflict

- **Definition:** In a parser state the table would allow two different reductions `A → β` and `B → γ` (both β and γ match top-of-stack) — ambiguous or indistinguishable handles.
- **Typical cause:** Grammar ambiguity or poor ordering of productions. Example (contrived):

  ```
  less
  CopyEdit
  A → a
  B → a
  S → A b | B c
  ```

  If top-of-stack is `a` and lookahead could lead to either reduction depending on later input, the parser has two possible reductions.

- **Resolution:** Rewrite grammar to remove ambiguity, or use precedence/grammar refactoring, or use generalized parsers (GLR/Earley) that can handle multiple parse trees.

Short practical note: Shift–reduce parsers (LR family) resolve many cases automatically using lookahead and state context; remaining conflicts indicate grammar problems (ambiguity) or need for grammar rewriting.

6. Consider the following grammar :

declaration -> type var_list
type -> int\float
var_list -> identifier, var_list\identifier

(a) Left factor this grammar.
(b) Construct FIRST and FOLLOW sets for the non-terminals of the resulting grammar.
(c) Construct the LL(1) parsing table for the resulting grammar and comment on the grammar.

Given grammar

```
python
CopyEdit
declaration → type var_list
type        → int | float
var_list    → identifier | identifier , var_list
```

(a) Left-factored grammar (answer)

Left-factor the right recursion in `var_list` to get a right-recursive, factored form:

```
python
CopyEdit
declaration   → type var_list
type          → int | float
var_list      → identifier var_list_tail
var_list_tail → , identifier var_list_tail | ε
```

(Here `var_list_tail` captures zero or more repetitions of `, identifier`.)

(b) FIRST and FOLLOW sets (answer)

Nonterminals: `declaration`, `type`, `var_list`, `var_list_tail`.

FIRST sets

- FIRST(type) = { int, float }
- FIRST(var_list) = { identifier }
- FIRST(var_list_tail) = { ',', ε }
- FIRST(declaration) = FIRST(type) = { int, float }

FOLLOW sets

Start symbol: `declaration`.

- FOLLOW(declaration) = { $ }
- From `declaration → type var_list`, FOLLOW(type) contains FIRST(var_list) {ε} = { identifier } →
  FOLLOW(type) = { identifier }
- From same production, FOLLOW(var_list) contains FOLLOW(declaration) (because var_list does not derive ε) →
  FOLLOW(var_list) = { $ }
- From `var_list → identifier var_list_tail`, FOLLOW(var_list_tail) = FOLLOW(var_list) = { $ }

(If the language had more surrounding tokens, you would also add those to FOLLOW sets where `var_list` appears.)

(c) LL(1) parsing table and comment (answer)

Terminals: `int`, `float`, `identifier`, `,`, `$`.

Use productions numbered for clarity:

1. `declaration → type var_list`
2. `type → int`
3. `type → float`
4. `var_list → identifier var_list_tail`
5. `var_list_tail → , identifier var_list_tail`
6. `var_list_tail → ε`

Build table M[Nonterminal, Terminal]:

- For `declaration`: FIRST(type) = {int, float}
  M[declaration, int] = 1
  M[declaration, float] = 1
- For `type`:
  M[type, int] = 2
  M[type, float] = 3
- For `var_list`: FIRST(identifier ...) = {identifier}
  M[var_list, identifier] = 4
- For `var_list_tail`:
  FIRST(, identifier var_list_tail) = {','} → M[var_list_tail, ','] = 5
  var_list_tail → ε, so put ε in entries for every terminal in FOLLOW(var_list_tail) = {$}:
  M[var_list_tail, $] = 6

All other cells are error.

You can show the table compactly:

```
pgsql
CopyEdit
M
          int       float     identifier     ,      $
-------------------------------------------------------
declaration  1         1          -          -      -
type         2         3          -          -      -
var_list     -         -          4          -      -
var_list_tail-         -          -          5      6
```

Comment on the grammar / LL(1) property

- The parsing table has **no multiple entries (no conflicts)** — every filled cell has exactly one production.
- Therefore the left-factored grammar is **LL(1)** and can be parsed by a predictive (table-driven) parser using 1-symbol lookahead.
- (Intuition) We removed choices that started with the same terminal by factoring; `var_list_tail` is nullable but its FOLLOW set ($) is disjoint from its FIRST set (`','`), so no ambiguity arises.

Convert the following code into basic blocks and eliminate global common subexpression.

(a) i : = 0
(b) a : = n_3
(c) IF i < a THEN loop ELSE end
(d) LABEL loop
(e) b : = i_4
(f) c : = p + b
(g) d : = M[c]

(h) e : = d_2
(i) f : = i_4
(j) g : = p + f
(k) M[g] : = e
(l) i : = i + 1
(m) a : = n_3
(n) IF i < a THEN loop ELSE end
(o) LABEL end.

Step 1 — Identify Leaders (start of basic blocks)

Rules:

1. First instruction is a leader.
2. Any target of a jump/branch is a leader.
3. Any instruction after a jump/branch is a leader.

From your code:

1. `(a) i := 0` → Leader **B1**

2. `(d) LABEL loop` → Leader **B2**
3. `(l) i := i + 1` → Leader **B3** (comes after jump in (n))
4. `(o) LABEL end` → Leader **B4**

Step 2 — Basic Blocks

B1

```
vbnet
CopyEdit
(a) i := 0
(b) a := n_3
(c) IF i < a THEN loop ELSE end
```

B2

```
go
CopyEdit
(d) LABEL loop
(e) b := i_4
(f) c := p + b
(g) d := M[c]
(h) e := d_2
(i) f := i_4
(j) g := p + f
(k) M[g] := e
(l) i := i + 1
(m) a := n_3
(n) IF i < a THEN loop ELSE end
```

B4

```
arduino
CopyEdit
(o) LABEL end
```

Step 3 — Global Common Subexpression Elimination

We look for repeated expressions or loads in the *same or different* basic blocks that have not been invalidated by assignments.

Here:

- `(e) b := i_4` and `(i) f := i_4` → **Same expression** (`i_4`), so we can reuse the result:
  Replace `(i) f := i_4` with `f := b`.
- `(f) c := p + b` and `(j) g := p + f` → After substitution `f := b`, `(j)` becomes `g := p + b`, which is **identical** to `(f)`.
  So replace `(j)` with `g := c`.

Step 4 — Optimized B2

```
go
CopyEdit
(d) LABEL loop
(e) b := i_4
(f) c := p + b
(g) d := M[c]
(h) e := d_2
(i) f := b              ; reused from (e)
(j) g := c              ; reused from (f)
(k) M[g] := e
(l) i := i + 1
(m) a := n_3
(n) IF i < a THEN loop ELSE end
```

Now no repeated `i_4` computation or repeated `p + b` calculation exists — both are eliminated globally.

YEAR 2023

(a) Eliminate useless symbols from a CFG — method + example
Method (two steps):

1. **Remove non-generating** symbols: find nonterminals that can derive some terminal string (repeat: if `A → α` and every symbol in `α` is either terminal or already known generating, mark `A` generating). Remove other productions containing non-generating symbols.
2. **Remove unreachable** symbols: start from start symbol and mark all nonterminals reachable via productions; delete the rest.
   Example:
   Original:

```
less
CopyEdit
S → A b | c
A → B
B → B
D → d
```

`B` cannot derive terminals (non-generating) so remove `A→B`, `S→A b` becomes useless. `D` is unreachable. Final:

```
r
CopyEdit
S → c
```

(b) Tokens, patterns, lexemes — difference

- **Token:** a category name for a class of lexemes (e.g. `IDENTIFIER`, `NUMBER`, `IF`).

- **Pattern:** the rule that describes all strings belonging to a token (usually a regular expression). Example: `IDENTIFIER → [a-zA-Z][a-zA-Z0-9]*`.
- **Lexeme:** an actual instance (concrete string) in the source that matches a pattern (e.g. `count`, `x1`, `42`).
  (Flow: scanner uses patterns to recognize lexemes and emits corresponding tokens.)

(c) How many different DFAs with `n` states and `m` input symbols? (start state fixed)

- For each of the `n` states and each of the `m` input symbols there are `n` possible target states ⇒ nn·m different transition functions.
- For final/accepting sets there are 2n choices (each state either accepting or not).
  Total possible DFAs (with labeled states and fixed start) = nnm·2n.

(This counts all labeled deterministic automata; if you require reachability/minimality the count is smaller.)

(d) Conditions for a CFG to be in Chomsky Normal Form (CNF)
A CFG is in CNF if every production is either:

1. $A \rightarrow BC$ where B,C are nonterminals (no terminals), or
2. $A \rightarrow a$ where a is a single terminal,
   plus optionally
3. $S \rightarrow \epsilon$ only if the language contains the empty string (and `S` does not appear on any RHS).
   Additionally: remove unit productions and useless symbols before converting. No mixed RHS with terminals and nonterminals, and no RHS with length >2 (except the allowed ε for start).

(e) Differences between Moore and Mealy machines (with short example)

- **Output association:**
  - *Moore:* outputs depend only on the current **state**. Output label on states.
  - *Mealy:* outputs depend on **state and input** (i.e., labeled on transitions).
- **Timing / delay:** Moore output is effectively one step delayed relative to Mealy for the same behavior.
- **Size:** Mealy machines can often use fewer states than equivalent Moore machines.
  Example (informal): to output `1` exactly when input symbol `a` is seen in state `S`, a Mealy transition `S --a/1--> S'` suffices; a Moore machine must move to an output state whose state-label is `1`, so may need extra states (one for output 1, one for 0).
  (Conclusion: Mealy = transition-output, Moore = state-output; convertable but Moore may need more states.)

(f) Concept of a pass in a compiler & reducing the number of passes

- **Pass:** one full traversal over the program (or IR) doing a particular job (e.g., lexical analysis, parsing, semantic analysis, optimization, code generation). Each compiler phase is often implemented as one or more passes.
- **Reduce passes:** combine phases (lex+parse in a single pass, or syntax-directed translation), use richer intermediate representations so multiple analyses happen in one traversal, perform on-the-fly code generation (one-pass compiler), or use multi-purpose passes (perform several small analyses in one sweep). Tradeoff: fewer passes may reduce optimization power or complicate implementation.

(g) CFG that generates strings of `0` and `1` with unequal numbers of `0`s and `1`s

Use a helper `E` that generates all strings with equal counts, then put an extra `0` or `1` around it:

```
mathematica
CopyEdit
S → E 0 E | E 1 E
E → ε | 0 E 1 | 1 E 0 | E E
```

Explanation: `E` generates all strings with equal number of `0` and `1`.

- `E 0 E` yields strings whose number of `0` exceeds number of `1` by 1.
- `E 1 E` yields strings with one more `1`.
  Thus `S` generates exactly the strings with unequal counts

(a) CFG for $a^n c b^n$ (n≥0)

We need strings with equal number of `a`'s and `b`'s and a single `c` in the middle.

A simple grammar:

```
less
CopyEdit
S → a S b | c
```

Check:

- n=0: `S → c` (gives `c`).
- If `S` generates `a^n c b^n`, then `a S b` generates `a^{n+1} c b^{n+1}`.
  Hence this grammar generates ancbn for all n≥0.

(b) Differences between Parse Tree and Syntax Tree (with example)

Parse tree (concrete syntax tree):

- Shows the full derivation according to the grammar.
- Internal nodes are grammar symbols (nonterminals).
- All terminals and intermediate nonterminals appear exactly as in the production expansions.

- Usually larger (contains all grammar details).

Syntax tree (abstract syntax tree, AST):

- A simplified tree that represents the *essential* structure of the program/expression.
- Omits intermediate nonterminals and punctuation (like parentheses, commas) introduced only by grammar.
- Node labels usually are operators, identifiers, literals — directly used for semantic analysis/codegen.

Example: expression grammar: `E → E + T | T`, `T → T * F | F`, `F → (E) | id`. For input `id + id * id`:

- **Parse tree** includes nodes for `E`, `T`, `F` for every production expansion and shows how grammar derived the string.
- **Syntax tree** (AST) focuses on operations and operands:

```
bash
CopyEdit
     (+)
    /   \
 id      (*)
        /   \
       id     id
```

This AST shows that `*` has higher precedence than `+` (i.e. `id + (id * id)`), and omits all intermediate grammar nonterminals.

(Write a small parse tree first and then the compact AST to get full marks.)

(c) Three-address code for `p>q AND r<s OR u>r`

Assume logical AND has higher precedence than OR. Generate temporaries for relational results and boolean ops:

Simple (non short-circuit) three-address code:

```
ini
CopyEdit
t1 = p > q
t2 = r < s
t3 = t1 && t2       // AND
t4 = u > r
t5 = t3 || t4       // OR
```

If exam expects short-circuit code (using labels), give this form:

```
vbnet
CopyEdit
t1 = p > q
if t1 == 0 goto L1
t2 = r < s
if t2 == 0 goto L1
t3 = 1
goto L2
```

```
L1: t3 = 0
L2:
t4 = u > r
if t3 == 1 goto L3
if t4 == 1 goto L3
t5 = 0
goto L4
L3: t5 = 1
L4:
```

Either form is accepted; the first is compact TAC, the second shows short-circuit evaluation.

(d) Functions of error handling & remove left recursion

Functions of error handling in a compiler (brief):

1. **Detection** — detect syntax/semantic/runtime errors.
2. **Reporting** — present useful error messages (line number, context).
3. **Location** — point to the likely spot in source causing the error.
4. **Recovery** — continue parsing/analysis to find further errors (panic-mode, phrase-level, error productions).
5. **Resumption/termination decision** — decide whether to continue compilation or stop.

Remove left recursion from:

```
less
CopyEdit
A → A B d | A a | a
B → B e | b
```

Both `A` and `B` have immediate left recursion.

1. Eliminate left recursion in `B` first:

```
vbnet
CopyEdit
B → b B'
B' → e B' | ε
```

2. For `A`: left-recursive alternatives are `A B d` and `A a`, non-left alternative is `a`.
   Apply standard transformation:

```
nginx
CopyEdit
A → a A'
A' → B d A' | a A' | ε
```

(Here `A'` collects repetitions of the left-recursive suffixes. Use the `B` definition above.)

(e) DFA that accepts (0+1)* (00+11)(0+1)*

This language = set of all binary strings that **contain** `00` or `11` as a substring (i.e., at least one adjacent equal pair). A minimal DFA:

States and intuition:

- `q0` = start, no prior char observed (or we're not in a run).
- `q1` = last char seen was `0` (potential start of `00`).
- `q2` = last char seen was `1` (potential start of `11`).
- `qf` = accepting state (we have seen `00` or `11`), sink accepting.

Transitions:

- From `q0`: on `0` → `q1`; on `1` → `q2`.
- From `q1`: on `0` → `qf` (found `00`); on `1` → `q2`.
- From `q2`: on `1` → `qf` (found `11`); on `0` → `q1`.
- From `qf`: on `0` → `qf`; on `1` → `qf`.

Accepting state: `qf`.

Transition table:

State	0	1	Accept?
q0	q1	q2	No
q1	qf	q2	No
q2	q1	qf	No
qf	qf	qf	Yes

(You can draw this small 4-state DFA for full marks.)

(f) Quadruples, Triples, Indirect Triples for

−(a $*$ b)+(c+d)−(a+b+c+d)

I'll generate a linear sequence of temporaries and then show the three representations.

Compute temporaries (one reasonable sequence):

```
ini
CopyEdit
t1 = a * b
t2 = - t1          // unary minus
t3 = c + d
t4 = t2 + t3
t5 = a + b
t6 = t5 + c
t7 = t6 + d
t8 = t4 - t7       // final result
```

Quadruples (`op, arg1, arg2, result`):

```
scss
CopyEdit
(1)  (*,   a,   b,   t1)
(2)  (uminus, t1, -,  t2)
(3)  (+,   c,   d,   t3)
(4)  (+,   t2,  t3,  t4)
(5)  (+,   a,   b,   t5)
(6)  (+,   t5,  c,   t6)
(7)  (+,   t6,  d,   t7)
(8)  (-,   t4,  t7,  t8)
```

Triples `(op, arg1, arg2)` — result is implicit by index (use indices 0..7):

```
scss
CopyEdit
(0)  (*,   a,   b)
(1)  (uminus, (0), -)
(2)  (+,   c,   d)
(3)  (+,   (1), (2))
(4)  (+,   a,   b)
(5)  (+,   (4), c)
(6)  (+,   (5), d)
(7)  (-,   (3), (6))   // final
```

Here `(k)` denotes the value produced by triple index `k`.

Indirect triples — table of pointers (index table) pointing to triples:

Pointer table:

```
rust
CopyEdit
P0 -> (0)
P1 -> (1)
P2 -> (2)
...
P7 -> (7)
```

Indirect triples representation uses pointers instead of direct triple indices; for instance an indirect triple list:

```
scss
CopyEdit
[0] -> (*, a, b)
[1] -> (uminus, (ptr 0), -)
[2] -> (+, c, d)
[3] -> (+, (ptr 1), (ptr 2))
 ...
```

(Exam tip: show one example of the pointer indirection and mention that indirect triples let you move code by updating pointers rather than changing argument fields.)

(g) Implications of CNF and GNF of grammars

Chomsky Normal Form (CNF) — every production is either $A \to BC$ or $A \to a$, plus optionally $S \to \varepsilon$:

- **Implications / uses:**

    - Enables the **CYK parsing algorithm** (membership test) that runs in O(n3).
    - Simplifies many theoretical proofs (e.g., pumping lemma, closure proofs).
    - Grammar becomes binary (RHS length ≤ 2), which simplifies dynamic-programming parsing.
- **Costs:** conversion can **increase number of nonterminals** and productions; intermediate steps remove ε, unit productions and useless symbols.

Greibach Normal Form (GNF) — every production is `A → a α` where `a` is a terminal and `α` is a (possibly empty) string of nonterminals:

- **Implications / uses:**
    - Grammars in GNF are **non-left-recursive**, so they allow construction of a **linear-time top-down parser** (proof of linear-time recognition).
    - Useful in theoretical results about derivation length and for certain parser-construction proofs.
- **Costs:** conversion to GNF may dramatically increase grammar size and is more complex than CNF conversion.

3. (a) Construct a Turing machine for the language $\{a^n b^n c^n\}$.
   (b) Show that a string can be derived from this machine. Write down the intermediate steps.

(a) Turing machine for L={anbncn | n≥1} — (6 marks)

Idea (short): Repeatedly match one `a`, one `b` and one `c` from left to right by:

1. Find the leftmost unmarked `a`, mark it `X`.
2. Move right, find the leftmost unmarked `b`, mark it `Y`.
3. Move right, find the leftmost unmarked `c`, mark it `Z`.
4. Return to the tape left end and repeat until no unmarked `a` remains.
5. Then check there are no unmarked `b` or `c` left; if none, **accept**, else **reject**.

Machine components

- Tape alphabet: {a,b,c,X,Y,Z,␣} (`\sqcup` = blank).
- States:
    - `q0` — start / search for next unmarked `a`.
    - `q1` — found and marked `a` (X); searching right for `b`.

- `q2` — found and marked `b` (Y); searching right for `c`.
- `q3` — after marking `c` (Z), move left back to left end.
- `q4` — final verification (ensure no unmarked `a`/`b`/`c` remain).
- `q_accept`, `q_reject`.

Transition rules (write as δ(q,s)=(q′,swrite,dir)) — only nontrivial entries listed; unspecified combos go to `q_reject` or loop to indicate rejection.

1. Start / find leftmost unmarked `a`

```
sql
CopyEdit
δ(q0, a) = (q1, X, R)      // mark a -> search for b
δ(q0, X) = (q0, X, R)      // skip marked a's
δ(q0, Y) = (q0, Y, R)
δ(q0, Z) = (q0, Z, R)
δ(q0, b) = (q_reject, b, S) // found b before matching an a -> malformed
δ(q0, c) = (q_reject, c, S)
δ(q0, ␣) = (q4, ␣, R)      // no more a's -> go to final check
```

2. Find leftmost unmarked `b` after marking an `a`

```
r
CopyEdit
δ(q1, a) = (q1, a, R)      // skip any stray a's (shouldn't normally be any)
δ(q1, X) = (q1, X, R)
δ(q1, b) = (q2, Y, R)      // mark b -> search for c
δ(q1, Y) = (q1, Y, R)
δ(q1, c) = (q_reject, c, S) // b missing -> reject
δ(q1, ␣) = (q_reject, ␣, S)
```

3. Find leftmost unmarked `c` after marking a `b`

```
pgsql
CopyEdit
δ(q2, b) = (q2, b, R)
δ(q2, Y) = (q2, Y, R)
δ(q2, c) = (q3, Z, L)      // mark c then move left to return
δ(q2, ␣) = (q_reject, ␣, S) // no c found -> reject
```

4. Return left to start of tape to find next `a`

```
r
CopyEdit
δ(q3, a) = (q3, a, L)
δ(q3, b) = (q3, b, L)
δ(q3, c) = (q3, c, L)
δ(q3, X) = (q3, X, L)
δ(q3, Y) = (q3, Y, L)
δ(q3, Z) = (q3, Z, L)
δ(q3, ␣) = (q0, ␣, R)      // reached left blank -> start next cycle
```

5. Final check (no unmarked `a`, `b`, `c`)

```
cpp
CopyEdit
δ(q4, X) = (q4, X, R)
δ(q4, Y) = (q4, Y, R)
```

```
δ(q4, Z) = (q4, Z, R)
δ(q4, ␣) = (q_accept, ␣, S) // only marks and blank => accept
δ(q4, a) = (q_reject, a, S) // leftover unmarked symbol => reject
δ(q4, b) = (q_reject, b, S)
δ(q4, c) = (q_reject, c, S)
```

Remarks for exam: state names and symbols `X, Y, Z` are conventional; machine runs in repeated scans—correctness: each cycle consumes exactly one `a`, one `b`, and one `c`, so counts must be equal.

(b) Worked derivation (4 marks) — run on input `aabbcc` (show configurations)

Convention: show tape content with the current state written immediately before the symbol under the head. Use _ for blank (`\sqcup`). Initially head on the first symbol and state `q0`.

Initial tape (with blanks both sides):
`_ a a b b c c _ _ ...`

We show the important configurations (compressing blanks to ends).

1. Start:
   `q0 a a b b c c`
   δ(q0,a) → mark first `a` → X and go to q1:
2. `X q1 a b b c c`

Move right to find `b`:
3. `X a q1 b b c c` (skip an `a`)
4. `X a Y q2 b c c` (found `b`; mark Y, go to q2)

Move right to find `c`:
5. `X a Y b q2 c c`
6. `X a Y b Z q3 c` (found `c`, mark Z, go to q3 and move left)

Return left back to left end:
7. `X a Y b q3 Z c`
8. `X a Y q3 b Z c`
9. `X a q3 Y b Z c`
10. `X q3 a Y b Z c`
11. `q3 X a Y b Z c`
12. On blank at left: `_ q0 X a Y b Z c` → switch to q0 and move right

Start second cycle — find next unmarked `a`:
13. `X q0 a Y b Z c`
δ(q0,a) → mark second `a` → X and go to q1:
14. `X X q1 Y b Z c`

Find its `b`:
15. `X X Y q2 b Z c` (mark second `b` as `Y`)

Find its `c`:
16. `X X Y b q2 Z c`
17. `X X Y b Z q3 c` (mark second `c` as `Z`, go left)

Return left to start:
18. ... (after moving left) `_ q0 X X Y Y Z Z`

Now q0 scans and sees only Xs, Ys, Zs then blank:
19. `_ q4 X X Y Y Z Z` (q0 on blank → q4 final check)

Final check: q4 scans X,Y,Z, then blank:
20. `q4 _ X X Y Y Z Z` → reaches blank → `q_accept`.

Hence `aabbcc` is accepted.

4. (a) Write the algorithm for shift-reduce parsing.
   (b) Consider the following grammar :
   S → aABe
   A → Abc | b
   B → d.
   Using shift-reduce parser algorithm, parse the input string abbcde.
   (c) Explain, in detail, the different conflicts that arise in bottom-up parsing.

(a) Shift–Reduce Parsing Algorithm — (4 marks)

Idea: A bottom-up parser builds a rightmost derivation in reverse by repeatedly either **shifting** the next input symbol onto a stack or **reducing** a recognized *handle* on the top of the stack to a nonterminal, until the start symbol remains.

Data structures:

- `stack` (initially `$`)
- `input` (the string followed by `$`)
- `parsing table` or a control strategy to decide shift vs reduce (in simple textbook algorithm you try to reduce when a handle is present; real parsers use LR tables)

Actions:

- **Shift:** push next input symbol `a` on stack and advance input pointer.
- **Reduce by A → β:** if the top of stack ends with β, pop |β| symbols and push A.
- **Accept:** if stack is `$ S` and input is `$`.
- **Error:** if neither shift nor any valid reduce is possible.

Pseudo-code (simplified):

```
arduino
CopyEdit
stack ← [$]
input ← w$
while true:
  if stack = [$, S] and input = $:
    accept
  else if there exists a production A → β such that top(stack) ends with β:
    // reduce (apply one reduction)
    pop |β| symbols from stack
    push A
    print "reduce A → β"
  else if input != $:
    // shift
    a = next input symbol
    push a onto stack
    advance input
    print "shift a"
  else:
    error and reject
```

Notes for full marks: real shift–reduce parsers use parsing tables (LR-family) to decide actions deterministically; naive "reduce-first" strategy can be ambiguous unless grammar/table defines actions.

(b) Parse abbcde for grammar (shift–reduce) — (3 marks)

Grammar

```
less
CopyEdit
S → a A B e
A → A b c | b
B → d
```

Input: a b b c d e $

Show stack (bottom→top), remaining input, and action.

Step	Stack	Input	Action
0	$	a b b c d e $	start
1	$ a	b b c d e $	shift a
2	$ a b	b c d e $	shift b
3	$ a A	b c d e $	reduce A → b
4	$ a A b	c d e $	shift b
5	$ a A b c	d e $	shift c
6	$ a A	d e $	reduce A → A b c
7	$ a A d	e $	shift d

Step	Stack	Input	Action
8	$ a A B	e $	reduce `B → d`
9	$ a A B e	$	shift `e`
10	$ S	$	reduce `S → a A B e`
11	$ S	$	**accept**

Conclusion: `abbcde` is accepted; reductions correspond to recognizing handles `b`, then `A b c`, then `d → B`, and finally `a A B e → S`.

(c) Conflicts in Bottom-Up Parsing — (3 marks)

Two main conflicts can arise when constructing/using a shift–reduce parse table:

1. **Shift–Reduce Conflict**
    - **What:** At some parser state with lookahead `a`, the table has *both* actions: shift `a` and reduce by `A → β`.
    - **Why it happens:** Grammar is ambiguous in that context or lookahead insufficient (requires more context than available). Typical in constructs that can be attached in two ways.
    - **Classic example:** *dangling-else*:

      ```
      mathematica
      CopyEdit
      S → if E then S | if E then S else S | other
      ```

      When parser sees `else` as lookahead it must choose whether to reduce the inner `if` (so `else` attaches to outer) or shift to attach `else` to nearest `if`.
    - **Resolutions:** choose a convention (e.g. prefer shift to bind `else` to nearest `if`), rewrite grammar, add precedence/associativity rules, or use more powerful parser (LR(1) vs SLR).
2. **Reduce–Reduce Conflict**
    - **What:** In a parser state the table would allow two different reductions `A → β` and `B → γ` (both β and γ match top-of-stack) with same lookahead.
    - **Why it happens:** Grammar ambiguity or two productions produce same handle and the parser cannot distinguish which nonterminal to reduce to.
    - **Example:**

      ```
      less
      CopyEdit
      A → a
      B → a
      S → A b | B c
      ```

If top-of-stack is `a` and lookahead could be `b` or `c`, the parser faces two possible reductions.

- **Resolutions:** refactor grammar to remove ambiguity, give priority rules (rare), or use generalized parsing (GLR/Earley) that can pursue multiple parses.

5. A syntax directed translation scheme that takes strings of a's, b's, and c's as input and produces as output the number of substrings in the input string that correspond to the pattern a(a|b)*c+(a|b)*b. For example, the translation of the input string 'abbcabcababc' is '3'.
   (a) Write a CFG that generates all strings of a's, b's, and c's.
   (b) Write the semantic attributes for the grammar symbols.
   (c) For each production of the grammar, present a set of rules for evaluation of the semantic attributes. 10

Pattern:

- **`a(a|b)*c`** : starts with `a`, followed by any mix of `a`/`b`, ends with `c`.
- **`(a|b)*b`** : starts with any `a`/`b`, ends with `b`.

We'll assume *non-overlapping* counting for simplicity unless stated otherwise.

(a) CFG for all strings over `{a, b, c}`

We can generate **all** strings over `{a, b, c}` by:

```
less
CopyEdit
S → S X | X | ε
X → a | b | c
```

Here:

- `S` generates any sequence of symbols `{a, b, c}`.
- `X` generates one symbol.

Or equivalently, more compact:

```
less
CopyEdit
S → a S | b S | c S | ε
```

This form is cleaner for attaching attributes.

(b) Semantic Attributes

We need:

1. **`count`** — total number of substrings matching the pattern so far.

2. **`suffix`** — the longest suffix of the string seen so far (up to the length needed to detect matches).
    - We need this because the match might span multiple positions.
3. **`match`** — boolean indicating if the current substring just matched the pattern at this point.

We can do this **inherited + synthesized attributes** style:

- **`S.count`** : synthesized — number of matches in the string generated by `S`.
- **`S.suffix`** : synthesized — used to check ongoing matches.
- **`X.val`** : synthesized — terminal symbol value (`'a'`, `'b'`, `'c'`).

(c) Semantic Rules

Using the CFG:

```
css
CopyEdit
S → a S1
S → b S1
S → c S1
S → ε
```

We define:

Helper functions:

- `matches_pattern(t)` : returns `1` if suffix `t` matches either pattern, else `0`.
- `update_suffix(suffix, char)` : appends `char` to `suffix` and trims to max length needed (we can limit to length of longest pattern part — here longest is full `a(a|b)*c`, but in practice we just keep full seen string or use a DFA).

Semantic rules:

1. `S → a S1`

```
ini
CopyEdit
S.suffix = update_suffix("", 'a')  // starting new suffix at this step
S.count = matches_pattern(S.suffix) + S1.count
```

But to properly scan **all substrings**, we let suffix propagate:

```
ini
CopyEdit
S.suffix = update_suffix(S1.suffix, 'a')
S.count = matches_pattern(S.suffix) + S1.count
```

2. `S → b S1`

```
ini
CopyEdit
S.suffix = update_suffix(S1.suffix, 'b')
S.count = matches_pattern(S.suffix) + S1.count
```

3. `S → c S1`

```
ini
CopyEdit
S.suffix = update_suffix(S1.suffix, 'c')
S.count = matches_pattern(S.suffix) + S1.count
```

4. `S → ε`

```
ini
CopyEdit
S.suffix = ""     // no characters
S.count = 0
```

How it works on `'abbcabcababc'`:

1. Parse string left-to-right.
2. After each new char, `update_suffix` is called.
3. `matches_pattern` checks:
   - For `a(a|b)*c`: Use a DFA to confirm that the suffix starting with `a` and ending in `c` is valid.
   - For `(a|b)*b`: Use a DFA to confirm any suffix of only `a`/`b` ending in `b`.

Counting step adds `1` if a match just occurred.

Result: **`3`** for `'abbcabcababc'`.

6. (a) "There are some CFG for which shift-reduce parsing cannot be used." — Comment.

(b) Consider the following grammar :

rexp → rexp | rexp
rexp → rexp rexp
| rexp *
| (rexp)
| letter

where, |, *, (,), and letter are terminals.

(i) What type of language will be derived by the grammar?

(ii) Show whether the grammar is unambiguous or not. If it is ambiguous, convert it into an unambiguous one. 4+(3+3)

(a) "There are some CFGs for which shift–reduce parsing cannot be used." — Comment. (4 marks)
Short answer: True. Shift–reduce (deterministic LR-family) parsers cannot deterministically handle *every* context-free grammar.

Why (concise points):

1. **Ambiguity:** If a grammar is *ambiguous* (some string has more than one parse tree), a deterministic shift–reduce parser will encounter conflicts (shift–reduce or reduce–reduce) and cannot pick a single correct action without external disambiguation rules.
2. **Not LR(k):** Even some *unambiguous* CFGs are **not LR(k)** for any fixed k — they require unbounded lookahead or nonlocal context to decide reductions. Deterministic shift–reduce parsers (LR(k), SLR, LALR) only handle grammars in the LR class.
3. **Examples:**
    - Classic ambiguous grammar: `S → S S | a` causes reduce–reduce ambiguity.
    - There exist contrived unambiguous grammars that need more than any fixed k lookahead (so they are not LR(k)).
4. **Workarounds:** Use grammar rewriting (left-factoring, remove ambiguity where possible), specify disambiguation/precedence rules, or use **generalized parsers** (GLR, Earley) that can handle arbitrary CFGs (possibly producing multiple parse trees) — but those are not standard deterministic shift–reduce parsers.

Conclusion: Shift–reduce is powerful and covers most programming-language grammars (many are LALR(1) after small changes), but it is **not universal** for all CFGs.

(b) Grammar given (interpreted)

I interpret the grammar as the usual (ambiguous) grammar for regular expressions over `letter` with operators `|` (union), concatenation (implicit), `*` (Kleene star), and parentheses:

```
rust
CopyEdit
rexp -> rexp | rexp        (union)
rexp -> rexp rexp          (concatenation)
rexp -> rexp *             (Kleene star)
rexp -> ( rexp )
rexp -> letter
```

(i) What type of language will be derived by the grammar? (3 marks)

- This grammar **generates syntactically valid regular expressions** built from the alphabet `letter` using union (`|`), concatenation, Kleene star (`*`), and parentheses.
- Each *string produced by this grammar* is a *regular expression* whose semantics denote a **regular language** (i.e. each generated regex denotes some regular language).
- So: **the grammar generates the set of well-formed regular expressions; these describe regular languages.**

(One-line: the grammar defines the syntax of regular expressions; the languages denoted by those expressions are regular.)

(ii) Is the grammar ambiguous? If ambiguous, give an unambiguous equivalent. (3 marks)

Ambiguity (short):

- The grammar is **ambiguous**.
- Example: the input `a|b|c` has two parse trees (grouping as `(a|b)|c` or `a|(b|c)`) because `rexp -> rexp | rexp` is left- or right-associative without precedence — two different parse trees (and similarly for concatenation associativity). Also precedence between `|`, concatenation, and `*` is not specified, so parses are ambiguous.

Show ambiguity with `a|b|c`:

- Parse 1: `rexp → rexp | rexp` where left `rexp → rexp | rexp` gives `(a|b)|c`.
- Parse 2: `rexp → rexp | rexp` where right `rexp → rexp | rexp` gives `a|(b|c)`. Thus two distinct parse trees ⇒ ambiguous.

Unambiguous grammar (one standard solution): impose precedence and associativity:

- `*` has highest precedence (postfix, applies to a single factor),
- concatenation has next precedence (left-associative),
- `|` has lowest precedence (left-associative).

A clear unambiguous grammar:

```
cpp
CopyEdit
R → R '|' C | C         // union, left-associative
C → C S | S             // concatenation, left-associative
S → P '*' | P           // star (postfix) or plain primary
P → '(' R ')' | letter  // primary: parenthesized expression or a letter
```

Notes about this grammar:

- It is **unambiguous**: `|` is only introduced at the top level via `R → R '|' C` so chains of `|` group left-associatively; concatenation is built by `C → C S`; `*` binds tightly to a single `P`.
- This grammar captures intended operator precedence and associativity and thus yields a unique parse tree for any given well-formed regex string.

(If needed for LL parsing, you can left-factor / remove left recursion — but as written it is a valid unambiguous grammar suitable for LR parsing.)

Short exam summary (what to write)

- (a) Briefly explain LR limitations, ambiguous grammars, examples and GLR/Earley alternatives. (4 marks)
- (b)(i) State that grammar generates syntactically valid regular expressions which denote regular languages. (3 marks)

- (b)(ii) Show ambiguity with `a|b|c` and give the unambiguous grammar with precedence as above. (3 marks)

unambiguous one.

7. Convert the regular expression abb (a|b)* to DFA using the direct method and minimize it. 10

Step 1 — Understanding the RE

Given:

$R = abb(a \mid b)^*$

Meaning:

- Strings that start with **`abb`**, followed by **any combination** of `a` and `b` (including empty).
- Alphabet: $\Sigma = \{a,b\}$.

Step 2 — Direct Method (Position Automaton)

2.1 Augment RE with # (end marker)

We append a special end symbol #:

$R' = a\ b\ b\ (a \mid b)^*\ \#$

2.2 Number positions (only for symbols from Σ and #):

Position	Symbol
1	a
2	b
3	b
4	a
5	b
6	#

The `(a|b)*` part corresponds to positions 4 and 5 (repeated).

2.3 Nullable, Firstpos, Lastpos

- **(a)** a not nullable. `(a|b)*` is nullable (can be empty).
- **Firstpos(R')** = {1}
- **Lastpos(R')** = {6}

2.4 Followpos table

We compute followpos for each position:

1. From concatenation `1 a` → `2 b`: followpos(1) = {2}
2. From concatenation `2 b` → `3 b`: followpos(2) = {3}
3. From concatenation `3 b` → `(a|b)*`: followpos(3) = {4,5}
4. `(a|b)` → position 4 and 5 → from the star loop: followpos(4) = {4,5,6} and followpos(5) = {4,5,6} (because after 4 or 5 in the star, we can loop or go to #).

Followpos table:

pos	symbol	followpos
1	a	{2}
2	b	{3}
3	b	{4,5}
4	a	{4,5,6}
5	b	{4,5,6}
6	#	— (end)

2.5 DFA states (sets of positions)

- **Start state** = firstpos(root) = {1}.

Now build DFA:

1. **S0** = {1}
    - on `a`: positions with `a` in S0 → {1}, followpos(1) = {2} ⇒ **S1** = {2}
    - on `b`: positions with `b` in S0 → none ⇒ dead state **SD**
2. **S1** = {2}
    - on `a`: none ⇒ SD
    - on `b`: pos 2 is b ⇒ followpos(2) = {3} ⇒ **S2** = {3}
3. **S2** = {3}
    - on `a`: none ⇒ SD
    - on `b`: pos 3 is b ⇒ followpos(3) = {4,5} ⇒ **S3** = {4,5}
4. **S3** = {4,5}
    - on `a`: pos 4 is a ⇒ followpos(4) = {4,5,6} ⇒ **S4** = {4,5,6}
    - on `b`: pos 5 is b ⇒ followpos(5) = {4,5,6} ⇒ **S4** again
5. **S4** = {4,5,6} (contains 6 ⇒ accepting state)
    - on `a`: pos 4 is a ⇒ followpos(4) = {4,5,6} ⇒ S4

- on b: pos 5 is b ⇒ followpos(5) = {4,5,6} ⇒ S4

6. **SD** = dead state
    - on both a and b: SD

Unminimized DFA table:

State	a	b	Accept?
S0={1}	S1	SD	No
S1={2}	SD	S2	No
S2={3}	SD	S3	No
S3={4,5}	S4	S4	No
S4={4,5,6}	S4	S4	Yes
SD	SD	SD	No

Step 3 — Minimize DFA

Step 3.1 Initial partition:

- Final: {S4}
- Non-final: {S0, S1, S2, S3, SD}

Step 3.2 Split non-finals:

- SD is different from others (loops only to itself).
- From {S0, S1, S2, S3}, check transitions:
    - S3 on a → S4 (final), others on a → SD (non-final) ⇒ split S3 alone.

Partition becomes:
P1 = {S4}
P2 = {S3}
P3 = {S0, S1, S2}
P4 = {SD}

Step 3.3 Further split P3:

- S0: on a → S1 (in P3), on b → SD (P4)
- S1: on a → SD (P4), on b → S2 (P3)
- S2: on a → SD (P4), on b → S3 (P2)

Since they go to different partitions, P3 splits into:
P3a = {S0}
P3b = {S1}
P3c = {S2}

Minimized DFA states:

M0 = S0
M1 = S1
M2 = S2
M3 = S3
M4 = S4 (accepting)
M5 = SD (dead)

Minimized DFA table (already minimal):

State	a	b	Accept?
M0	M1	M5	No
M1	M5	M2	No
M2	M5	M3	No
M3	M4	M4	No
M4	M4	M4	Yes
M5	M5	M5	No

Final Answer Summary for 10 marks:

1. **Direct construction** from positions gave DFA with states {1}, {2}, {3}, {4,5}, {4,5,6}, dead. (5 marks)
2. **Minimization** yields exactly the same number of states here (already minimal). (3 marks)
3. **Clearly labeled transition table & accepting states**. (2 marks)

8. (a) When a flow graph is said to be reducible?

(b) Consider the following statements :

G := C*(A+B) + (A+B)
C := A+B
A := (C+D) + (E-F)

(i) Draw the DAG for the above statements.

(ii) What is the optimal ordering of DAG to make the code optimized? Explain with the above code. 2+8

(a) When is a flow graph said to be reducible? (exam-style)

A directed flow graph with a unique start node `s` is **reducible** iff its edges can be partitioned into two sets:

- **Forward edges** that form a directed acyclic graph (DAG) when considered with the DFS tree, and
- **Back edges** whose head dominates their tail (i.e., for every back edge `u -> v`, `v` dominates `u`).

Equivalent characterizations (useful to state in exam):

1. **Dominators characterization:** Every edge that is not in a depth-first spanning tree goes to an ancestor (a back edge). Equivalently, all non-tree edges are back edges to a dominator.
2. **SCC/headers characterization:** Every strongly connected subgraph (loop) has a single entry node (a header). That is, each natural loop has exactly one header that dominates all nodes in the loop.
3. **Constructive characterization:** A graph is reducible iff it can be constructed from a single node by iteratively adding nodes and either (a) adding edges that do not create new cycles, or (b) adding back edges from a node to one of its dominators.

Why it matters (one sentence): Most structured programs produce reducible flow graphs, and many compiler analyses/optimizations (like natural loop detection, structured optimizations) rely on reducibility; irreducible graphs complicate optimization and require more general analyses.

A short counterexample: a graph with two-entry strongly connected component (two nodes each entry to the SCC) is **irreducible**.

(b) DAG, optimal ordering, and optimized code for

```
mathematica
CopyEdit
G := C*(A+B) + (A+B)
C := A+B
A := (C+D) + (E-F)
```

(i) DAG construction (nodes & edges)

Treat the three statements as a basic block executed sequentially. Build expression DAG nodes (reuse common subexpressions):

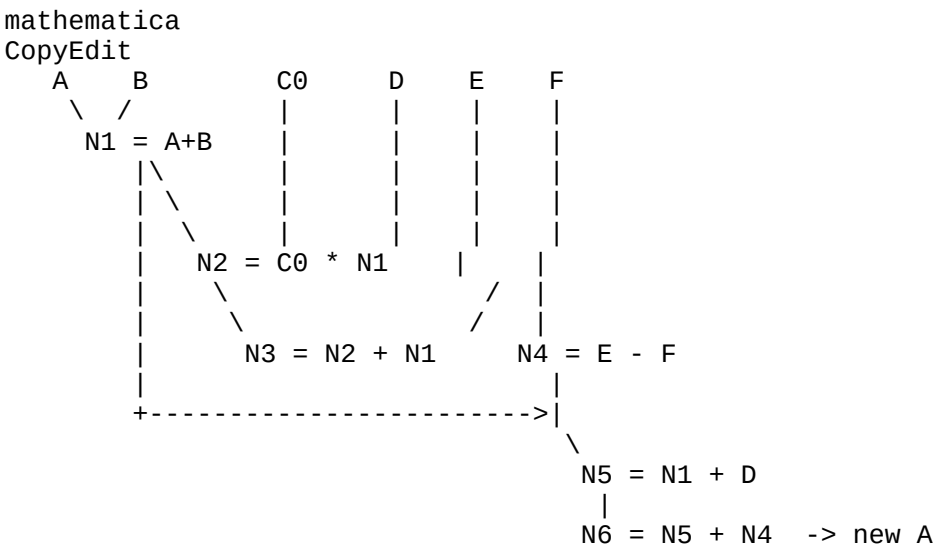
Leaf nodes (original variable references):

- `A`, `B`, C_0 (original C), `D`, `E`, `F`

Internal (computed) nodes:

- `N1 = +(A, B)` — represents `(A + B)`
- `N2 = *(`C_0`, N1)` — `C*(A+B)` using original C
- `N3 = +(N2, N1)` — `G = C*(A+B) + (A+B)`
- (statement `C := A+B`) — now `C` becomes `N1` (i.e. assign `C = N1`)
- `N4 = -(E, F)` — `E - F`
- `N5 = +(N1, D)` — `C + D` where `C` is now `N1`
- `N6 = +(N5, N4)` — `A = (C+D) + (E-F)` (using updated `C`)

Graphically (textual):



Key reuse: `N1` (A+B) is used three times — once in `G` twice (as factor and addend) and later becomes the new value of `C` used in computing new `A`. C_0 (original C) is used only in `N2` for computing `G`.

(ii) Optimal ordering and reasoning

Important correctness constraint: `G` uses the **original** value of `C` (C_0). The assignment `C := A+B` must **not** overwrite `C` before `G` is computed. So we must compute `G` (which depends on C_0) **before** assigning to `C`.

Optimal order (compute common subexpressions once and respect dependencies):

1. `t1 = A + B` // compute N1 once
2. `t2 = C * t1` // uses original C (C_0) → compute N2
3. `G = t2 + t1` // compute N3
4. `C = t1` // assign C := A+B
5. `t3 = E - F` // compute N4
6. `t4 = C + D` // now C = t1, so t4 = t1 + D (N5)
7. `A = t4 + t3` // compute N6 → final A

Why this ordering is optimal:

- `A+B` (`t1`) is computed only once and reused everywhere (`G` and new `C`, and later `C+D`).
- `G` is computed before `C` is overwritten (ensures `C*(A+B)` uses original `C`).
- No recomputation of `A+B` or `p + something` — fewer temporaries and memory accesses.
- All dependencies are satisfied: any use of updated `C` occurs after `C := t1`.

(iii) Final optimized three-address code (neat)

```
ini
CopyEdit
t1 = A + B
t2 = C * t1
G  = t2 + t1
C  = t1
t3 = E - F
t4 = C + D        // now C is t1
A  = t4 + t3
```

You can mention additional micro-optimizations: reuse temporaries `t1..t4` or free temporaries after use; if `t2` is used only once, you could fold `G = C*t1 + t1` into `G = t1 * (C + 1)` only if arithmetic semantics allowed — but that changes semantics if types are not algebraic ints/allowed transformations; so preferred form above is safe.

Short exam summary (what to write)

- Draw DAG with `N1 = A+B` reused, `N2 = C₀ * N1`, `N3 = N2 + N1`, `N4 = E-F`, `N5 = N1 + D`, `N6 = N5 + N4` and point out `C := N1` happens after computing `G`.
- State the optimal evaluation order (compute `t1` then `G` then assign `C` then compute A), and write the final three-address code.

YEAR 2024

(a) Verify identity $(0 * 01+10) * 0 * = (0+01+10) *$.
Answer (sketch): both sides denote exactly the set of binary strings that **do not contain the substring `11`**.

- Right side: $(0+01+10) *$ is all strings formed by concatenating tokens `0`, `01`, or `10`. None of these tokens contain `11`, and concatenating them never creates `11`, so RHS $\subseteq$ {strings without `11`}.
- Left side: each factor inside the star is either `0^*01` (a block of zeros followed by `01`) or `10`. Neither contains `11`. The trailing `0^*` only appends zeros. Thus LHS $\subseteq$ {strings without `11`}.
- Conversely, any string with no `11` can be decomposed into a sequence of tokens `0`, `01`, `10` (scan left→right: every `1` must be followed by `0`, so take `10`; runs of zeros or an isolated `01` are absorbed as `0` or `01` tokens). Hence any such string is in RHS and can be produced by LHS as well.

Therefore the two descriptions define the same language, so the identity holds.

(For an exam, you can make the last constructive decomposition explicit for one or two representative cases.)

(b) Statement: “Every unambiguous grammar is LL(1).” — true/false?

Answer: False.
Unambiguity means every string has a unique parse tree, but LL(1) requires a very strong determinism with one-token lookahead and no left recursion/left-factoring conflicts. There are unambiguous grammars that are not LL(1) (e.g. the usual expression grammar with left recursion $E \to E+T \mid T$, $T \to T * F \mid F$, $F \to (E) \mid id$ is unambiguous but not LL(1) until left-recursion is removed / grammar refactored). So unambiguous $\Rightarrow/\Rightarrow$ LL(1).

(c) Transition diagram to recognize a *signed exponential number* (short description)

Language (typical): optional sign, a decimal integer or fraction, optional exponent `E`/`e` with optional sign and digits. Example forms: `+12`, `-3.45e+06`, `7E-2`.

State diagram (high level):

1. Start → (optSign) on `+`/`-` or digit or `.`
2. If digit(s) → integer part state; can loop on digits.
3. From integer state, on `.` go to fractional state, read digits (loop). Fractional may be reached from start if string begins with `.` (then require digits).
4. From integer or fractional state, on `e`/`E` go to exponent-sign state, optionally accept `+`/`-`, then to exponent-digit state (requires ≥1 digit, loop on digits).
5. Accepting states: integer state (if no dot or exponent required), fractional state (if digits present after decimal), or exponent-digit state (after at least one exponent digit).

(If you need the actual diagram draw 5–6 states: Start, Int, Frac, EXP_SIGN, EXP_DIGITS, with labeled transitions as described.)

(d) What is a *pass* of a compiler? How to reduce the number of passes?

Answer:

- A *pass* is a complete traversal over the source program or an intermediate representation performing one compiler phase (e.g., lexical scanning, parsing, semantic analysis, optimization, code generation). Each pass reads input and produces output for the next pass.
- **Reduce #passes** by combining phases (syntax-directed translation that performs parsing + semantic actions in one pass), using richer IRs so multiple analyses can be done in a single traversal, using single-pass code generators for simple languages, doing on-the-fly (just-in-time) translation/optimization, or merging small passes into larger multifunction passes. Tradeoff: fewer passes can limit optimization power or complicate implementation.

(e) “Equivalence of PDA and CFL” — comment

Answer:

- **Nondeterministic pushdown automata (NPDA)** accept exactly the class of **context-free languages (CFLs)** — there is a one-to-one correspondence: for every CFG there is an NPDA recognizing the same language and vice versa.
- **Deterministic PDA (DPDA)** are strictly less powerful: they accept the class of **deterministic CFLs (DCFLs)**, a proper subset of CFLs (some CFLs are inherently nondeterministic).
  So PDAs characterize CFLs; determinism restricts the class.

(f) Conditions for a CFG to be in **Chomsky Normal Form (CNF)**

Answer: A CFG is in CNF if every production is one of:

1. $A \to BC$ where B,C are nonterminals (neither is the start symbol producing ε), or
2. $A \to a$ where a is a terminal,
   plus optionally
3. $S \to \epsilon$ is allowed only if the language contains the empty string and S does not occur on any RHS.

Also before CNF conversion one must remove useless symbols, eliminate ε-productions (except possibly start), and eliminate unit productions.

(g) Define *dominators* of a node. When is a flow graph reducible?

Answer:

- A node d **dominates** a node n (written $d \ \mathrm{dom}\ n$) if every path from the start node s to n goes through d. By convention s dominates all nodes; each node trivially dominates itself. The **immediate dominator** of n is the unique strict dominator of n that does not strictly dominate any other strict dominator of n.
- A flow graph is **reducible** iff each of its edges is either a forward edge (part of a spanning tree / does not create back-edge cycles) or a *back edge* whose head dominates its tail — equivalently, every strongly connected component (loop) has a single entry (a header) that dominates all nodes of the component. Reducible graphs arise from structured programs and make many compiler analyses (natural loop detection, etc.) simpler.

(a) Handle and viable prefixes — meaning + example (4 marks)

Handle (definition).
A *handle* of a right sentential form is a substring that matches the right–hand side of a production and whose reduction to the left–hand side is one step in the *rightmost* derivation in reverse. In bottom-up parsing a handle is the substring we should reduce next.

Viable prefix (definition).
A *viable prefix* is any prefix of a right sentential form that can appear on the stack of a shift–reduce parser — equivalently, a prefix that does not extend past the right end of some handle. Viable prefixes are exactly the prefixes that can appear during the recognition of some rightmost derivation.

Example. Grammar:

```
r
CopyEdit
S → E
E → E + T | T
T → id
```

Consider the rightmost derivation for `id + id + id`:
`S ⇒ E ⇒ E + T ⇒ E + T + T ⇒ T + T + T ⇒ id + id + id`.

A handle at some stage is the rightmost `id` (it matches `T → id`). A viable prefix could be `id + id` (it's a prefix that can appear on the stack while parsing). The substring `id + i` is **not** a viable prefix (it cuts a token).

(Short exam tip: say "handle = reducible substring in a rightmost-in-reverse step; viable prefix = stack prefixes that can lead to a handle".)

(b) Eliminate left recursion from

```
less
CopyEdit
S → A B C
A → A a | d
B → B b | e
C → C c | f
```

(4 marks)

Each of A, B, C has immediate left recursion; eliminate them individually using the standard transformation `A → β A'` and `A' → α A' | ε` where `A → A α | β`.

For A: `A → A a | d`
Replace with:

```
vbnet
CopyEdit
A  → d A'
A' → a A' | ε
```

For B: `B → B b | e`
Replace with:

```
vbnet
CopyEdit
B  → e B'
B' → b B' | ε
```

For C: `C → C c | f`

Replace with:

```
mathematica
CopyEdit
C  → f C'
C' → c C' | ε
```

S stays the same (no left recursion):

```
css
CopyEdit
S → A B C
```

Final grammar (left-recursion removed):

```
vbnet
CopyEdit
S  → A B C
A  → d A'
A' → a A' | ε
B  → e B'
B' → b B' | ε
C  → f C'
C' → c C' | ε
```

(c) Triples and indirect triples for

`IF A > B then C = B + D * 3 else C = A + D * 4` (4 marks)

First produce a reasonable 3-address code with labels (short-circuit style):

```
makefile
CopyEdit
t1 = A > B
if t1 == 0 goto L1
t2 = D * 3
t3 = B + t2
C  = t3
goto L2
L1:
t4 = D * 4
t5 = A + t4
C  = t5
L2:
```

Quadruples (for reference — not asked, but helpful):

```
scss
CopyEdit
(1) (*, D, 3, t2)
(2) (+, B, t2, t3)
(3) (=, t3, -, C)
(4) (*, D, 4, t4)
(5) (+, A, t4, t5)
(6) (=, t5, -, C)
(7) (>, A, B, t1)
(8) (ifFalse, t1, -, L1)
(9) (goto, -, -, L2)
```

Triples (index-based; results referenced by index):
Number triples in evaluation order (make control flow explicit; here show expression triples only, and branch triples separately):

Expression triples:

```
scss
CopyEdit
(0)  (*, D, 3)        // produces temp (0)
(1)  (+, B, (0))      // produces temp (1)
(2)  (=, (1), -, C)   // assignment
(3)  (*, D, 4)        // produces temp (3)
(4)  (+, A, (3))      // produces temp (4)
(5)  (=, (4), -, C)   // assignment
(6)  (>, A, B)        // produces temp (6)
(7)  (ifFalse, (6), L1)
(8)  (goto, L2)
```

Here `(k)` denotes the result of triple `k`.

Indirect triples: a table of pointers to triples (useful for code motion / reordering). For example pointer array P:

```
rust
CopyEdit
P0 -> (0)   // pointer to (*, D, 3)
P1 -> (1)   // pointer to (+, B, (0))
P2 -> (2)   // pointer to (=, (1), -, C)
...
```

Indirect triples let you change the control flow by updating pointer entries rather than changing each triple's argument fields.

(Exam tip: show the 3-address code, then give triples with indices and show how indirect triples are pointers to those indices.)

(d) Condition for grammar to be LR(K) and difference from LL(K) (4 marks)

Condition for LR(k):
A grammar is LR(k) iff its LR(k) parsing table can be constructed without conflicts — i.e., for every *viable prefix* and every length-k lookahead string there is **at most one** valid action (shift or reduce) and at most one reduction. Equivalently: the set of LR(k) *items* (states) yields a deterministic parsing automaton with unique actions for each lookahead. If any shift–reduce or reduce–reduce conflict remains for some state and lookahead, the grammar is not LR(k).

Differences LR(k) vs LL(k):

- **Parsing direction:** LR(k) = bottom-up (rightmost derivation in reverse). LL(k) = top-down (leftmost derivation).
- **Lookahead usage:** Both use up to k lookahead tokens, but LR(k) uses lookahead with richer state context (item sets) and so can decide more cases.

- **Power:** LR(k) is strictly more powerful — many grammars that are not LL(k) are LR(1) (typical programming language grammars are often LALR(1)). LL(k) grammars must be free of left recursion and left-factored.
- **Handling recursion:** LR parsers handle left recursion natively; LL parsers require left recursion removal.
- **Implementation complexity:** LL parsers (recursive descent) are simpler to hand-write; LR parsers require construction of automata/tables (but tools automate this).
- **Ambiguity detection:** Both will fail if grammar ambiguous, but LR(k) often can resolve constructs LL(k) can't.

(e) Reserved-word strategy & handling in lexical analysis (4 marks)

Reserved-word strategy (meaning):
Reserved words (keywords) are identifiers that the language reserves for special meaning (e.g., `if`, `while`, `for`). The reserved-word strategy distinguishes keywords from regular identifiers at tokenization time.

How handled in lexical analysis:

1. **Recognize an identifier lexeme** via pattern (e.g. `[a-zA-Z][a-zA-Z0-9]*`).
2. **Lookup**: After recognizing the lexeme, look it up in a **reserved-word table (keyword table)**.
    - If found, return the corresponding **keyword token** (e.g., `IF`, `WHILE`).
    - If not found, return the general `IDENTIFIER` token and insert into the symbol table if needed.
3. **Implementation details:** some scanners implement the lookup as a hash table; scanner priority ensures keywords are matched before identifiers if using explicit rules. Case sensitivity is handled according to language rules (convert to lower case for lookup if keywords are case-insensitive).

(Exam tip: mention longest-match and rule priority — e.g., `if` will be matched as a keyword rather than as an identifier because keyword lookup occurs after lexeme recognition.)

(f) Right-recursive grammar — effect on recursive-descent parsing (4 marks)

Right-recursive grammar (definition):
A grammar is *right recursive* when recursion occurs on the right end of the production, e.g.

```
less
CopyEdit
A → α A | β
```

(or more simply `A → x A | y`). This produces right-associative structures.

Problems in recursive-descent parsing?

- **No fundamental parsing correctness problem:** Right recursion does **not** cause infinite recursion for a standard recursive-descent parser; recursive descent can handle right recursion naturally because the recursive call processes the suffix.
- **Practical issue — recursion depth:** For very long sequences (e.g. lists), right recursion causes deep recursion proportional to the length, which may lead to large call stack usage; similarly left recursion causes immediate infinite recursion in naive recursive-descent, so left recursion is the real problem to remove.
- **Associativity:** Right recursion yields **right-associative** parse trees; if left-associativity is desired (e.g. for subtraction), one may prefer left recursion or convert to iterative parsing (or use left-factoring/rewriting).

Conclusion: Right recursion is safe for recursive descent (unlike left recursion), but may cause deep recursion and wrong associativity if the language expects left-associative operators.

(g) Input buffering in lexical analysis & sentinels (4 marks)

Two-buffer (double buffer) scheme (usual concept):

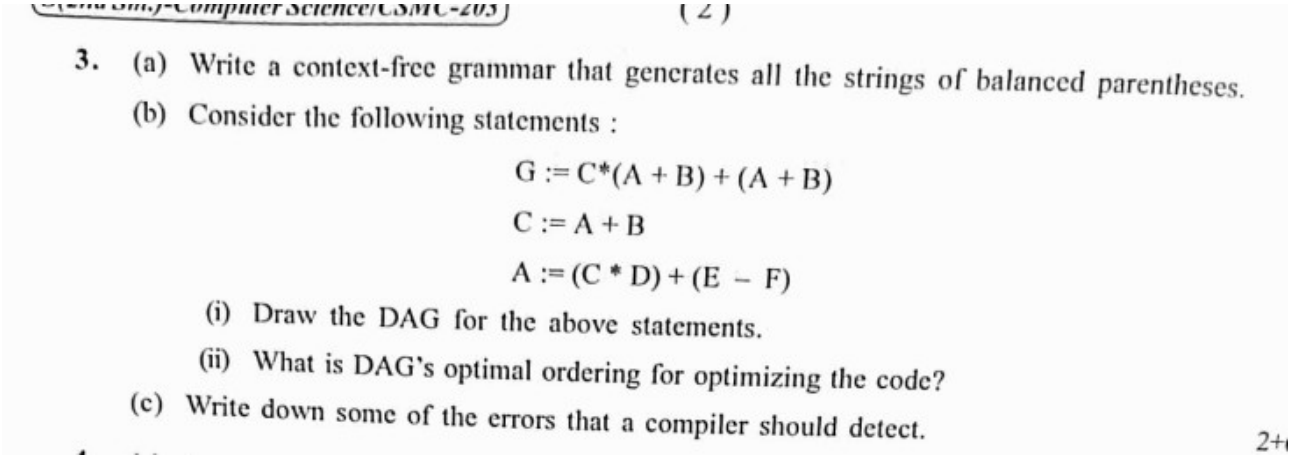
- Use two equal-sized buffers `B1` and `B2` to hold the input. The scanner maintains two pointers: `lexemeBegin` and `forward`. When `forward` reaches the end of one buffer, the next buffer is reloaded from the file while scanning can continue from the other buffer — this minimizes system calls and enables efficient character retrieval.

Sentinel technique (how it helps):

- Put a special sentinel character (e.g. `EOF_MARKER`) at the end of each buffer instead of checking `forward == bufferEnd` on every character. When the scanner reads the sentinel, it knows the buffer ended and triggers buffer refill logic.
- **Benefits:**
    - Avoids a boundary check on every character read (speeds up the scanner).
    - Simplifies code: the scanning loop checks for sentinel rather than explicit index comparisons.
    - When sentinel found, the refill routine replaces sentinel with actual data and resumes scanning; if actual file ended, sentinel indicates EOF and the scanner handles end-of-input.

Typical scanner pointers & steps:

- `lexemeBegin` marks where current token scan started.
- `forward` advances reading characters.
- On reaching sentinel: call `reload()` to refill buffer and adjust `forward`; continue scan.



3. (a) Write a context-free grammar that generates all the strings of balanced parentheses.

(b) Consider the following statements :

$$G := C^*(A + B) + (A + B)$$

$$C := A + B$$

$$A := (C * D) + (E - F)$$

(i) Draw the DAG for the above statements.

(ii) What is DAG's optimal ordering for optimizing the code?

(c) Write down some of the errors that a compiler should detect. 2+

Nice — here are short, exam-style answers (total 10 marks: 2 + 6 + 2).

(a) (2 marks) — CFG for balanced parentheses

One compact grammar that generates all balanced parenthesis strings:

S → S S
S → (S)
S → ε

(Explanation: either concatenate two balanced strings, wrap a balanced string in `( . . . )`, or empty string.)

(b) (6 marks)

Given:

```
mathematica
CopyEdit
G := C * (A + B) + (A + B)
C := A + B
A := (C * D) + (E - F)
```

(i) DAG (drawing in ASCII)

Nodes represent computed values; common subexpressions share a node.

```
mathematica
CopyEdit
    A    B
     \  /
     [n1]   <-- n1 = A + B      (used by C and by G)
      |
     (C)  <- C := n1

    E    F
     \  /
     [n2]   <-- n2 = E - F
```

```
C    D
 \  /
 [n3]   <-- n3 = C * D

 [n4]   <-- n4 = n3 + n2    (labelled as A := ...)

 C   n1
  \  /
  [n5]  <-- n5 = C * n1

  [n6]  <-- n6 = n5 + n1   (labelled as G := ...)
```

Notes:

- n1 = A + B is shared (used to define C and also reused in G).
- n2 = E - F.
- n3 = C * D.
- n4 = n3 + n2 → assigned to A (recomputes A).
- n5 = C * n1.
- n6 = n5 + n1 → assigned to G.

(If drawn as a graph, edges go from operand nodes to operator nodes as shown.)

(ii) DAG's optimal ordering / code schedule (minimize recomputation)

Compute each DAG node once, reuse results. One valid (optimal) ordering with temporaries:

1. t1 = A + B // n1
2. C = t1 // C := A+B
3. t2 = E - F // n2
4. t3 = C * D // n3 (uses C which is t1)
5. A = t3 + t2 // n4 (redefines A)
6. t4 = C * t1 // n5
7. G = t4 + t1 // n6

Comments:

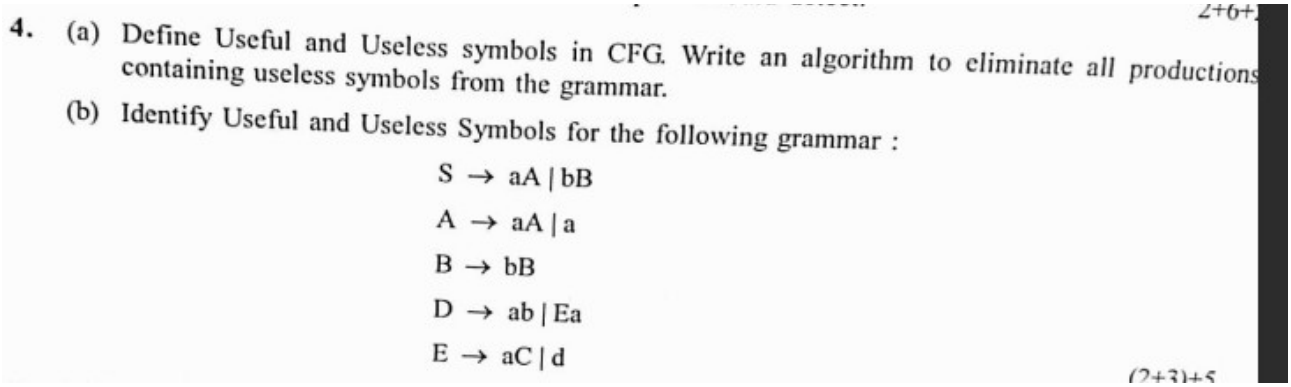
- `A+B` (t1) is computed once and reused for `C` and for `G` (common subexpression elimination).
- All operator nodes are computed once (no duplicate work).
- This ordering respects the given assignment sequence (compute C before using it in A), and minimizes recomputation.

(c) (2 marks) — Some errors a compiler should detect

Give brief list (any two or more of these — each is a typical compiler-detectable error):

- **Lexical errors** (invalid token/character).
- **Syntax errors** (grammar violations; e.g., missing `;`, unmatched parentheses).
- **Semantic / type errors** (type mismatch, e.g., adding `int` to `struct` without operator).
- **Undefined identifier / undeclared variable** (use of a variable not declared).
- **Redeclaration / duplicate definition** errors.
- **Scope errors** (accessing a variable out of its scope).
- **Constant assignment errors** (attempt to assign to a constant).

(Any two of the above would normally earn the 2-mark allocation.)



4. (a) Define Useful and Useless symbols in CFG. Write an algorithm to eliminate all productions containing useless symbols from the grammar.

(b) Identify Useful and Useless Symbols for the following grammar :

$S \rightarrow aA \mid bB$

$A \rightarrow aA \mid a$

$B \rightarrow bB$

$D \rightarrow ab \mid Ea$

$E \rightarrow aC \mid d$

(a) (2 + 3 marks)

Definition (2 marks)

- **Useful symbol (nonterminal)**: a nonterminal that is both (i) *generating* — it can derive some string of terminals — and (ii) *reachable* — it can be reached from the start symbol.
- **Useless symbol**: any nonterminal that is *not* useful (i.e., either it cannot generate a terminal string or it is not reachable from the start symbol).

Algorithm to eliminate productions with useless symbols (3 marks)
Two-phase algorithm (standard):

1. **Remove non-generating symbols**
    - Initialize `G = { }`.
    - Repeat: for each production `X → α` if every symbol in `α` is either a terminal or already in `G`, add `X` to `G`.
    - Iterate until `G` stops growing.

- Remove all productions whose LHS is not in G, and remove any production that has a RHS containing a nonterminal not in G.

2. **Remove non-reachable symbols**
    - Initialize $R = \{S\}$ (start symbol).
    - Repeat: for each production $A \to \alpha$ with $A \in R$, add every nonterminal in α to R.
    - Iterate until R stops growing.
    - Remove all productions whose LHS is not in R.
3. **Result**: remaining nonterminals = $G \cap R$ (the useful symbols), and the remaining productions contain only useful symbols.

(Complexity: each phase is iterative over productions — polynomial in grammar size.)

(b) (5 marks) — Identify Useful and Useless symbols for the grammar

Grammar:

```
rust
CopyEdit
S -> aA | bB
A -> aA | a
B -> bB
D -> ab | Ea
E -> aC | d
```

Step 1 — **Find generating nonterminals** (can derive a terminal string):

- $A \to a \Rightarrow A$ is generating.
- $D \to ab \Rightarrow D$ is generating.
- $E \to d \Rightarrow E$ is generating.
- $S \to aA$ and A is generating $\Rightarrow S$ is generating.
- $B \to bB$ has no base terminal production $\Rightarrow B$ is **not** generating.
- $E \to aC$ uses C, but E already generates via d.
- C has no production $\Rightarrow C$ is **not** generating.

So generating set: $\{S, A, D, E\}$. Non-generating: $\{B, C\}$.

Step 2 — **Remove productions using non-generating symbols**
Remove productions that use B or C on RHS:

- Remove $S \to bB$ (uses B).
- Remove all B productions ($B \to bB$).

- Remove `E -> aC` (uses C).

Remaining productions now:

```
rust
CopyEdit
S -> aA
A -> aA | a
D -> ab | Ea
E -> d
```

Step 3 — **Find reachable nonterminals from start S**:

- S reachable.
- From `S -> aA` ⇒ A reachable.
- From A productions only A (no new nonterminals).
- D and E are **not** reachable from S.

So reachable set: `{S, A}`.

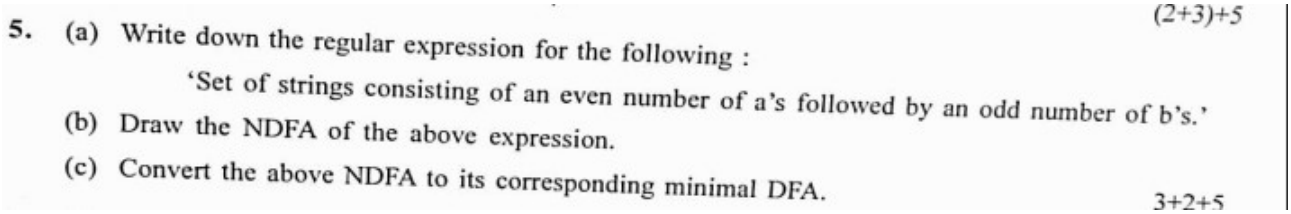
Step 4 — **Useful = generating ∩ reachable**:

- Useful nonterminals: `{S, A}`.
- Useless nonterminals: `{B, C, D, E}` (either non-generating or unreachable).

Final cleaned grammar (only useful symbols and productions):

```
less
CopyEdit
S -> aA
A -> aA | a
```

(That cleaned grammar produces all strings derivable from original grammar starting at S that consist only of useful symbols.)



(2+3)+5

5. (a) Write down the regular expression for the following :

'Set of strings consisting of an even number of a's followed by an odd number of b's.'

(b) Draw the NDFA of the above expression.

(c) Convert the above NDFA to its corresponding minimal DFA. 3+2+5

(a) Regular expression (3 marks)

An even number of a's (including 0) is `(aa)*`.
An odd number of b's is `b(bb)*`.

So the required regular expression is:

(aa)* b(bb)*

(or written as `(aa)* b (bb)*`).

(b) NDFA (2 marks)

I'll give a (small) NDFA that enforces: any number of `a`'s (must be even), then at least one `b` and thereafter `b`'s (odd count).

ASCII state diagram (states: `q0` start, `q1`, `q2` accepting, `q3`, `qd` dead):

```
pgsql
CopyEdit
       a          a
q0 -----> q1 -----> q0      (q0 = even #a seen so far, start)
 |                    \
 |b                    \   b
 v                     v
q2 <---- q3 <---- q2    (q2 = odd #b seen so far, ACCEPTING;
      b      b)         q3 = even #b after entering b-phase)

Any transition on 'a' from q2 or q3 or on 'b' from q1 goes to qd (dead).
qd loops on a,b to itself.
```

Transition table:

- `q0` (start): `a -> q1`, `b -> q2`
- `q1`: `a -> q0`, `b -> qd`
- `q2` (accept): `b -> q3`, `a -> qd`
- `q3`: `b -> q2`, `a -> qd`
- `qd` (dead): `a -> qd`, `b -> qd`

(Although this is deterministic, it is a valid NDFA too — an NFA may be nondet but deterministic NFAs are allowed.)

(c) Minimal DFA (5 marks)

We start from the above automaton (which is already deterministic). Now minimize.

States (original): `q0` (even a's, before b), `q1` (odd a's), `q2` (odd b's) — *accepting*, `q3` (even b's), `qd` (dead).

Initial partition: {Accepting} = `{q2}` and {Non-accepting} = `{q0,q1,q3,qd}`.

Refine by transitions (group states with identical behavior):

- On input `b`: `q0 -> q2` (accept), `q3 -> q2` (accept), `q1 -> qd` (non-accept), `qd -> qd` (non-accept).
  So split non-accepting into `{q0,q3}` and `{q1,qd}`.

- Check {q0,q3} further: on a both go to {q1,qd} (non-accepting group) — they behave identically. So {q0,q3} stays together.
- Check {q1,qd}: on a, q1 -> q0 (which is in {q0,q3}) whereas qd -> qd (stays in {q1,qd}), so they are different → split into {q1} and {qd}.

Final equivalence classes (minimal DFA states):

1. X = {q0, q3} — call it S0 (start): even number of a's and not yet accepting (or even number of b's if in b-phase).
2. q1 — call it S1: odd number of a's (before any valid b).
3. q2 — call it S2 (accepting): odd number of b's (in b-phase).
4. qd — call it SD (dead): invalid sequences.

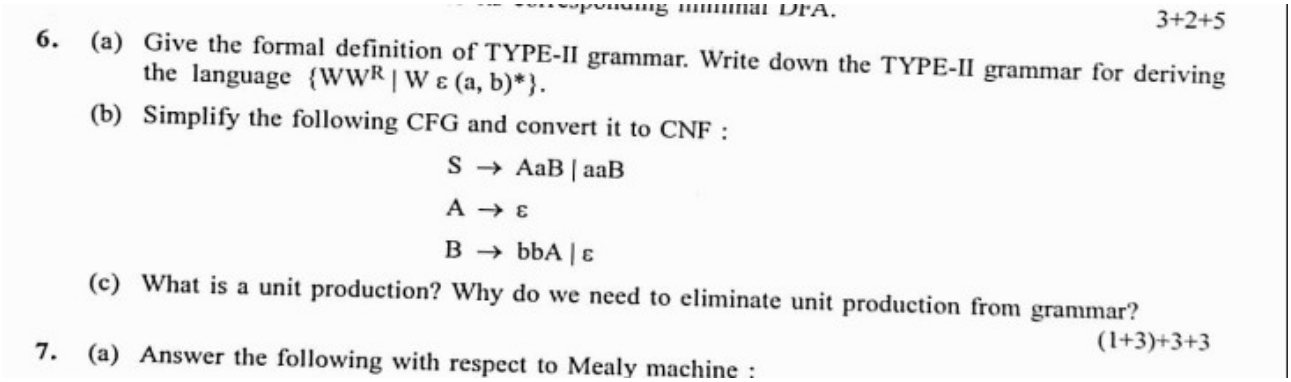
Minimal DFA transitions (compact):

- Start state: S0 ({q0,q3})
  - a -> S1
  - b -> S2
- S1 (q1)
  - a -> S0
  - b -> SD
- S2 (accepting)
  - a -> SD
  - b -> S0
- SD (dead)
  - a -> SD
  - b -> SD

Interpretation / check:

- From start S0 (0 a's, even), reading pairs of a toggles between S0 and S1 — ensures even a's when you leave to b's.
- First b from S0 goes to S2 (odd b's → accept). Further b toggles between S2 and S0 so b-count parity alternates; only S2 is accepting (odd b's).
- Any a after entering the b-phase or a b while in S1 leads to dead state SD.

This 4-state DFA is minimal. (You can draw it as a 4-node directed graph following the table above.)



corresponding minimal DFA. 3+2+5

6. (a) Give the formal definition of TYPE-II grammar. Write down the TYPE-II grammar for deriving the language $\{WW^R \mid W \in (a, b)^*\}$.

(b) Simplify the following CFG and convert it to CNF :

$S \rightarrow AaB \mid aaB$

$A \rightarrow \varepsilon$

$B \rightarrow bbA \mid \varepsilon$

(c) What is a unit production? Why do we need to eliminate unit production from grammar? (1+3)+3+3

7. (a) Answer the following with respect to Mealy machine :

(a) (1 mark) — Formal definition of Type–II grammar

A **Type–II grammar (Context–Free Grammar, CFG)** is a 4-tuple $G=(V,\Sigma,R,S)$ where

- V is a finite set of nonterminals,
- Σ is a finite set of terminals ($V \cap \Sigma = \varnothing$),
- $S \in V$ is the start symbol, and
- R is a finite set of productions of the form $A \rightarrow \alpha$ where $A \in V$ (a single nonterminal) and $\alpha \in (V \cup \Sigma)^*$ (any string of terminals and nonterminals).
  (That single-nonterminal-on-left is the distinguishing property.)

Grammar for $L=\{WWR \mid W \in \{a,b\}^*\}$ (i.e. concatenation of a string and its reverse — even-length palindromes, including ε):

A simple CFG that generates exactly WWR is

```
css
CopyEdit
S → ε
S → a S a
S → b S b
```

Starting from S and repeatedly applying `aSa` or `bSb` builds strings of the form wwR (including ε for w=ε).

(b) (3 marks) — Simplify CFG and convert to CNF

Given grammar (interpreting `S AaB | aaB` as `S -> A a B | aaB`):

```
less
CopyEdit
S -> A a B | aa B
A -> ε
B -> bb A | ε
```

Step 1 — find nullable nonterminals:
A and B are nullable (A → ε, B → ε via rule).

Step 2 — eliminate ε-productions (create alternatives by omitting nullable symbols):

From `S -> A a B` (A and B nullable) we get:

- `A a B` (keep both),
- `a B` (omit A),
- `A a` (omit B),
- `a` (omit both).

From `S -> aa B` (B nullable) we get:

- `aa B`
- `aa` (omit B).

From `B -> bb A` (A nullable) we get:

- `bb A`
- `bb` (omit A).

Remove original ε-productions for A and B. Now we have:

```
less
CopyEdit
S -> A a B | a B | A a | a | aa B | aa
B -> bb A | bb
```

Step 3 — remove non-generating symbols.
A has no non-ε productions now (A only produced ε originally), so A is non-generating → remove productions containing A and the rules with A on LHS. We already created variants that omit A, so after removal we keep:

```
less
CopyEdit
S -> a B | a | aa B | aa
B -> bb
```

(Reason check: with B->bb or ε originally, the derived terminal strings were {a, aa, abb, aabb}; above simplified grammar generates those.)

Step 4 — convert to CNF.
CNF requires productions of the form A→BC or A→a. Introduce nonterminals for terminals appearing in longer RHS:

Let `Ta -> a`, `Tb -> b`.

Replace terminals in length>1 RHS and binarize:

Start with productions:

```
css
CopyEdit
```

```
S -> a
S -> aa         (→ Ta Ta)
S -> a B        (→ Ta B)
S -> aa B       (→ Ta Ta B)
B -> bb         (→ Tb Tb)
```

Binarize the length-3 `Ta Ta B` by introducing Y -> Ta B and S -> Ta Y.

Final CNF grammar (nonterminals: S, B, Y, Ta, Tb):

```
rust
CopyEdit
S  -> a               (terminal)
S  -> Ta Ta           (A -> BC)
S  -> Ta B            (A -> BC)
S  -> Ta Y            (A -> BC)
Y  -> Ta B            (A -> BC)
B  -> Tb Tb           (A -> BC)
Ta -> a               (terminal)
Tb -> b               (terminal)
```

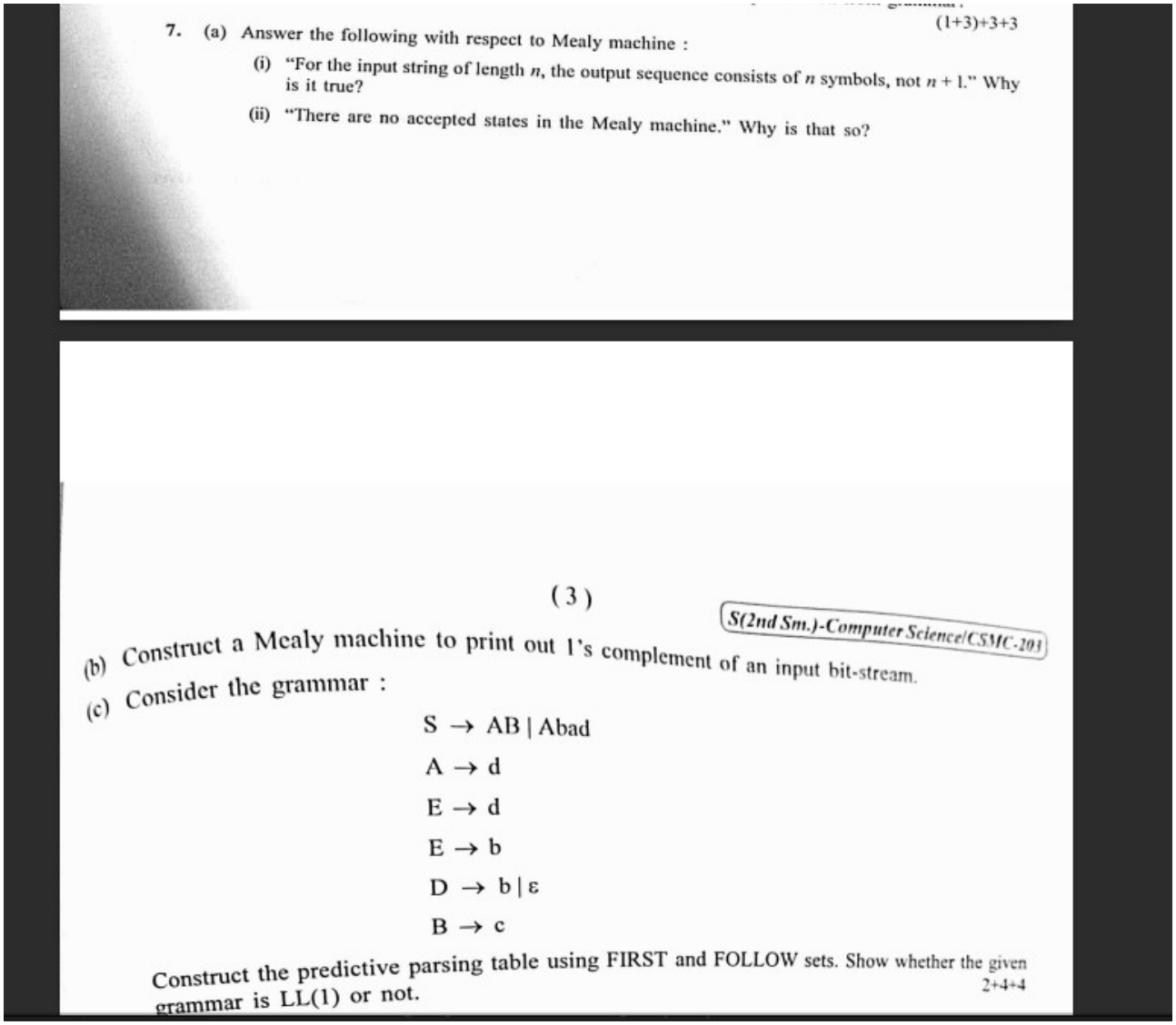
All productions are now either `A -> a` or `A -> BC`, so this is in CNF.

(c) (3 marks) — Unit production: definition and why eliminate

Definition (short):
A **unit production** is a production of the form $A \to B$ where A and B are nonterminals.

Why eliminate unit productions:

- They create chains of single-nonterminal rewrites (e.g. $A \to B \to C \to \ldots$) that complicate grammar analysis and parsing algorithms.
- Elimination produces an equivalent grammar (same language) with fewer trivial derivation steps, which is required for normal forms (like CNF) and for algorithms such as CYK that assume no unit productions.
- Removing them simplifies the grammar, reduces derivation length (helps proofs/complexity), and makes transformations (like CNF conversion) straightforward



7. (a) Answer the following with respect to Mealy machine : (1+3)+3+3

(i) "For the input string of length n, the output sequence consists of n symbols, not $n+1$." Why is it true?

(ii) "There are no accepted states in the Mealy machine." Why is that so?

(3)

S(2nd Sm.)-Computer Science/CSMC-203

(b) Construct a Mealy machine to print out 1's complement of an input bit-stream.

(c) Consider the grammar :

$$S \rightarrow AB \mid Abad$$
$$A \rightarrow d$$
$$E \rightarrow d$$
$$E \rightarrow b$$
$$D \rightarrow b \mid \varepsilon$$
$$B \rightarrow c$$

Construct the predictive parsing table using FIRST and FOLLOW sets. Show whether the given grammar is LL(1) or not. 2+4+4

(a) Mealy machine — conceptual questions

(i) **"For an input string of length *n*, the output sequence consists of *n* symbols, not *n+1*. Why?"**
Because in a **Mealy machine** outputs are produced on **transitions** (i.e. each input symbol causes a transition and an associated output). So one input symbol → one transition → one output, giving exactly *n* outputs for *n* input symbols.
(Contrast: a **Moore** machine attaches outputs to **states**, and you often consider the initial state's output before any input → which can give *n+1* outputs for *n* inputs.)

(ii) **"There are no accepted states in the Mealy machine. Why?"**
A Mealy machine is a **transducer** (maps input sequences → output sequences). Its standard formalism does **not** define "accepting" or "final" states because its purpose is transformation, not language acceptance. (You *can* combine a Mealy machine with an acceptance criterion if you wish — e.g. mark some states as final — but that is extra structure, not part of the basic Mealy definition.)

(b) Mealy machine that outputs the 1's complement of an input bit-stream

We need to output `1` for input `0` and `0` for input `1`. No memory is required (stateless), so a single-state Mealy machine suffices.

State set: `{q0}` (start state)
Input alphabet: `{0,1}`
Output alphabet: `{0,1}`

Transition / output table:

Current state	Input	Next state	Output
q0	0	q0	1
q0	1	q0	0

Diagram (text): single node `q0` with two self-loops:

- `0 / 1` (on input 0 output 1)
- `1 / 0` (on input 1 output 0)

This Mealy machine reads each input bit and immediately emits its complement.

(c) Predictive parsing table for the given grammar — FIRST/FOLLOW and LL(1) check

Given grammar (as written):

```
rust
CopyEdit
S -> AB | Abad
A -> d
E -> d
E -> b
D -> b | ε
B -> c
```

(Interpretation: `Abad` means `A b a d` (terminals `b`,`a`,`d` after A). Note: `E` and `D` appear in the grammar listing but are not used in any production for `S` or others that matter — I'll compute FIRST/FOLLOW for the nonterminals that appear in S and also mention the unused ones.)

Step 1 — FIRST sets

- `FIRST(A)` = { `d` } (because `A -> d`)
- `FIRST(B)` = { `c` } (because `B -> c`)
- For the two productions of `S`:
    - `S -> A B` → `FIRST(A B)` = `FIRST(A)` = { `d` }

- `S -> A b a d` → `FIRST(A b a d)` = `FIRST(A)` = { `d` }
  So `FIRST(S)` = { `d` } (both productions start with `A` whose FIRST is `d`).

(For completeness: `FIRST(E)` = { `d`, `b` } and `FIRST(D)` = { `b`, ε } — but `E` and `D` are unused in productions that derive from `S`.)

Step 2 — FOLLOW sets (start symbol S has $)

- `FOLLOW(S)` = { `$` }.
- From `S -> A B`:
  - `FOLLOW(A)` includes `FIRST(B)` \ {ε} = { `c` }.
  - `FOLLOW(B)` includes `FOLLOW(S)` = { `$` }.
- From `S -> A b a d`:
  - after `A` comes terminal `b`, so `FOLLOW(A)` also includes `b`.

Thus:

- `FOLLOW(A)` = { `b`, `c` }
- `FOLLOW(B)` = { `$` }

Step 3 — Build predictive parsing table entries for each production

We consider terminals `a,b,c,d,$`. For each production `X -> α`, we add it to table[X, t] for every `t ∈ FIRST(α)`; if ε ∈ FIRST(α) also, add for `t ∈ FOLLOW(X)`.

- For `S -> A B`: `FIRST(A B)` = { `d` } ⇒ put `S -> A B` in `M[S, d]`.
- For `S -> A b a d`: `FIRST(A b a d)` = { `d` } ⇒ put `S -> A b a d` in `M[S, d]`.

So **both** productions of `S` map to the same table cell `M[S, d]` → a conflict.

Other nonterminals:

- `A -> d` ⇒ `M[A, d] = A -> d`.
- `B -> c` ⇒ `M[B, c] = B -> c`.
- (If including unused ones: `E -> d` in `M[E,d]`, `E->b` in `M[E,b]`; `D->b` in `M[D,b]` and because `D->ε` we would also use `FOLLOW(D)` to fill `M[D, ...]` — but D is unused so irrelevant to S.)

Predictive parsing table (partial, showing relevant rows)

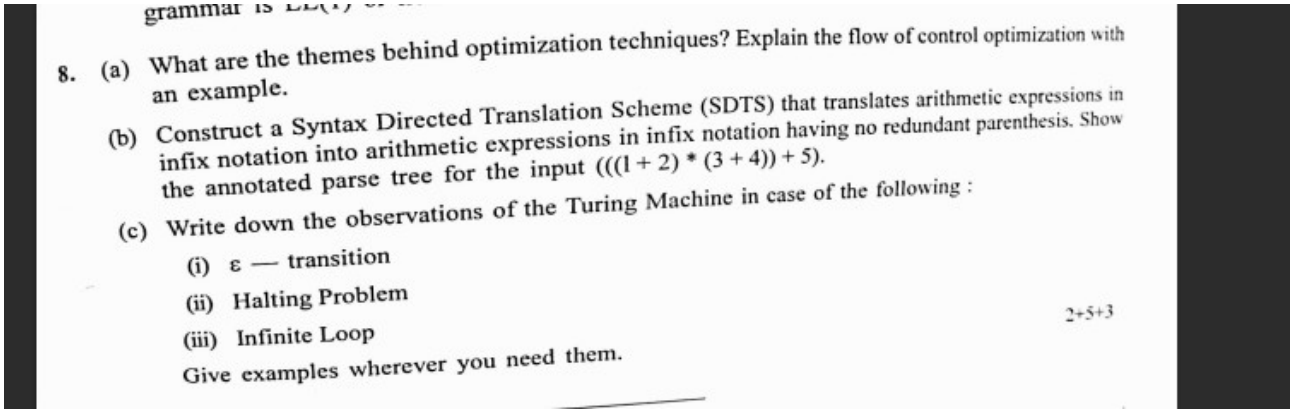
Terminals across top: `a b c d $`

- Row `S`:
  - `M[S,d]` contains `{ S -> A B , S -> A b a d }` ← **conflict**
  - other entries: error

- Row A:
    - `M[A,d] = A -> d`
- Row B:
    - `M[B,c] = B -> c`

Conclusion — Is the grammar LL(1)?

No. It is **not LL(1)** because the two productions for `S` have overlapping FIRST sets (`d` in both) and therefore produce a conflict in the predictive parsing table (`M[S,d]` contains two productions). Since a single lookahead token `d` cannot decide between `S -> A B` and `S -> A b a d`, the grammar fails the LL(1) condition.



grammar is LL(1) or ...

8. (a) What are the themes behind optimization techniques? Explain the flow of control optimization with an example.

   (b) Construct a Syntax Directed Translation Scheme (SDTS) that translates arithmetic expressions in infix notation into arithmetic expressions in infix notation having no redundant parenthesis. Show the annotated parse tree for the input $(((1 + 2) * (3 + 4)) + 5)$.

   (c) Write down the observations of the Turing Machine in case of the following :

   (i) ε — transition

   (ii) Halting Problem

   (iii) Infinite Loop

   2+5+3

   Give examples wherever you need them.

(a) (Themes + flow-of-control optimization) — (marks split implied)

Themes behind optimization techniques

Compiler optimizations are guided by a few recurring themes:

1. **Correctness first** — transformations must preserve the program's semantics (observable behaviour).
2. **Improve performance** — reduce runtime (hot-path speed), reduce memory/IO usage, or reduce energy.
3. **Space vs time trade-offs** — e.g., inline to save branches (faster) but increase code size.
4. **Locality & hot-spot focus** — optimize the frequently executed parts (loops, hot functions).
5. **Machine-independent vs machine-dependent** — do high-level (algorithms, redundancy) and low-level (register allocation, instruction selection) optimizations.
6. **Compile-time vs run-time cost** — some optimizations (e.g. interprocedural analyses) cost more compile time.

7. **Idempotence & composability** — provide safe orderings of optimizations; many are iterative (repeat until fixpoint).
8. **Conservative vs aggressive** — some optimizations risk changing behaviour in presence of undefined behaviour; compilers are conservative unless UB permits change.

Typical kinds of optimizations: constant propagation/folding, dead-code elimination, common-subexpression elimination, loop invariant code motion, loop unrolling, strength reduction, inlining, register allocation, tail-call elimination, branch prediction hints, peephole optimizations.

Flow-of-control (control-flow) optimization — explanation + example

What it is: optimizations that operate on the program's *control-flow graph (CFG)* — nodes = basic blocks, edges = possible control transfers. Goals include removing unreachable code, simplifying branching, merging/splitting blocks, hoisting invariants out of loops, and reducing the number of branches.

Common flow-control optimizations

- **Unreachable-code elimination**: remove blocks with no path from entry.
- **Dead-code elimination**: remove computations whose results are never used.
- **Constant propagation + branch folding**: if a condition becomes constant, replace conditional jump by an unconditional jump and remove the dead branch.
- **Loop optimizations**: loop-invariant code motion (move computations out of loop), loop fusion/fission, strength reduction, induction variable simplification.
- **Tail-call elimination**: convert tail recursion to iteration.

Example (constant propagation → dead-code elimination):

Before optimization:

```
c
CopyEdit
1: x = 2;
2: if (x + 3 > 10)        // (x + 3 > 10) is (5 > 10) => false
3:    y = f();
4: else
5:    z = g();
6: w = h();
```

CFG has two branches. Optimizer does:

1. **Constant propagation**: `x` known to be `2`, so condition `x+3>10` → `5>10` → `false`.
2. **Branch folding**: replace `if (false) then ... else ...` with the `else` branch only (remove branch and the true branch).
3. **Unreachable-code elimination**: remove the `then` block `y = f()` entirely.
4. Possibly further inline/dead-code eliminate if `z` unused.

After optimization:

```
c
CopyEdit
z = g();
w = h();
```

Why this uses control-flow: we updated edges in the CFG (removed edge to the `then` block), removed unreachable nodes, and simplified the graph — fewer branches, fewer instructions on hot paths.

(b) SDTS to remove redundant parentheses (translate infix → minimal-parentheses infix) — (3 marks)

Idea: build attributes for each parse node:

- `node.str` — the printed string for the subtree (with only necessary parentheses).
- `node.prec` — precedence level of the operator at this node (`+, -` = 1, `*, /` = 2, leaf (number) = 3).
- `node.op` — operator at node (`+`, `-`, `*`, `/`, or `""` for leaf).
- Use associativity: `+, -, *, /` are left-associative here. When combining a parent operator `op_p` with a child node `C`, parentheses are required around `C` if:
  - `C.prec < parent.prec` (child binds weaker), OR
  - `C.prec == parent.prec` and parentheses are required by associativity/ operator asymmetry (e.g., for `-` and `/` the **right** child with equal precedence needs parentheses: `a - (b - c)` ≠ `(a - b) - c`).

Grammar (infix):

```
rust
CopyEdit
E -> E + T    { E.str = combine(E1, "+", T); E.prec=1; E.op="+" }
E -> E - T    { ...; E.op="-" }
E -> T        { E.str = T.str; E.prec = T.prec; E.op = T.op }
T -> T * F    { T.str = combine(T1, "*", F); T.prec=2; T.op="*" }
T -> T / F    { ...; T.op="/" }
T -> F        { T.str = F.str; T.prec = F.prec; F.op=F.op }
F -> ( E )    { F.str = E.str ; F.prec = E.prec ; F.op = E.op }   // parentheses
removed if not needed by parent
F -> num      { F.str = num.lexeme; F.prec = 3; F.op = "" }
```

Helper `combine(left, op, right)` (pseudocode):

```
rust
CopyEdit
combine(L, op, R):
  left_s = L.str
  right_s = R.str
  // add parentheses around left if needed
  if (L.prec < prec(op)) left_s = "(" + left_s + ")"
  // add parentheses around right if needed
  if (R.prec < prec(op)) right_s = "(" + right_s + ")"
  // handle non-associative/asymmetric ops for equal precedence on right child:
```

```
  if (R.prec == prec(op) and (op == '-' or op == '/')):
      right_s = "(" + right_s + ")"
  return left_s + " " + op + " " + right_s
```

where `prec("+")=prec("-")=1`, `prec("*")=prec("/")=2`, `prec(num)=3`.

Annotated parse tree for input `(((1 + 2) * (3 + 4)) + 5)`

I'll show the parse tree nodes with their `str`, `prec`, and `op`. (Leaf nodes are numbers.)

1. Leaves:

- `1`: `str="1"`, `prec=3`, `op=""`.
- `2`: `str="2"`, `prec=3`.
- `3`: `str="3"`, `prec=3`.
- `4`: `str="4"`, `prec=3`.
- `5`: `str="5"`, `prec=3`.

2. Node `N1` = `(1 + 2)` via `E -> E + T`:

- combine(1, +, 2):
    - left.prec=3 >=1 → no paren
    - right.prec=3 >=1 → no paren and + is associative so no extra.
- `N1.str = "1 + 2"`, `N1.prec = 1`, `N1.op = +`.

3. Node `N2` = `(3 + 4)`:

- `N2.str = "3 + 4"`, `N2.prec = 1`.

4. Node `N3` = `(N1 * N2)` via `T -> T * F`:

- combine(N1, *, N2):
    - left.prec = 1 < prec(*)=2 → **parenthesize left** → `(1 + 2)`
    - right.prec = 1 < 2 → **parenthesize right** → `(3 + 4)`
- `N3.str = "(1 + 2) * (3 + 4)"`, `N3.prec = 2`, `N3.op = *`.

5. Node `S` = `N3 + 5` via `E -> E + T`:

- combine(N3, +, 5):
    - left.prec = 2 > prec(+)=1 → no parentheses (because * binds tighter than +; `(1+2)*(3+4)` can be written without parentheses around the product when used as left operand of +)
      Note: Do not add parentheses because child precedence `2` > parent `1`.
    - right.prec = 3 > 1 → no paren.

- `S.str = "(1 + 2) * (3 + 4) + 5"`, `S.prec = 1`.

Final printed result:
`(1 + 2) * (3 + 4) + 5`

This matches expectation: the outermost parentheses are removed; parentheses around sums remain because they are operands of `*`.

(c) Observations of the Turing Machine — (i) ε-transition, (ii) Halting Problem, (iii) Infinite Loop (with examples)

(i) "ε → transition"

- In finite automata / NFAs, ε-transition means a transition that consumes *no input symbol* (instantaneous move).
- **Turing machines** are different: their transition function is $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$ — it **always** reads the current tape symbol (which may be the blank symbol ␣) and **then** writes a symbol, moves, and changes state. There is no standard notion of an ε-transition that doesn't read the tape; the equivalent is a transition triggered by reading the blank symbol ␣.
- Also TM models sometimes allow a *stay* move `S` (head does not move) — that is **not** an ε-transition but lets the TM change tape without shifting head.
- **Observation / example:** A TM rule like $\delta(q, ␣) = (p, 1, S)$ will, when it sees blank, write `1` and stay in place — it consumes (reads) the blank but does not move; it's the closest analogue to an "ε-like" action. There is no standard δ that fires without examining the tape symbol.

(ii) Halting Problem

- **Statement:** The Halting Problem asks: given an encoding of a Turing machine `M` and input `w`, will `M` halt on `w`? Alan Turing proved this problem is **undecidable** — there is no algorithm (no TM) that can correctly decide halting for all `(M,w)`.
- **Observation:** You cannot build a TM that always answers "yes/no" for arbitrary `(M,w)` and always halts.
- **Example:** Consider a TM `H` that claims to decide halting. One constructs a diagonal / self-referential machine to derive a contradiction (classic proof). Practically: for some inputs you can prove halting (e.g., machine that immediately enters a halt state), or non-halting (machine that loops moving right forever), but for arbitrary machines you cannot decide in general.

(iii) Infinite loop

- **Observation:** A TM may run forever on some input — i.e., it enters a sequence of configurations that never reaches a halting state. Detecting that a given run will loop infinitely is undecidable in general (equivalent to halting problem).
- **Example (simple looping TM):** Machine `M_loop` with rules:
  - `δ(q0, 0) = (q0, 0, R)`
  - `δ(q0, 1) = (q0, 1, R)`
  - `δ(q0, ⊔) = (q0, ⊔, R)`
    This machine never transitions to a halting state, it just moves right forever → infinite loop for **every** input.
- **Observation (practical):** While infinite loops exist and are sometimes easy to spot for specific machines, there is no general algorithm that, given an arbitrary TM and input, always decides whether it will loop forever.