Solutions

Each answer shows the steps, then the **result**.

2.1

- a **compiler** translates the **whole** program to machine code first (producing an executable)
- an **interpreter** translates and **runs each statement** as it goes (no executable)

2.2

- **lexical analysis → syntax analysis (parsing) → semantic analysis**

2.3

- **tokens** (keywords, identifiers, operators, literals)

2.4

- `3 4 +`

3.1

- any four: it reads a statement, does **lexical** then **syntax** analysis
- checks it / its types
- **executes** the statement's action immediately
- reports any error at once and usually stops
- repeats for each statement – no executable is produced – max 4

3.2

- `<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

3.3

- `3 4 +` for the bracket
- `2 *` last: `3 4 + 2 *`

3.4

- push `5`, push `2`; `-` → pop 2 and 5, $5 - 2 = 3$, push `3`
- push `3` (stack `3, 3`)
- `*` → $3 \times 3 = 9$, result **9**

3.5

- to make the object (machine) code run **faster** and/or use **less memory**, without changing the result it produces

3.6 any three:

- **whitespace** and **comments** are removed
- the code is split into **tokens** / each lexeme is identified
- **identifiers** are added to the **symbol table**
- keywords and operators are replaced by tokens / lexical errors such as invalid identifiers are reported

3.7 any three:

- the tokens are checked against the **grammar / syntax rules** of the language
- a **parse tree** is built from the tokens
- **syntax errors** are found and **reported**
- the output is passed on to code generation / semantic analysis

3.8

(a)

- `8 2 -` and `5 9 +` for the two brackets
- `8 2 - 5 9 + * 7 /`

(b)

- `a b *` is `a * b`; `c d -` is `c - d`
- `a * b + (c - d)`

(c)

- push 4, 3, 6; `*` gives $3 \times 6 = 18$: stack `4, 18`
- `+` gives $4 + 18 = 22$: stack `22`
- push 15, 4; `-` gives $15 - 4 = 11$: stack `22, 11`
- `/` gives $22/11 =$ **2**

(d)

- `4` is not a `<digit>`: a `<digit>` can only be `1`, `2` or `3`