# 01. Pseudocode Syntax & Keywords - Comprehensive Textbook Reference

> **Module Focus**: Algorithmic logic evaluation, keywords ( `READ` , `DISPLAY` , `SET` , `IF-THEN-ELSE` , `FOR` , `WHILE` , `REPEAT-UNTIL` , `CASE` , `SEQUENCE` ), variable scoping, parameter passing mechanisms ( `Pass-by-Value` vs `Pass-by-Reference` ), and execution flow rules for standardized recruitment/ placement exams (CoCubes, AMCAT, eLitmus, GATE).

---

## 1. Input, Output & Assignment Operators

CoCubes and standard recruitment assessment pseudocode uses a simplified syntax layout to evaluate algorithmic comprehension independent of programming language specifics (C, C++, Java, Python).

### 1.1 Input Keywords ( `READ` / `INPUT` )

- **`READ var`** / **`INPUT var`**: Suspends execution to accept data from standard input and stores it in variable `var` .
- **Initialization Trap**: In pseudocode evaluation, referencing a variable before a `READ` or `SET` statement causes an *uninitialized variable error* or undefined behavior.

### 1.2 Output Keywords ( `DISPLAY` / `PRINT` / `OUTPUT` )

- **`DISPLAY var`** / **`PRINT expression`**: Outputs text string literals, variable contents, or results of arithmetic/logical expressions to standard output ( `stdout` ).
- **Formatting & Concatenation**:
- `PRINT "Result: ", x` : Prints literal string followed by value of `x` .
- `PRINT x + y` : Evaluates expression `x + y` before displaying result.

### 1.3 Assignment Operators ( `SET` / `:=` / `=` )

- **`SET var = expression`** or **`var := expression`**: Evaluates the right-hand side `expression` and assigns the result to variable `var` .
- **Assignment vs Equality Comparison**:
- `SET x = 5` / `x := 5` : Assignment statement (mutates state).
- `x == 5` or `x = 5` inside `IF (...)` : Equality comparison (evaluates to `TRUE` or `FALSE` ). Note: In CoCubes pseudocode, single `=` inside condition blocks represents comparison ( `IF x = 5 THEN` ), whereas `SET x = 5` represents assignment.

---

## 2. Conditional Control Structures

### 2.1 Standard `IF-THEN-ELSE` Branching

```
IF (condition_1) THEN
    // Block executed if condition_1 is TRUE
ELSE IF (condition_2) THEN
    // Block executed if condition_1 is FALSE and condition_2 is TRUE
ELSE
    // Block executed if all preceding conditions are FALSE
END IF
```

**Evaluation Rules & Short-Circuit Logic:**

1. **Conditions evaluate sequentially** from top to bottom. Once a condition evaluates to `TRUE`, its block executes and control jumps immediately past `END IF`.
2. **Short-Circuit Evaluation**:
3. `A AND B`: If `A` is `FALSE`, `B` is **not evaluated** (entire expression is `FALSE`).
4. `A OR B`: If `A` is `TRUE`, `B` is **not evaluated** (entire expression is `TRUE`).

### 2.2 Nested Conditionals & Dangling-Else Resolution

In unformatted or complex pseudocode questions, an `ELSE` statement binds to the **innermost unmatched** `IF` statement unless explicitly grouped by `BEGIN...END` or indentation blocks.

### 2.3 Worked Example: Short-Circuiting and Condition Evaluation

**Problem Pseudocode:**

```
INTEGER a = 5, b = 10, c = 0
IF (a > 10 AND (c = b + 1) > 0) THEN
    PRINT "Branch 1"
ELSE IF (b > 5 OR (c = a + 2) > 0) THEN
    PRINT "Branch 2"
ELSE
    PRINT "Branch 3"
END IF
PRINT "c = ", c
```

**Execution Trace:**

1. **Initial State**: `a = 5`, `b = 10`, `c = 0`.
2. **First `IF` Condition**: `(a > 10 AND (c = b + 1) > 0)`
3. Left operand `a > 10` $\implies 5 > 10 \implies$ `FALSE`.
4. Due to **Short-Circuit `AND`**, right operand `(c = b + 1) > 0` is **NOT evaluated**. `c` remains `0`.
5. **Second `ELSE IF` Condition**: `(b > 5 OR (c = a + 2) > 0)`

6. Left operand `b > 5` $\implies 10 > 5 \implies$ **`TRUE`**.
7. Due to **Short-Circuit** **`OR`**, right operand `(c = a + 2) > 0` is **NOT evaluated**. `c` remains `0`.
8. Executes `PRINT "Branch 2"`.
9. **Final Output**:
10. `Branch 2`
11. `c = 0`

---

## 3. Loop Mechanics & Comparison Matrix

### 3.1 Architectural Overview of Loops

#### 1. **`FOR`** Loop (Count-Controlled / Pre-Tested)

- **Syntax**: `text FOR var = start TO end [STEP s] // Loop body END FOR`
- **Execution Rule**: The counter `var` is initialized to `start`. Before each iteration, `var <= end` (for positive `s`) is checked. After each iteration, `var` is incremented by `s` (default `s = 1`).
- **Iteration Count Formula**: $$\text{Iterations} = \max\left(0, \left\lfloor \frac{\text{end} - \text{start}}{\text{step}} \right\rfloor + 1\right)$$

#### 2. **`WHILE`** Loop (Condition-Controlled / Pre-Tested)

- **Syntax**: `text WHILE (condition) // Loop body END WHILE`
- **Execution Rule**: Evaluates `condition` **BEFORE** entering the loop body. If condition is `FALSE` initially, the body **NEVER** executes (0 times).

#### 3. **`REPEAT-UNTIL`** Loop (Condition-Controlled / Post-Tested)

- **Syntax**: `text REPEAT // Loop body UNTIL (condition)`
- **Execution Rule**: Executes loop body **FIRST**, then evaluates `condition` at the bottom.
- **CRITICAL LOGICAL INVERSION**:
- `WHILE (condition)` repeats as long as `condition` is **`TRUE`**.
- `REPEAT ... UNTIL (condition)` repeats as long as `condition` is **`FALSE`**, and **STOPS** the moment `condition` becomes **`TRUE`**!

---

## 3.2 Side-by-Side Comparison Matrix

| Feature / Loop Type | **`FOR`** Loop | **`WHILE`** Loop | **`REPEAT-UNTIL`** Loop |
|---|---|---|---|
| **Control Mechanism** | Counter-controlled | Condition-controlled | Condition-controlled |
| **Testing Point** | **Pre-tested** (Top of loop) | **Pre-tested** (Top of loop) | **Post-tested** (Bottom of loop) |
| **Minimum Iterations** | **0** (if `start > end`) | **0** (if condition is `FALSE`) | **1** (Always executes at least once!) |

| Feature / Loop Type | `FOR` Loop | `WHILE` Loop | `REPEAT-UNTIL` Loop |
|---|---|---|---|
| **Continuation Condition** | Counter $\le$ `end` | Condition evaluates to `TRUE` | Condition evaluates to `FALSE` |
| **Termination Criterion** | Counter $>$ `end` | Condition evaluates to `FALSE` | Condition evaluates to `TRUE` |
| **Typical Use Case** | Known total iteration count | Indefinite looping with pre-check | User input validation, post-processing |

---

## 3.3 Comparative Worked Example: `WHILE` vs `REPEAT-UNTIL`

Consider identical initial conditions ($x = 10$) tested on both loop types with identical condition expressions `(x >= 10)`.

### Code Snippet A (`WHILE` Loop):

```
INTEGER x = 10, count = 0
WHILE (x >= 10)
    count = count + 1
    x = x - 1
END WHILE
PRINT "WHILE -> count: ", count, ", x: ", x
```

### Code Snippet B (`REPEAT-UNTIL` Loop):

```
INTEGER x = 10, count = 0
REPEAT
    count = count + 1
    x = x - 1
UNTIL (x >= 10)
PRINT "REPEAT-UNTIL -> count: ", count, ", x: ", x
```

### Step-by-Step Trace Comparison Table:

| Step | `WHILE` Loop Execution | `REPEAT-UNTIL` Loop Execution |
|---|---|---|
| **Initial** | `x = 10`, `count = 0` | `x = 10`, `count = 0` |
| **Iteration 1** | Check: `(10 >= 10)` $\implies$ `TRUE`.<br>Execute body: `count` = 1, `x` = 9.<br>Next Check: `(9 >= 10)` $\implies$ `FALSE` (Exit loop). | Execute body: `count` = 1, `x` = 9.<br>Check `UNTIL (9 >= 10)` $\implies$ `FALSE`.<br>Since condition is `FALSE`, **CONTINUE LOOPING**! |
| **Iteration 2** | Loop already terminated. | |

| Step | `WHILE` Loop Execution | `REPEAT-UNTIL` Loop Execution |
|---|---|---|
| | | Execute body: `count` = 2, `x` = 8.<br>Check `UNTIL (8 >= 10)` $\implies$ `FALSE`.<br>Continue looping endlessly (Infinite loop)! |
| **Final Result** | `WHILE -> count: 1, x: 9` | **Infinite Loop!** (Never terminates because `x` keeps decreasing) |

> [!KEY TAKEAWAY] Replacing `WHILE (cond)` directly with `REPEAT ... UNTIL (cond)` is a major trap! To achieve identical logic, `REPEAT ... UNTIL` requires the **negated condition**: `REPEAT ... UNTIL (NOT cond)`.

---

## 4. Multi-Way Branching (`CASE` / `SWITCH` Statement)

### 4.1 Formal Syntax & Keywords

```
CASE OF selector_var
    WHEN val_1 DO
        // Statement block 1
    WHEN val_2 DO
        // Statement block 2
    WHEN val_3 TO val_4 DO
        // Statement block for value range [val_3, val_4]
    OTHERS / DEFAULT
        // Default statement block if no matching case is found
END CASE
```

### 4.2 Structural Rules & Key Exam Traps

1. **No Implicit Fallthrough**: Unlike C/C++ `switch` statements without `break`, standard pseudocode `CASE` constructs execute **only the single matching branch** and then automatically exit `END CASE`.
2. **Exclusive Branches**: Once a matching `WHEN` block executes, subsequent `WHEN` clauses are skipped even if they match.
3. `DEFAULT` / `OTHERS`: Executes when `selector_var` matches none of the listed values. If omitted and no match occurs, no block executes.

### 4.3 Worked Example: `CASE` Evaluation

**Problem Pseudocode:**

```
INTEGER marks = 75, grade_code = 0

CASE OF (marks / 10)
```

```
    WHEN 9, 10 DO
        grade_code = 1
    WHEN 7 TO 8 DO
        grade_code = 2
    WHEN 5 TO 6 DO
        grade_code = 3
    DEFAULT
        grade_code = 4
END CASE

PRINT "grade_code = ", grade_code
```

**Step-by-Step Trace:**

1. **Evaluate Selector**: `marks / 10` $\implies 75 / 10 = 7$ (Integer division).
2. **Match Case**:
3. `WHEN 9, 10`: $7 \neq 9$ and $7 \neq 10 \implies$ No match.
4. `WHEN 7 TO 8`: $7 \in [7, 8] \implies$ **MATCH!**
5. **Execute Block**: `grade_code = 2`.
6. **Exit** `CASE`: Skips remaining branches (`WHEN 5 TO 6`, `DEFAULT`).
7. **Output**: `grade_code = 2`.

---

## 5. Subroutines & Parameter Passing Mechanisms

Parameter passing determines how data is transferred between a calling program block and a subprogram (function/procedure).

---

### 5.1 Pass-by-Value Mechanics

- **Definition**: The caller passes a **copy of the value** of the actual argument to the formal parameter of the function.
- **Memory Allocation**: A distinct new memory location is allocated for the formal parameter.
- **Scope Isolation**: Any changes, assignments, or arithmetic operations performed on the formal parameter inside the function **do NOT alter** the caller's original variable in the outer scope.

```
Function UpdateVal(Integer num)
    num = num + 10
    PRINT "Inside UpdateVal: num = ", num
End Function
```

---

### 5.2 Pass-by-Reference Mechanics

- **Definition**: The caller passes the **memory address / reference** (`&` or `REF`) of the actual argument to the function.

- **Memory Allocation**: No new memory is created for the value; the formal parameter acts as an **alias** referencing the exact same memory cell as the caller variable.
- **Direct Mutation**: Any modification to the formal parameter inside the function **immediately mutates** the caller's variable in the outer scope!

```
Function UpdateRef(Integer &num)
    num = num + 10
    PRINT "Inside UpdateRef: num = ", num
End Function
```

---

### 5.3 Deep Dive Comparison Matrix

| Property | Pass-by-Value | Pass-by-Reference |
|---|---|---|
| **Parameter Notation** | `Integer x` | `Integer &x` or `REF Integer x` |
| **Data Transferred** | Copy of variable's value | Memory address of variable |
| **Memory Allocation** | Separate memory cell created | Shares caller's existing memory cell |
| **Caller Impact** | Outer variable remains **unchanged** | Outer variable is **directly mutated** |
| **Safety / Side-Effects** | High safety (no side effects) | Lower safety (side effects on caller state) |

---

### 5.4 Worked Example 1: Pass-by-Value Execution Trace

**Pseudocode Code:**

```
FUNCTION Calculate(Integer x, Integer y)
    x = x * 2
    y = y + 5
    PRINT "Inside Function: x = ", x, ", y = ", y
END FUNCTION

MAIN:
    INTEGER a = 10, b = 20
    Calculate(a, b)
    PRINT "In Main: a = ", a, ", b = ", b
END MAIN
```

**Step-by-Step Execution Trace Table:**

| Line / Scope | Action / Statement | Memory State of `a` | Memory State of `b` | Formal `x` | Formal `y` | Output |
|---|---|---|---|---|---|---|
| | | `10` | `20` | Unallocated | Unallocated | - |

| Line / Scope | Action / Statement | Memory State of `a` | Memory State of `b` | Formal `x` | Formal `y` | Output |
|---|---|---|---|---|---|---|
| **Main (L1)** | Declare `a = 10, b = 20` | | | | | |
| **Call** | `Calculate(a, b)` (Pass-by-value) | `10` | `20` | `10` (Copy) | `20` (Copy) | - |
| **Func (L2)** | `x = x * 2` | `10` | `20` | `20` | `20` | - |
| **Func (L3)** | `y = y + 5` | `10` | `20` | `20` | `25` | - |
| **Func (L4)** | `PRINT` inside function | `10` | `20` | `20` | `25` | `Inside Function: x = 20, y = 25` |
| **Return** | Function completes, `x, y` destroyed | `10` | `20` | Destroyed | Destroyed | - |
| **Main (L6)** | `PRINT` inside Main | **`10`** | **`20`** | - | - | `In Main: a = 10, b = 20` |

**Final Program Output:**

```
Inside Function: x = 20, y = 25
In Main: a = 10, b = 20
```

---

## 5.5 Worked Example 2: Pass-by-Reference Execution Trace

**Pseudocode Code:**

```
FUNCTION ModifyRef(Integer &x, Integer &y)
    x = x + y
    y = x * 2
    PRINT "Inside ModifyRef: x = ", x, ", y = ", y
END FUNCTION

MAIN:
    INTEGER p = 4, q = 6
    ModifyRef(p, q)
    PRINT "In Main: p = ", p, ", q = ", q
END MAIN
```

**Step-by-Step Execution Trace Table:**

| Line / Scope | Action / Statement | `p` (Ref by `x`) | `q` (Ref by `y`) | Formal `x` Alias | Formal `y` Alias | Output |
|---|---|---|---|---|---|---|
| **Main (L1)** | Declare `p = 4, q = 6` | `4` | `6` | Unbound | Unbound | - |
| **Call** | `ModifyRef(p, q)` | `4` | `6` | Binds to `p` | Binds to `q` | - |
| **Func (L2)** | `x = x + y` ($4 + 6 = 10$) | **`10`** | `6` | Alias to `p` (`10`) | Alias to `q` (`6`) | - |
| **Func (L3)** | `y = x * 2` ($10 \times 2 = 20$) | `10` | **`20`** | Alias to `p` (`10`) | Alias to `q` (`20`) | - |
| **Func (L4)** | `PRINT` inside function | `10` | `20` | `10` | `20` | `Inside ModifyRef: x = 10, y = 20` |
| **Return** | Function completes | `10` | `20` | Unbound | Unbound | - |
| **Main (L6)** | `PRINT` inside Main | **`10`** | **`20`** | - | - | `In Main: p = 10, q = 20` |

**Final Program Output:**

```
Inside ModifyRef: x = 10, y = 20
In Main: p = 10, q = 20
```

---

## 5.6 Worked Example 3: Mixed Parameter Passing (Value + Reference)

A favorite topic in CoCubes assessment exams is calling a function where one parameter is passed by value and another is passed by reference, with nested arithmetic.

**Pseudocode Code:**

```
FUNCTION Compute(Integer val, Integer &ref)
    val = val + 5
    ref = ref + val
    PRINT "Inside Compute: val = ", val, ", ref = ", ref
END FUNCTION

MAIN:
    INTEGER x = 3, y = 7
    Compute(x, y)
    PRINT "In Main: x = ", x, ", y = ", y
END MAIN
```

**Step-by-Step Execution Trace Table:**

| Line / Scope | Action / Statement | `x` Memory | `y` Memory | Formal `val` (Value) | Formal `ref` (Ref) | Output |
|---|---|---|---|---|---|---|
| **Main** | Declare `x = 3, y = 7` | `3` | `7` | - | - | - |
| **Call** | `Compute(x, y)` | `3` | `7` | `3` (Copy of `x`) | Binds to `y` | - |
| **Func L2** | `val = val + 5` ($3 + 5 = 8$) | `3` | `7` | **`8`** | Alias to `y` (`7`) | - |
| **Func L3** | `ref = ref + val` ($7 + 8 = 15$) | `3` | **`15`** | `8` | Alias to `y` (`15`) | - |
| **Func L4** | `PRINT` inside function | `3` | `15` | `8` | `15` | `Inside Compute: val = 8, ref = 15` |
| **Return** | Function exits | `3` | `15` | Destroyed | Unbound | - |
| **Main L6** | `PRINT` inside Main | **`3`** | **`15`** | - | - | `In Main: x = 3, y = 15` |

**Final Program Output:**

```
Inside Compute: val = 8, ref = 15
In Main: x = 3, y = 15
```

---

## 6. Summary Cheat Sheet for Exam Solving

1. **Short-Circuit Evaluation**:
2. In `A AND B`, if `A` is `0` / `FALSE`, `B` is **skipped**.
3. In `A OR B`, if `A` is `1` / `TRUE`, `B` is **skipped**.
4. **Loop Iteration Limits**:
5. `FOR i = a TO b STEP s` : Executes $\max(0, \lfloor(b - a)/s\rfloor + 1)$ times.
6. `WHILE (cond)` : Checked **before** entry. Min iterations = **0**.
7. `REPEAT ... UNTIL (cond)` : Checked **after** entry. Min iterations = **1**. Stops when `cond` is **TRUE**.
8. **`CASE` Statements**:
9. Pseudocode `CASE` has **NO fallthrough** unless explicitly written.
10. Range `a TO b` includes both boundaries $a$ and $b$.
11. **Parameter Passing**:
12. **`Pass-by-Value`** **(`Integer x`)**: Main variable value **never changes**.
13. **`Pass-by-Reference`** **(`Integer &x`)**: Main variable **changes directly** with inside function edits.