Name: DUKUNDIMANA Clovis **ID:** 27122

PL/SQL Collections, Records, and GOTO statements

PL/SQL Expense Tracker

1. Project Overview

This PL/SQL script simulates a **Budget Enforcement System**. It processes a list of personal expenses sequentially. As soon as the cumulative total exceeds a predefined limit ($100), the system triggers an emergency exit using a GOTO statement, simulating a "circuit breaker" logic that prevents further processing.

2. Architecture & Concepts

A. PL/SQL Records (TYPE ... IS RECORD)

A **Record** is a composite data type. It allows you to group different pieces of information (fields) into a single logical unit.

- **Analogy:** A "Row" in a database table or a "Struct" in C/C++.
- **In this script:** expense_rec groups the item_name (Text) and cost (Number).
- TYPE expense_rec IS RECORD (
-   item_name VARCHAR2(50),
-   cost   NUMBER
-);

B. PL/SQL Collections (TYPE ... IS TABLE)

A **Collection** is an ordered group of elements, all of the same type.

- **Analogy:** An Array or a List in Java/Python.
- **In this script:** We use a **Nested Table** (expense_list_type). It holds a list of our expense_rec records.
- **Key Operations:**
  - v_my_expenses.EXTEND(3): Nested tables are initially empty. We must explicitly allocate memory for rows.
  - v_my_expenses(i): We access elements using a numeric index (1-based).

C. The GOTO Statement

The GOTO statement performs an **unconditional branch** to a labeled line of code.

- **Syntax:** GOTO label_name; jumps to <<label_name>>.
- **Use Case:** While often avoided in modern programming, GOTO is valid in PL/SQL for specific scenarios:
    - Exiting deeply nested loops.
    - Jumping to a specific error handling block that is not a standard SQL Exception.

3. Logic Trace (Step-by-Step)

Here is exactly what happens when the script runs:

Step	Action	State Variables	Description
1	**Init**	Total = 0	The collection is created and populated with 3 items (Lunch, Taxi, Dinner).
2	**Loop 1**	Total = 20	"Lunch" ($20) is processed. Total is $20. **$20 < $100**, so continue.
3	**Loop 2**	Total = 50	"Taxi" ($30) is processed. Total is $20 + $30 = $50. **$50 < $100**, so continue.
4	**Loop 3**	Total = 130	"Dinner" ($80) is processed. Total is $50 + $80 = $130.
5	**Check**	**TRIGGERED**	The IF condition (130 > 100) evaluates to **TRUE**.
6	**Jump**	**GOTO**	The command GOTO budget_overflow is executed.
7	**Skip**	*Skipped*	The loop is forcibly abandoned. The "Success" message lines are **never executed**.
8	**Label**	*Landing*	Execution resumes at <<budget_overflow>>.
9	**Output**	*Error Msg*	The script prints "FAILED - Budget Exceeded!" and ends.

4. Code Breakdown

The Safety Valve: Why RETURN matters

In the "Success Path" (Section 6 of the code), there is a RETURN; statement.

```
DBMS_OUTPUT.PUT_LINE('Status: SUCCESS');

RETURN;

<<budget_overflow>>
```

Why is it there? PL/SQL executes top-to-bottom. If we finish the loop successfully (e.g., total spent was only $50), the code would naturally continue downwards. Without RETURN, the success message would print, and *then* the code would "fall through" into the budget_overflow section, printing the error message too! RETURN ensures we exit the program before hitting the error label.

5. Summary of Syntax

Feature	**Syntax Used**	**Purpose**
Define Record	TYPE name IS RECORD (field type, ...);	Structure data
Define List	TYPE name IS TABLE OF record_type;	Create a list
Allocate Memory	collection_var.EXTEND(n);	Create empty slots in the list
Looping	FOR i IN 1 .. collection.COUNT LOOP	Iterate over the list
Branching	GOTO label_name;	Jump to specific code
Labeling	<<label_name>>	Mark the destination for GOTO