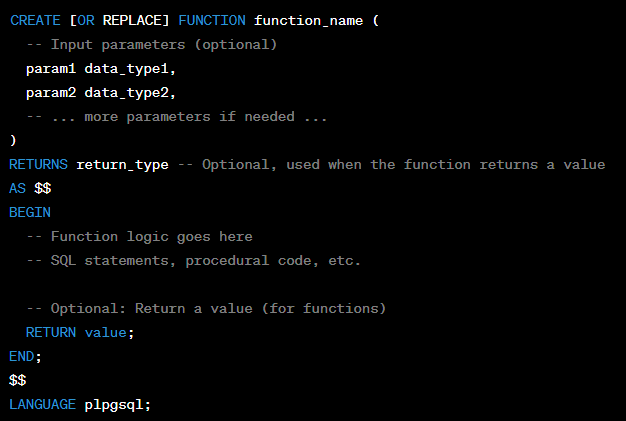
***Structure of Function:***



* CREATE [OR REPLACE] FUNCTION: This statement is used to create a new function with the specified name. The OR REPLACE clause allows to modify the existing function if it already exists with the same name.

***Return Types of a Function:***

The possible return types of a PostgreSQL function are:

1. **Scalar types:** These are the basic type of return type. They include integer, float, string, boolean, date, time, and timestamp.
2. **Composite types**: These are types that consist of multiple scalar types. For example, a record type is a composite type that consists of a set of fields, each of which can be a scalar type.
3. **Table types:** These are types that represent a table. They can be used to return a set of rows from a table.
4. **Void:** This type indicates that the function does not return a value.

***Types:***

***Function category according to implementation:***

1. **Procedural Language Functions:**

- Procedural language functions are functions written in procedural languages like PL/pgSQL - These functions can include more complex logic and flow control using loops, conditions, and variables.

Example of a PL/pgSQL function that updates employee information:

CREATE FUNCTION update\_employee\_info(emp\_id INTEGER, new\_name TEXT)

RETURNS VOID

AS $$

BEGIN

UPDATE employees SET name = new\_name WHERE id = emp\_id;

END;

$$ LANGUAGE plpgsql;

2. **Built-in Functions:** abs(x), sqrt(x), concat(str1, str2), current\_date etc.

***Function category according to return type:***

1. **Scalar functions**: Returns a single value of basic data types, such as an integer, a float, a string, or an object. Example:

CREATE OR REPLACE FUNCTION add\_numbers(a INTEGER, b INTEGER)

RETURNS INTEGER AS

$$

BEGIN

RETURN a + b;

END;

$$ LANGUAGE plpgsql;

1. Aggregate functions: Returns a single value that is calculated from a set of values. For example:

SELECT SUM(salary) FROM employees;

1. Table functions: Table functions return a table. Example:

CREATE OR REPLACE FUNCTION get\_students()

RETURNS TABLE (id INTEGER, name TEXT)

AS $$

BEGIN

RETURN QUERY SELECT id, name FROM students;

END;

$$ LANGUAGE plpgsql;

1. Window functions: Operates on a set of rows in a table. Example:

SELECT name, salary, ROW\_NUMBER() OVER (ORDER BY salary DESC) as rank

FROM employees;

1. Conversion functions: convert between different data types. For example, the TO\_CHAR() function converts a number to a string, and the TO\_DATE() function converts a string to a date.
2. Void Function: Returns nothing. Example:

CREATE OR REPLACE FUNCTION insert\_employee(name TEXT, salary NUMERIC)

RETURNS VOID AS

$$

BEGIN

INSERT INTO employees (name, salary) VALUES (name, salary);

END;

$$

LANGUAGE plpgsql;

1. Composite type: User can define the type to return.

CREATE TYPE person AS (first\_name TEXT, last\_name TEXT);

CREATE OR REPLACE FUNCTION get\_person()

RETURNS person

AS $$

DECLARE

p person;

BEGIN

p.first\_name := 'John';

p.last\_name := 'Doe';

RETURN p;

END;

$$

LANGUAGE plpgsql;

***What type of input parameters PostgreSQL function can take?***

In PostgreSQL functions, you can use various types of input parameters to receive data and values from the caller. PostgreSQL supports a wide range of data types for function parameters, which allows you to handle different kinds of input data. Here are the types of input parameters that PostgreSQL functions can take:

1. Basic Data Types:

- Integer Types: `INTEGER`, `SMALLINT`, `BIGINT`

- Floating-Point Types: `REAL`, `DOUBLE PRECISION`

- Numeric Types: `NUMERIC`, `DECIMAL`

- Character Types: `CHAR`, `VARCHAR`, `TEXT`

- Boolean Type: `BOOLEAN`

2. Date and Time Types:

- `DATE`, `TIME`, `TIMESTAMP`, `INTERVAL`

3. Enumerated Types:

- User-defined enumeration types

4. Composite Types:

- User-defined composite types (a group of fields similar to a struct or record)

5. Array Types:

- Arrays of basic data types, composite types, or other arrays

6. Range Types:

- Continuous ranges of data types (e.g., `INT4RANGE`, `DATERANGE`)

7. Domain Types:

- User-defined domains based on existing data types

8. Special Types:

- `ANYELEMENT`: Accepts any data type

- `ANYARRAY`: Accepts any array type

- `ANYENUM`: Accepts any enumerated type

9. PostgreSQL Object Types:

- `JSON`, `JSONB`, `UUID`, `XML`, `TSVECTOR`, `HSTORE`, etc.

10. Composite Input Types:

- A function can take a single parameter of a composite type that encapsulates multiple values.

***Best practices:***

1. Write the function using CREATE OR REPLACE FUNCTION. That way we can just reload the file to update the function definition.
2. In one window of the editor, create functions, and test those functions in another window.
3. Whenever possible, use the STRICT keyword to specify that the function should return NULL if any of the input arguments are NULL. This helps avoid unexpected behavior. The STRICT keyword is also a best practice to use in functions and procedures that are used to perform operations that should only return a single row.
4. Don't use functions for tasks better suited for other tools: Sometimes, certain tasks are better handled by the application code rather than within the database functions.
5. Volatility: PostgreSQL functions can have different volatility levels, such as stable, volatile, or immutable. Choosing the appropriate volatility level can impact performance. For example, marking a function as "stable" can enable better query optimization opportunities.

***Common Problems:***

1. When replacing an existing function with CREATE OR REPLACE FUNCTION, there are restrictions on changing parameter names. You cannot change the name already assigned to any input parameter (although you can add names to parameters that had none before).
2. Function behavior and features may change between different versions of PostgreSQL.

***Exception Handling:***

We can use EXCEPTION block for exception handling. We can use RAISE to give the error message. For example:

CREATE OR REPLACE FUNCTION safe\_divide(dividend numeric, divisor numeric)

RETURNS numeric AS

$$

DECLARE

result numeric;

BEGIN

IF divisor = 0 THEN

RAISE EXCEPTION 'Divisor cannot be zero.';

END IF;

result := dividend / divisor;

RETURN result;

EXCEPTION

WHEN others THEN

-- Log the error and return NULL to indicate an error condition

RAISE NOTICE 'Error: %', SQLERRM;

RETURN NULL;

END;

$$

LANGUAGE plpgsql;

***Limitations:***

1. Calling is complex from Java class when the function has a void type.

Function – Best practices , common problems , exception handling in function.