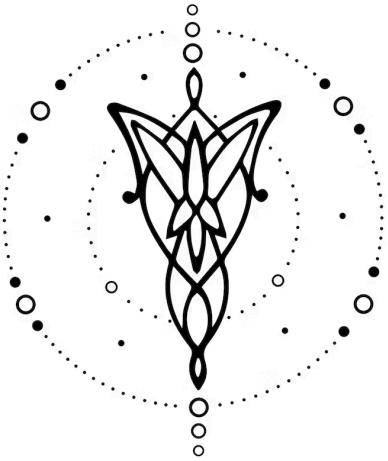


Especificación

Diseño e Implementación de un Lenguaje

v1.0.0

1. Equipo

Nombre	Apellido	Legajo	E-mail
Franco	Bonesi	64.239	fbonesi@itba.edu.ar
Román	Berruti	63.533	rberruti@itba.edu.ar
Matías	Romanato	62.072	maromanato@itba.edu.ar
Luca	Rossi	63.730	lucarossi@itba.edu.ar

2. Repositorio

La solución y su documentación serán versionadas en el repositorio: https://github.com/romamati/TLA_TPE

3. Dominio

Contexto y Motivación

El conglomerado Elven Door busca optimizar la escalabilidad y consistencia de sus desarrollos. Con la diversidad de equipos de ingeniería y la multiplicidad de tecnologías empleadas, es necesario contar con una solución unificada que permita que todas las aplicaciones se desarrollen sobre una misma base. La actualización centralizada del compilador y del DSL facilitará mejoras en rendimiento, seguridad y prácticas de desarrollo. En este contexto, "UMLNator" se plantea como un DSL (Domain Specific Language) y su compilador, cuya función principal es recibir código SQL y, a partir de este, generar el código en PlantUML. Este código podrá ser posteriormente procesado en plataformas como el editor de PlantUML para obtener la representación visual del modelo de datos.

Abstracción del Dominio

El dominio que aborda "UMLNator" se centra en la transformación y visualización de estructuras de datos relacionales. Concretamente, el lenguaje está diseñado para:

- **Entrada:** Aceptar especificaciones en SQL, orientadas a la definición de bases de datos (DDL), que incluyan la creación de tablas, definición de campos, claves primarias y foráneas, y restricciones.
- **Salida:** Generar código en PlantUML que represente de forma visual:
    - **Entidades:** Cada tabla del SQL se traduce en una entidad en PlantUML.
    - **Atributos:** Los campos o columnas se convierten en atributos de las entidades correspondientes.
    - **Relaciones:** Las claves foráneas se mapean a asociaciones entre entidades, donde se pueden expresar las cardinalidades (por ejemplo, uno a uno, uno a muchos, muchos a muchos).

Esta abstracción permite que el diseño de la base de datos se complemente con una representación visual automatizada, facilitando la comprensión, validación y mantenimiento de la estructura de datos.

Especificidad del Dominio

El dominio abordado es **específico** ya que se focaliza en el mapeo entre dos paradigmas:

- **Lenguaje de Definición de Datos (SQL):** Utilizado para describir la estructura lógica de una base de datos relacional.

- **Lenguaje de Visualización (PlantUML):** Utilizado para representar visualmente la arquitectura y las interrelaciones de los datos a partir de la generación automatizada de código.

A diferencia de otros DSLs que podrían abordar aspectos generales del modelado o la simulación, "UMLNator" se especializa exclusivamente en la conversión de especificaciones SQL a código en PlantUML, permitiendo a los desarrolladores transformar de forma directa la definición lógica de una base de datos en una visualización clara y validable.

Objetivos del Dominio

- **Automatización de la Transformación:** Facilitar la conversión directa de código SQL a código en PlantUML, eliminando procesos manuales y minimizando la posibilidad de errores.

- **Mejora de la Developer Experience (DX):** Proveer una herramienta intuitiva que, a partir de un script SQL, genere de forma automática el código visual que representa la estructura de la base de datos, mejorando la comprensión y el diseño.

- **Integración y Escalabilidad:** Permitir que cada equipo del conglomerado trabaje sobre un mismo lenguaje de especificación, facilitando la integración de mejoras y actualizaciones centralizadas en el compilador y en el DSL, y asegurando consistencia en todo el ecosistema de desarrollos.

4. Construcciones

El lenguaje desarrollado debería ofrecer las siguientes construcciones, prestaciones y funcionalidades:

(I). Permitir la creación y definición de una o más tablas mediante CREATE TABLE.
(II). Permitir la especificación de atributos, tipos de datos y restricciones en la creación de las tablas.
(III). Se podrán definir claves foráneas entre las entidades.
(IV). Se podrá indicar que un atributo hace referencia al de otra tabla mediante la palabra clave REFERENCES.
(V). Se ofrecerán tipos de datos estándar tales como INT, VARCHAR, BOOLEAN, FLOAT, etc.
(VI). Se permitirá definir restricciones como PRIMARY KEY, FOREIGN KEY, NOT NULL, UNIQUE, etc.

(VII). Se permitirá definir restricciones como ON DELETE CASCADE, ON DELETE SET NULL, etc.

5. Casos de Prueba

Se proponen los siguientes casos iniciales de prueba de **aceptación**:

(I). Un programa que crea una tabla.
(II). Un programa que crea una tabla con un Primary Key.
(III). Un programa que crea una tabla con una clave compuesta, usando Unique y Not Null.
(IV). Un programa que crea dos tablas, una referenciando a otra mediante un Foreign Key.
(V). Un programa que defina la acción ON DELETE en la referencia de una tabla a otra.
(VI). Un programa que defina la acción ON UPDATE en la referencia de una tabla a otra
(VII). Un programa que crea una tabla con 5 atributos con distintos tipos de datos.
(VIII). Un programa que crea 3 tablas, de las cuales 2 referencian el mismo atributo de una tercera.
(IX). Un programa que crea 3 tablas donde una referencia a 2 tablas.
(X). Un programa que crea 3 tablas, de las cuales 2 referencian atributos distintos de la tercera .

Además, los siguientes casos de prueba de **rechazo**:

(I). Un programa malformado.
(II). Un programa SQL que use sintaxis más allá de la creación de tablas y su manejo de atributos.
(III). Un programa que defina una tabla con más de una Primary Key.
(IV). Un programa que use tipos de datos inexistentes.
(V). Un programa con un Foreign Key que referencie una tabla o atributo inexistente.

6. Ejemplos

Creación de dos tablas en SQL, se pueden añadir atributos con su tipo de dato y propiedades. Hay propiedades como AUTO_INCREMENT o DEFAULT CURRENT_TIMESTAMP que no las toma en cuenta porque son internas al DBMS y no se suelen representar porque involucran funciones definidas en el sistema:

```sql
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    phone VARCHAR(20),
    created_at TIMESTAMP
);
```

```
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    customer_id INT NOT NULL,
    order_date TIMESTAMP,
    total_amount DECIMAL(10,2) NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id) ON DELETE
CASCADE
);
```

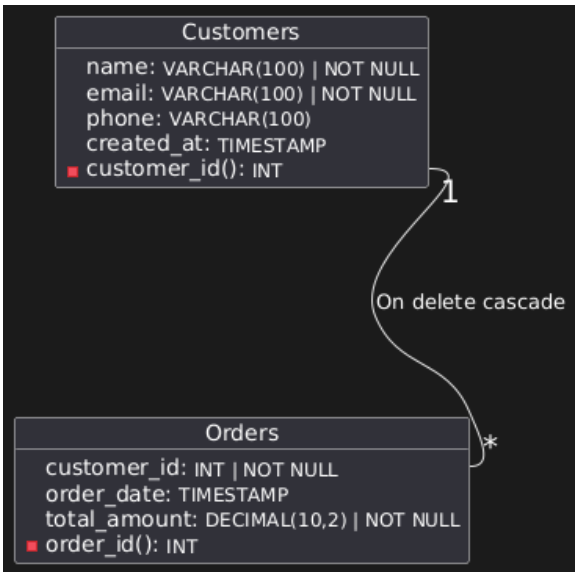
Al pasar por el compilador el código anterior, se debería obtener un nuevo código PlantUML que puede describir visualmente las tablas creadas:

```
@startuml
object Customers {
name: <size:12>VARCHAR(100) | NOT NULL
email: <size:12>VARCHAR(100) | NOT NULL
phone: <size:12>VARCHAR(100)
created_at: <size:12>TIMESTAMP
-customer_id(): <size:12>INT
}

object Orders {
customer_id: <size:12>INT | NOT NULL
order_date: <size:12>TIMESTAMP
total_amount: <size:12>DECIMAL(10,2) | NOT NULL
-order_id(): <size:12>INT
}

Customers::customer_id "<size:20><color:#FFFFFF>1" ---
"<size:20><color:#FFFFFF>*" Orders::customer_id : On delete cascade
@enduml
```

Al pasar ese código PlantUML por la página correspondiente, se debería obtener el diagrama esperado:


Customers
name: VARCHAR(100) | NOT NULL
email: VARCHAR(100) | NOT NULL
phone: VARCHAR(100)
created_at: TIMESTAMP
customer_id(): INT
1
On delete cascade
*
Orders
customer_id: INT | NOT NULL
order_date: TIMESTAMP
total_amount: DECIMAL(10,2) | NOT NULL
order_id(): INT