Procesador del lenguaje SQL en Prolog para DES
SQL language processor for DES

Trabajo de Fin de Grado
Curso 2022–2023

Autor
Long Lin

Director
Fernando Sáenz Pérez

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Procesador del lenguaje SQL en Prolog para DES
SQL language processor for DES

Trabajo de Fin de Grado en Ingeniería Informática

Autor
Long Lin

Director
Fernando Sáenz Pérez

Convocatoria: *Septiembre* 2023

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

27 de agosto de 2023

Agradecimientos

No podría culminar este proyecto sin dedicar un sincero agradecimiento a mi director, el profesor Fernando Sáenz Pérez. Su inestimable ayuda, dedicación y compromiso con el proyecto han sido esenciales para su desarrollo. Agradezco profundamente su prontitud al responder correos, lo que no solo evidencia su profesionalismo, sino también su interés genuino en el avance y mejora continua de mi trabajo. Además, quiero resaltar la generosidad con la que compartió materiales y recursos, que han sido pilares en la consolidación de este Trabajo de Fin de Grado.

Resumen

En este trabajo se aborda el diseño e implementación de un procesador de lenguaje SQL en Prolog con el objetivo de mejorar la eficiencia y rendimiento del sistema de bases de datos deductivas DES. Se identificaron y superaron varias limitaciones del procesador anterior. Para superar estas limitaciones, se propuso un diseño que separa la fase de análisis léxico de la fase sintáctica, ofreciendo así mayor flexibilidad para incorporar nuevas funcionalidades y una estructura mejor definida que facilita la depuración. Si bien DES admite diversos lenguajes de consulta, este proyecto se centra específicamente en SQL. La decisión de enfocarse únicamente en SQL para este proyecto se debe a que fue llevado a cabo por un solo alumno, lo que determinó ciertas limitaciones en cuanto a su alcance.

Palabras clave

Prolog, SQL, Procesador de Lenguaje, Sistema de Bases de Datos Deductivas, DES, Análisis Léxico, Análisis Sintáctico.

Abstract

In this work, the design and implementation of an SQL language processor in Prolog is addressed, with the aim of improving the efficiency and performance of the deductive database system DES. Several limitations of the previous processor were identified and overcome. To address these limitations, a design was proposed that separates the lexical analysis phase from the syntactic phase, thus offering greater flexibility to incorporate new functionalities and a better-defined structure that facilitates debugging. While DES supports various query languages, this project specifically focuses on SQL. The decision to focus solely on SQL for this project is due to it being undertaken by a single student, which set certain boundaries regarding its scope.

Keywords

Prolog, SQL, Language Processor, Deductive Database System, DES, Lexical Analysis, Syntactic Analysis.

Índice

Índice de tablas

Índice de listados

Capítulo 1

Introducción

En la actualidad, los sistemas de bases de datos, siendo componentes esenciales de la informática, se han infiltrado en todos los aspectos de nuestra vida, desde las compras en línea, los registros médicos hasta las redes sociales. Todo esto depende del respaldo de las bases de datos. DES (http://des.sourceforge.net/), como un sistema de bases de datos deductivas. Puede realizar inferencias complejas en datos relacionales y admite múltiples lenguajes de consulta.

1.1. Motivación

Dentro de la temática de procesadores de lenguajes se exploran diversas técnicas y herramientas, entre las más clásicas se encuentran el uso de autómatas finitos, gramáticas libres de contexto, analizadores léxicos y sintácticos, entre otros. Estas metodologías han demostrado ser eficientes en el procesamiento de una amplia variedad de lenguajes de programación (véase la sección 3.1).

Dentro de estas metodologías, las gramáticas de cláusulas definidas (DCG - Definite Clause Grammars) de Prolog han emergido como una herramienta esencial, sobre todo en sistemas como DES. Las DCG son herramientas utilizadas comúnmente en Prolog para el procesamiento de lenguajes. A diferencia de las técnicas tradicionales, las DCG ofrecen una representación declarativa de las gramáticas, permitiendo una definición más intuitiva y concisa (véase el funcionamiento de las DCG en la sección 3.2.3).

Sin embargo, a pesar de la ventaja que supone el uso de DCG en DES, hay margen para mejoras adicionales. Con el crecimiento constante en volumen y complejidad de los datos, la eficiencia de la consulta de bases de datos se ha convertido en un punto focal. Aunque DES tiene una ventaja en el apoyo a varios lenguajes de consulta, su procesador de lenguaje puede enfrentar cuellos de botella en el rendimiento al manejar algunas consultas complejas. Optimizar su procesador de lenguaje es fundamental para aprovechar al máximo el potencial de DES.

1.2. Objetivos

Este proyecto se centra en examinar y mejorar el procesador actual de DES. Nuestros principales objetivos son:

- El objetivo es diseñar y desarrollar un nuevo procesador para SQL en Prolog. Al llevar a cabo el análisis léxico y el análisis sintáctico de manera separada para el lenguaje SQL, no solo se busca un rendimiento excepcional, sino también un código más organizado. Esta estructuración separada facilita cambios futuros, depuración y la incorporación de nuevas funcionalidades, al proporcionar una clara distinción entre las distintas fases del análisis.
- Desarrollar un marco de prueba que permita comparar el rendimiento entre el procesador antiguo y el recién desarrollado. Esta comparativa asegurará que los resultados producidos por el nuevo procesador sean consistentes y equivalentes a los del procesador anterior, validando así su correcta funcionalidad.

1.3. Plan de trabajo

Durante el desarrollo del proyecto, utilizaremos SWI-Prolog, una implementación popular del lenguaje de programación Prolog (véase la sección 2.2). A continuación, se presenta un plan de trabajo detallado:

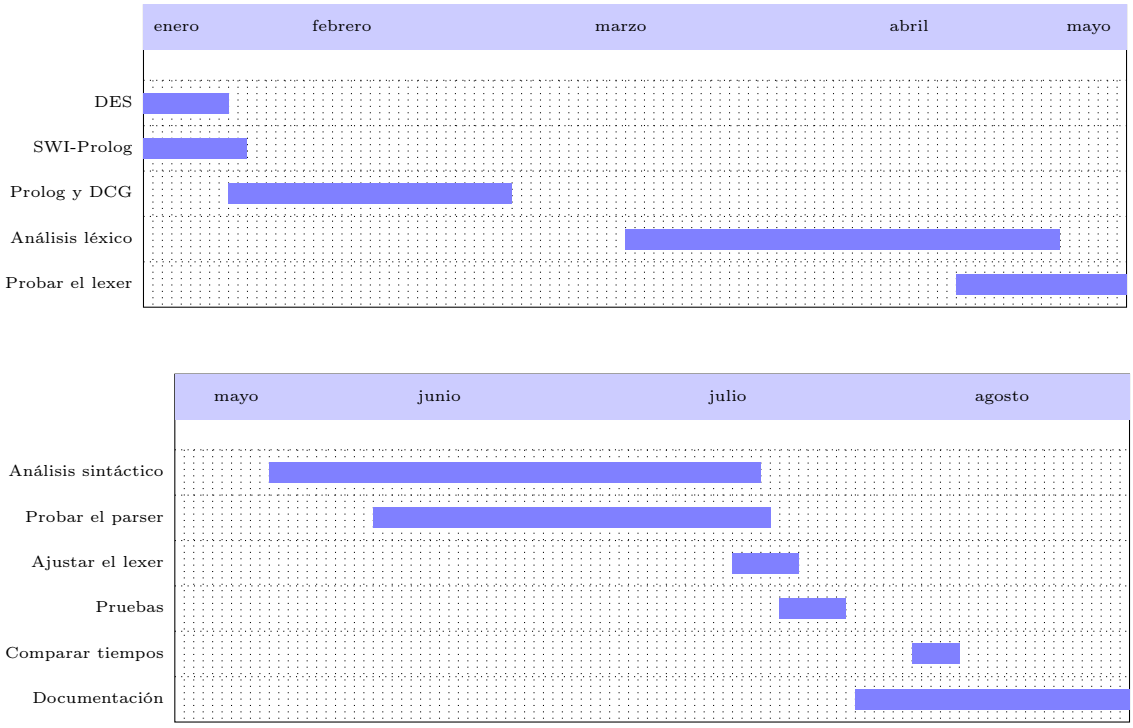
1. **Familiarización con herramientas y técnicas**:
   a) Familiarizarse con DES: Un sistema de bases de datos deductivas que permite inferencias en datos relacionales y admite varios lenguajes de consulta (véase la sección 2.1).
   b) Explorar la plataforma SWI-Prolog: Familiarizarse con el entorno de desarrollo, su sintaxis y características clave (véase la sección 2.2).
   c) Aprender Prolog y DCG: Adquirir habilidades de programación en Prolog y uso de DCG (Definite Clause Grammar) (véase la sección 3.2.3).
2. **Diseño del procesador**:
   a) Diseñar el análisis léxico (lexer): Desarrollar el analizador léxico encargado de tokenizar la entrada.
   b) Diseñar el análisis sintáctico (parser): Crear el analizador sintáctico que construirá el árbol de sintaxis abstracto.
   c) Ajustar el análisis léxico (lexer): Realizar ajustes y mejoras en el análisis léxico basados en el analizador sintáctico desarrollado.
3. **Prueba y gestión de errores**:
   a) Probar el analizador léxico (lexer): Validar la funcionalidad del análisis léxico.
   b) Probar el analizador sintáctico (parser): Validar la funcionalidad del análisis sintáctico.
4. **Pruebas**:
   a) Realizar pruebas generales: Comparar los resultados del nuevo procesador con el sistema actual de DES.

b) Comparar tiempos de ejecución: Evaluar la eficiencia del nuevo procesador en comparación con el procesador actual.

5. **Documentación**:

a) Elaborar la memoria del proyecto.

Diagrama de Gantt



1.4. Estructura de la memoria

En este Trabajo de Fin de Grado (TFG) se presentan los siguientes capítulos que se describen a continuación:

- **Capítulo 2: Estado de la cuestión.** En este capítulo se ofrece una revisión de los trabajos e investigaciones previas relacionados con el DES, la plataforma donde programamos con Prolog (SWI-Prolog), y la importancia de SQL.
- **Capítulo 3: Conceptos previos.** Se introducen conceptos fundamentales para entender el trabajo desarrollado.
  - *Procesadores de lenguajes:* La función de los procesadores de lenguajes y las fases a seguir.
  - *Prolog y programación lógica:* Se ofrece una introducción a Prolog, describiendo de manera breve sus componentes básicos, funcionamiento y su aplicación en el procesamiento de lenguajes.

- **Capítulo 4: Estructura general del proyecto.** Este capítulo detalla la arquitectura del proyecto, describiendo cada uno de los archivos .pl generados y su función específica dentro del análisis de SQL usando Prolog.
- **Capítulo 5: Análisis léxico.** Se aborda el primer nivel de análisis, el léxico, explicando su estructura y las categorías léxicas que se consideran en SQL. Además, se discuten las dificultades encontradas y los resultados obtenidos de este análisis.
- **Capítulo 6: Análisis sintáctico.** En este capítulo se desglosa el análisis sintáctico, explicando cómo se procesan diferentes sentencias y estructuras de SQL. Se discute la identificación de diversos componentes, como tipos de datos, constantes y condiciones, entre otros. También se abordan las dificultades y resultados del análisis sintáctico.
- **Capítulo 7: Gestión de errores.** Se describe cómo se realiza la gestión de errores dentro del proceso de análisis, detallando cómo se detectan y cómo se informan al usuario.
- **Capítulo 8: Pruebas y comparaciones.** En este capítulo se presentan las pruebas realizadas y se compara el tiempo de ejecución entre diferentes enfoques o versiones.
- **Capítulo 9: Conclusiones y trabajo futuro.** Se resume el trabajo realizado, se presentan las conclusiones y se sugieren líneas de trabajo futuras.
- **Apéndices:**
  - **A. Palabras claves en DES:** Enumeración de las palabras clave utilizadas en DES.
    - A.1. Comandos: Listado de comandos disponibles.
    - A.2. Funciones: Las funciones utilizadas.
    - A.3. Operadores: Los operadores (Operadores simbólicos, Operadores de comparación y Operadores textuales).
    - A.4. Símbolos de puntuación: Signos de puntuación.
  - **B. Generación de código Prolog para detección de palabras claves en el analizador léxico (Python):** Explicación del proceso de generación de código para la detección de palabras clave en el analizador léxico utilizando Python.
  - **C. Más ejemplos de análisis léxico:** Conjunto ampliado de ejemplos relacionados con el análisis léxico.
  - **D. Gramática SQL de DES V6.7:** Descripción detallada de la gramática SQL específica para la versión V6.7 de DES.
  - **E. Más ejemplos de gestión de errores en el analizador sintáctico:** Presentación extendida de ejemplos que ilustran el manejo de errores en el análisis sintáctico.

Capítulo 2

Estado de la cuestión

Este capítulo explora herramientas y lenguajes fundamentales en el ámbito de las bases de datos deductivas. Se detallará el sistema educativo DES, la implementación SWI-Prolog y el lenguaje SQL.

2.1. El sistema educativo DES

El sistema educativo DES, es un sistema de bases de datos deductivas que proporciona un amplio rango de lenguajes de consulta sobre un modelo relacional de datos. Estos lenguajes incluyen Datalog, SQL, Álgebra Relacional, Cálculo Relacional de Tuplas (TRC - Tuple Relational Calculus) y Cálculo Relacional de Dominios (DRC - Domain Relational Calculus). (DES, 2023).

DES tiene las siguientes características:

- Es un software gratuito y de código abierto bajo la licencia LGPL.
- Multiplataforma, lo que facilita su portabilidad y adaptabilidad a diferentes sistemas operativos.
- Soporta la deducción en Datalog con estratificación y negación.
- Proporciona soporte completo para SQL, incluyendo características avanzadas como reuniones externas y funciones de agregación.
- Contiene características avanzadas como tipos de datos, restricciones de integridad, razonamiento hipotético y razonamiento difuso.
- Incorpora herramientas de depuración declarativa y trazadores gráficos para facilitar el desarrollo y la depuración.
- Permite la generación de casos de prueba para SQL, asegurando la calidad y robustez del software desarrollado.
- Ofrece una API textual que facilita la integración con otros sistemas y herramientas.
- Está equipado con una completa aritmética y comprobación semántica para SQL.

2.2. SWI-Prolog

SWI-Prolog es una implementación popular y gratuita del lenguaje de programación Prolog, ampliamente utilizada en la academia y la industria. Está orientada especialmente hacia la construcción de aplicaciones del mundo real, como sistemas de bases de datos, interfaces gráficas y aplicaciones web. La plataforma proporciona una extensa biblioteca estándar y paquetes adicionales, lo que la convierte en una elección preferida para el desarrollo en Prolog (SWI, 2023).

En SWI-Prolog, se puede aprovechar el sistema de módulos para estructurar y encapsular el código. Un módulo es una unidad de organización que permite encapsular predicados relacionados, permitiendo una organización lógica del código y previniendo la colisión de nombres entre predicados. Cada módulo en SWI-Prolog tiene su propio espacio de nombres, los predicados con el mismo nombre, pero en diferentes módulos, no interfieren entre sí. Este sistema de módulos es vital cuando se trata de proyectos más grandes y que permite una estructura clara.

SWI-Prolog destaca por su:

- Escalabilidad y eficiencia en el manejo de grandes conjuntos de datos.
- Rico conjunto de bibliotecas integradas y módulos adicionales.
- Fácil integración con otros lenguajes y herramientas, como C, C++, y Java.
- Herramientas de depuración y trazado avanzadas que simplifican el proceso de desarrollo.
- Capacidad para trabajar con archivos de texto, incluido el formato ASCII y otras codificaciones populares como UTF-8.

2.3. SQL

SQL (Structured Query Language), o lenguaje de consulta estructurada es un lenguaje de programación diseñado para gestionar y manipular bases de datos relacionales. Desde su origen en la década de 1970, SQL se ha consolidado como el estándar para interactuar con bases de datos relacionales (Melton y Simon, 2002).

DES, siendo un sistema educativo de bases de datos relacionales, emplea SQL como uno de sus principales lenguajes de consulta. Esta adopción no solo subraya la relevancia de SQL en el ámbito educativo, sino también en la industria de bases de datos en general. La eficiencia en el procesamiento de consultas SQL dentro de DES es esencial. Por ello, la imperante necesidad de contar con un analizador sintáctico optimizado justifica el principal objetivo de este TFG.

Capítulo 3

Conceptos previos

Este capítulo se centra en los conceptos fundamentales que serán la base de nuestro proyecto.

3.1. Procesadores de lenguajes

Los procesadores de lenguaje sirven para traducir o interpretar texto escrito en un lenguaje de programación a un formato que una computadora puede entender y ejecutar.

Un procesador de lenguajes en general puede dividir en varios pasos para llevar al cabo esa tarea:

- **Análisis léxico**: Transforma toma el programa fuente como entrada y lo divide en unidades léxicas o tokens, como una palabra clave, un identificador, un operador, etc. (Aho et al., 2006).
- **Análisis sintáctico**: En esta etapa, el procesador de lenguajes verifica que la secuencia de tokens cumpla con las reglas sintácticas del lenguaje de programación y construye un árbol de sintaxis abstracta (AST) que refleja la estructura gramatical del programa (Scott, 2009).
- **Análisis semántico**: Verifica la coherencia y validez del programa, asegurando que cumpla con las reglas semánticas del lenguaje. Esto puede incluir la verificación de tipos y la resolución de nombres, entre otras tareas.
- **Generación de código**: En esta etapa, el procesador de lenguajes convierte la representación estructurada del programa en un código ejecutable o interpretable.

La ejecución eficiente de estas etapas es crucial para el rendimiento general de los programas escritos en el lenguaje de programación.

En DES se combinan el análisis léxico y el análisis sintáctico en una única pasada, procediendo al análisis semántico inmediatamente después. Sin embargo, en la nueva implementación se hace por separado las dos primeras fases del análisis con el objetivo de optimizar cada una de ellas.

Es importante destacar que, durante la fase de análisis sintáctico se llevan a cabo ciertos chequeos semánticos con el objetivo de optimizar el rendimiento del procesador. Por ejemplo: la verificación de la existencia de una tabla o columna cuando se reconoce la

sentencia `ALTER TABLE` o asegurarse de que el número de columnas especificadas en la tabla coincide con el número de valores proporcionados en `INSERT INTO Table(Columns)`, etc. (véase la tabla 7.7).

3.2. Prolog y programación lógica

Prolog, cuyo nombre proviene de *PROgramming in LOGic*, es un lenguaje esencial en el ámbito de la inteligencia artificial. En Prolog, se establecen hechos y reglas que definen las relaciones entre distintos objetos. Este lenguaje ha encontrado aplicaciones en diversas áreas, como sistemas expertos, patrones de búsqueda y procesamiento del lenguaje natural (Bratko, 2011; O'Keefe, 1990).

En este trabajo utilizaremos Prolog como lenguaje de programación para desarrollar un procesador de lenguaje de SQL para DES.

3.2.1. Componentes básicos de Prolog

3.2.1.1. Términos

Los términos incluyen constantes y variables, los constantes pueden ser átomos o números. Los átomos empiezan con una letra minúscula, o están encerrados entre comillas simples. Por ejemplo, `select`, `'From'`, `3`, `'+'`

Las variables son identificadores que comienzan con una letra mayúscula o un guión bajo (_). Las variables en Prolog son lógicamente universales o existenciales, y Prolog utilizará unificación para intentar encontrar valores que hagan verdaderas las declaraciones lógicas. Por ejemplo, `Char`, `C`, `_T`.

Las estructuras son términos compuestos por otros términos, de la forma nombrePredicado(arg1, arg2, ..., argN). Por ejemplo, `cmd(where/'WHERE')` y `to_lowercase_code (Code, DCode)`.

3.2.1.2. Cláusulas

Las cláusulas en Prolog están basadas en cláusulas de Horn. $C_1 \wedge C_2 \wedge ... \wedge C_N \rightarrow C$, lo cual sería equivalente a escribir: C :- $C_1, C_2, ..., C_N$ (Apt, 1996).

3.2.1.3. Hechos y reglas

Los hechos son declaraciones que se toman como verdaderas, como `non_visible_code`, que define el tabulador y el retorno de carro como códigos invisibles (véase el listado 3.1).

```
non_visible_code(9).    % Tabulator
non_visible_code(13).   % Carriage return
```

Listado 3.1: Caracteres invisibles

Las reglas definen las relaciones o restricciones entre objetos. `is_number_code(Code)` es verdadero si Code corresponde a un dígito decimal en la tabla ASCII; es decir, un número del 0 al 9. Si Code es cualquier otra cosa (una letra, un signo de puntuación, etc.), el predicado es falso.

```
is_number_code(Code) :-
    "0" = [N0],
    "9" = [N9],
    N0 =< Code,
    N9 >= Code.
```

Listado 3.2: Es un dígito decimal

En el listado 3.2 podemos observar que:

- `"0" = [N0]`

  La cadena `"0"` se representa como una lista con un solo elemento. Al unificar `"0"` con `[N0]`, `N0` se vincula al código ASCII de `"0"`, que es 48.

- `"9" = [N9]`

  Similarmente, al unificar `"9"` con `[N9]`, `N9` se vincula al código ASCII de `"9"`, que es 57.

- `N0 =< Code`

  Comprueba que el código dado, `Code`, es mayor o igual al código ASCII de `"0"` (48).

- `N9 >= Code`

  Comprueba que el código dado, `Code`, es menor o igual al código ASCII de `"9"` (57).

3.2.1.4. Programa en Prolog

Un programa de Prolog está compuesto por hechos y reglas. Cada cláusula debe finalizar con un punto (`.`). Para la separación de reglas, se utilizan los signos (`,`) y (`;`), donde la coma indica una conjunción y el punto y coma indica una disyunción. En Prolog se pueden agregar comentarios utilizando el carácter % para comentarios de una línea, y `/* */` para comentarios que abarcan varias líneas.

3.2.2. Funcionamiento de Prolog

- **Unificación**.

  La unificación es una operación fundamental en Prolog que compara y encuentra coincidencias entre dos términos. Esencialmente, la unificación intenta encontrar una sustitución que haga que dos términos lógicos sean idénticos (Sterling y Shapiro, 1994; Bratko, 2011). Por ejemplo, la ecuación $p(X, f(Y)) = p(g(Z), f(Z))$ se puede resolver mediante unificación, que produce la sustitución $\{X/g(Z), Y/Z\}$.

  No todos los pares de términos son unificables, como en el caso de $p(X) = q(X)$ debido a que los nombres de los predicados difieren.

- **Vuelta atrás (Backtracking)**.

  Es otro aspecto central de la programación en Prolog. El backtracking es la técnica mediante la cual Prolog busca en el espacio de soluciones posibles para una consulta dada, retrocediendo y tomando diferentes caminos de búsqueda. Si durante el proceso de resolución de una meta, alguna submeta no es satisfactoria, Prolog realizará backtracking para intentar otra ruta de solución.

- **Corte** (`!`).

  Mediante el uso del predicado de corte (cut, '!'), es posible controlar el proceso de backtracking en Prolog, impidiendo que Prolog realice backtracking más allá del punto del corte. Esto puede aumentar la eficiencia de los programas evitando backtracking innecesario, pero su uso indebido puede complicar la comprensión del programa y provocar que se omitan algunos resultados.

- **Listas en Prolog**.

  En Prolog, una lista es una estructura de datos que puede contener cero o más elementos. Se representa mediante corchetes `[]` y los elementos se separan con comas. Por ejemplo:

  - `[a, 1, c]` es una lista con tres elementos: `a`, `1` y `c`.
  - `[1, 2, 3, 4]` es una lista con cuatro elementos numéricos.
  - `[[a, b], [c, nombre(pepe), e], f, g]` (las listas en Prolog pueden ser anidadas).
  - `[]` representa una lista vacía.

  Para acceder a los elementos de una lista en Prolog, se utiliza la notación `[Cabeza|Resto]`. La `cabeza` se refiere al primer elemento de la lista, mientras que el `resto` es la lista resultante después de eliminar el primer elemento.

  Por ejemplo:

  `[Cabeza | RestoDeLaLista] = [f, r, o, m]`, se asignará `f` a la variable `Cabeza` y `[r, o, m]` a `RestoDeLaLista`.

3.2.3. Prolog para el procesamiento de lenguajes (el uso de DCG)

Las gramáticas definidas por cláusulas (DCG por sus siglas en inglés: Definite Clause Grammar) representan un instrumento poderoso en Prolog, destinado a facilitar el procesamiento del lenguaje.

La notación DCG simplifica la lectura del código, omitiendo información que no es esencial. Esta concisión, en comparación con la notación clásica de Prolog, reduce las posibilidades de cometer errores. Es especialmente útil para la creación de analizadores sintácticos, como los utilizados para interpretar expresiones SQL.

Con la notación DCG es posible declarar gramáticas libres de contexto directamente en Prolog.

La estructura se asemeja a la del Prolog clásico, pero se reemplaza `:-` por `-->`. Las cláusulas que no requieren argumentos ocultos, se encierra entre llaves `{ }`.

Los símbolos terminales se expresan dentro de listas (entre `[` y `]`) y la lista vacía `[]` se usa para indicar que no se consume nada de la entrada. (Pereira y Shieber, 2005; Sterling y Shapiro, 1994).

```
column(attr(R,C,_AS)) -->
  relname(R),
  [punct('.'):_],
  colname(C).
column(attr(_T,C,_AS)) -->
```

```
  colname(C),
  {\+ evaluable_symbol(C)}.
```

Listado 3.3: Reconocimiento de una columna con DCG

El predicado `column` del listado 3.3, se utiliza para detectar el nombre de una columna. En la primera cláusula intenta reconocer el nombre de una relación, seguido por un punto (`.`) y, finalmente, el nombre de la columna.

La segunda cláusula está diseñada para reconocer directamente el nombre de la columna sin necesidad de un nombre de relación previo. Se asegura de que el nombre de la columna no sea un símbolo que pueda ser evaluado (como puede ser una operación o una función) utilizando el predicado `evaluable_symbol`. Esta cláusula es útil para detectar columnas que no están prefijadas por un nombre de relación, lo cual es común en ciertas consultas SQL.

Se omite la necesidad de los argumentos de lista de entrada y salida que serían requeridos en Prolog clásico, como lo podemos ver en el código del listado 3.4.

```
column(attr(R, C, _AS), S0, S) :-
    relname(R, S0, S1),
    S1 = [punct('.'):_ | S2],
    colname(C, S2, S).
column(attr(_T, C, _AS), S0, S) :-
    colname(C, S0, S),
    \+ evaluable_symbol(C).
```

Listado 3.4: Reconocimiento de una columna sin DCG

Capítulo 4

Estructura general del proyecto

El proyecto se divide en múltiples módulos, cada uno con responsabilidades bien definidas (véase la sección 2.2). El diseño modular gestiona por separado los diferentes aspectos del sistema, desde el análisis de las sentencias SQL (análisis léxico y análisis sintáctico) hasta la gestión de errores y pruebas. Al organizar el sistema en módulos separados se garantiza que cada componente puede ser desarrollado, probado y mejorado de manera independiente.

A continuación, se presentan detalladamente cada uno de los módulos que componen el sistema:

1. **lexer.pl**

   Este archivo se encarga del análisis léxico. Puede leer directamente de un archivo, `lex('test/testXXX.sql')`, o analizar una sentencia de SQL proporcionada como una cadena encerrada por comillas dobles, como en `lex("select t from v")`. Además, se proporciona la función `lexer:test` para realizar pruebas. Se aborda en detalle en el capítulo 5.

2. **parser.pl**

   Este es el componente principal que, a partir de los tokens generados por el lexer, produce un árbol sintáctico. Al igual que el lexer, ofrece la capacidad de leer de un archivo o de una sentencia SQL proporcionada entre comillas dobles. Para ejecutar pruebas sobre todos los casos de test se puede utilizar `parser:test`. El proceso completo se detalla en el capítulo 6.

3. **des_data.pl**

   Este archivo contiene predicados relacionados con la base de datos, como `current_db/1`, `current_db/2`, `my_table/3`, `my_attribute/5` y `my_view/9`. Estos predicados se utilizan para obtener información sobre la base de datos, tablas, vistas y atributos. Para facilitar las pruebas, se han incluido estos metadatos directamente en el archivo. Una vez que los resultados de este proyecto se integren en DES, las instrucciones del lenguaje de definición de datos (DDL - Data Definition Language) se encargarán de crear estos metadatos. Es importante destacar que este archivo no requiere integración en DES, dado que los predicados contenidos ya existen en DES.

4. **des.pl y utils.pl**

   Estos archivos contienen predicados esenciales de la versión actual de DES que fueron utilizados en `parser.pl` para facilitar la construcción del analizador sintáctico (parser).

   Al igual que con `des_data.pl`, no es necesario integrar estos módulos en DES. No se hicieron cambios en los predicados de estos módulos, asegurando así una incorporación sin complicaciones en DES.

5. **error_.pl**

   En este archivo se gestionan los errores tanto del analizador léxico (lexer) como del analizador sintáctico (parser). El manejo y gestión de estos errores se explican con mayor detalle en el capítulo 7.

6. **test.pl**

   El archivo test.pl se encarga de la gestión de pruebas. Nos ayudan a asegurar que nuestro código funcione correctamente y a identificar cualquier problema o error que pueda surgir.

   Test.pl está diseñado para probar el lexer y el parser. Para cada módulo, se realizan una serie de pruebas, con nombres de la forma testXXX, donde XXX es un número con ceros a la izquierda. Se puede ejecutar uno en concreto o todas las pruebas de un módulo en orden.

   El proceso de pruebas es bastante sencillo: cuando se llama a el predicado lexer:test o parser:test, se inician las pruebas para el módulo especificado. La función irá buscando y ejecutando cada prueba en el módulo, empezando por test001, luego test002, y así sucesivamente. Si alguna prueba falla, se detiene la ejecución. Si todas las pruebas pasan con éxito, se muestra un mensaje de éxito. Para eso tenemos la función `test(Module, GoalName, Input, Expected)`. Esta función toma un módulo, el nombre de una función a probar, una entrada para esa función y el resultado esperado. Entonces, ejecuta la función con la entrada dada y compara el resultado con el resultado esperado. Si coinciden, la prueba ha pasado. Si no, se considera que la prueba ha fallado y muestra la diferencia entre el resultado y el resultado esperado.

   También es posible activar una prueba específica utilizando los comandos `lexer:testXXX` o `parser:testXXX`, donde `XXX` es un número complementado con ceros a la izquierda.

7. **Carpeta test**

   La carpeta `test` contiene una serie de archivos con extensión .sql diseñados para evaluar el funcionamiento del analizador léxico y del analizador sintáctico. En su conjunto, la carpeta tiene 31 archivos distintos. Algunos de estos archivos están destinados solo para las pruebas del lexer, otros se utilizan para evaluar tanto el analizador léxico como el analizador sintáctico.

Capítulo 5

Análisis léxico

En este capítulo se detallan los procedimientos asociados al análisis léxico del proyecto. Antes de entrarnos en los detalles del análisis léxico, se hace imprescindible presentar las herramientas que fundamentan esta etapa. Específicamente se introduce el concepto de Extended Definite Clause Grammars (EDCG) en Prolog.

5.1. DCG extendidas (EDCG)

Las Extended Definite Clause Grammars (EDCG) son una extensión de Definite Clause Grammars (DCG) de Prolog. La extensión EDCG permite ocultar múltiples argumentos, en lugar de solo dos que se ocultan en las DCG estándar. Esta capacidad de manejar múltiples acumuladores ofrece una flexibilidad adicional, lo que ha resultado ser esencial en aplicaciones avanzadas (Van Roy, 1990).

Las DCG estándar utilizan dos argumentos implícitos para representar la entrada y salida de una lista. Las EDCG, por otro lado, permiten añadir argumentos adicionales, denominados acumuladores para mantener un estado a lo largo de la ejecución. El uso de acumuladores facilita la construcción de un analizador sintáctico más complejo.

La notación de EDCG se utiliza `-->>` en lugar del `-->` usado en DCG estándar.

```
edcg:acc_info(position, X, In, Out, acc_pos(X, In, Out)).
```

Listado 5.1: Declaración de EDCG

A continuación voy a explicar qué significa cada argumento de `edcg:acc_info/5` en el listado 5.1, que es la declaración de EDCG en la fase de análisis léxico del proyecto.

- `position`: Este es el nombre del acumulador. Este acumulador se encarga de mantener un registro de la posición actual en la entrada, es decir, las coordenadas línea y columna en el texto fuente que se está analizando.

  La posición se representa mediante el término `pos(Line, Column)`.

- `X`: Es una variable que representará una operación o acción que queremos realizar en el acumulador. Existen varios predicados relacionados con `position` como `inc_line`, `inc_col`, `add_col`, y `get_pos` que interactúan directamente con el acumulador `position` (véase el listado 5.2).

- `In y Out`: Estas variables representan el valor del acumulador `position` antes y después de la operación, respectivamente.
- `acc_pos(X, In, Out)`: Esta es la acción que se llevará a cabo cuando se encuentre la operación `X` en el texto de entrada.

En resumen, esta definición expresa: cuando se encuentre una operación `X` relacionada con el acumulador `position`, se debe utilizar el predicado `acc_pos/3` para modificar ese acumulador.

```
inc_line -->>
  [add_line(1)]:position.

inc_col -->>
  [add_col(1)]:position.

add_col(N) -->>
  [add_col(N)]:position.

get_pos(Position) -->>
  [get_pos(Position)]:position.

acc_pos(add_col(I), pos(L, C), pos(L, C1)) :-
  C1 is C+I,
  !.
acc_pos(add_line(I), pos(L, _C), pos(L1, 1)) :-
  L1 is L+I,
  !.
acc_pos(get_pos(Position), Position, Position).
```

Listado 5.2: Actualización de posiciones de tokens

Este código de listado 5.2 describe el manejo y actualización de la posición (línea y columna) en un texto.

- `inc_line`: Esta regla se utiliza para incrementar el número de línea en la posición actual por una unidad. El término `[add_line(1)]:position` indica que esta operación modifica el acumulador `position`. Se usa cuando detecta un salto de línea.
- `inc_col`: Similar a `inc_line`, esta regla incrementa el contador de columna por una unidad, indicando que se ha avanzado un carácter en la línea actual.
- `add_col(N)`: Incrementa la columna en N unidades.
- `get_pos(Position)`: Esta regla permite obtener la posición actual en el texto. `[get_pos(Position)]:position` indica que esta operación consulta el valor actual del acumulador `position` y lo unifica con el argumento `Position`.
- `acc_pos(add_col(I), pos(L, C), pos(L, C1))`:
  - *Entrada*: posición actual `pos(L, C)` y el incremento `I`.

- *Salida*: la nueva posición `pos(L, C1)`.
- *Funcionalidad*: Incrementa la columna `C` en `I` unidades para obtener `C1`.

- `acc_pos(add_line(I), pos(L, _C), pos(L1, 1))`:
  - *Entrada*: posición actual `pos(L, _C)` y el incremento `I`.
  - *Salida*: la nueva posición `pos(L1, 1)`.
  - *Funcionalidad*: Incrementa la línea `L` en `I` unidades para obtener `L1` y reinicia el contador de columna a `1`, ya que se ha avanzado a una nueva línea.

- `acc_pos(get_pos(Position), Position, Position)`:
  - *Entrada y Salida*: la posición actual `Position`.
  - *Funcionalidad*: Devuelve la posición actual.

El uso del corte (`!`) en las primeras dos cláusulas de `acc_pos/3` asegura que, una vez que se ha elegido una cláusula particular para manejar una operación, no se retrocederá y se probarán otras cláusulas.

En la siguiente sección se empiezan con la estructura general del análisis léxico.

5.2. Estructura general del analizador léxico

El primer paso que realiza el analizador léxico (lexer) es restablecer todos los errores. A continuación, se invoca al predicado `lex_codes/2` (véase el listado 5.3) con el propósito de tokenizar la lista que se ha leído de la entrada estándar o de un archivo.

```
lex_codes(Codes, Tokens) :-
    token_pos_list(Tokens, pos(1,1), _Pos, Codes, []),
    !.
```

Listado 5.3: Tokenizar (I)

En `token_pos_list` (ver 5.4), el primer paso es `separators_star`, que detecta cero o más separadores (como espacio, salto de línea, final de archivo, etc.). Luego, `get_pos(TokPos)` obtiene la posición actual del token (inicialmente 1,1). A continuación, se invoca a `token(Token)` para obtener el próximo token y `get_pos(NxtPos)` para determinar la posición después de reconocer dicho token. El predicado `separator (Token, NextToken)` decide, basándose en el token actual, si existe un token consecutivo (sin espacios) que deba ser identificado. Si `NextToken` es "no", añade únicamente el token actual a la lista. En caso contrario, añade tanto el token actual como el `NextToken` a la lista. Después, intenta tokenizar de manera recursiva el resto de la entrada.

El último caso de `token_pos_list` gestiona el final de la entrada.

```
token_pos_list(TokenPosList) -->>
    separators_star,
    get_pos(TokPos):position,
    token(Token),
    get_pos(NxtPos):position,
```

```
    separator(Token, NextToken),
    !,
    {(NextToken == no
    -> TokenPosList = [Token:TokPos|RemainingTokenPosList]
    ; TokenPosList = [Token:TokPos,
                NextToken:NxtPos|RemainingTokenPosList])},
    token_pos_list(RemainingTokenPosList).
token_pos_list([]) -->>
    separators_star.
```

Listado 5.4: Tokenizar (II)

Dentro de la regla `token_pos_list`, se usa el acumulador `position` con el predicado `get_pos/1` para obtener la posición de `TokPos` y `NxtPos` de los tokens detectados.

Cada token reconocido se representa en el formato:

`tipoDeToken(Token):pos(Fila,Columna).`

En la siguiente sección se especifican los tipos de tokens para el análisis léxico.

5.3. Categorías léxicas

A continuación se presentan los tokens reconocidos siguiendo el flujo del programa en Prolog. Es importante tener en cuenta el orden, ya que tras el reconocimiento de cada token viene un corte (!).

5.3.1. Números

Reconocen los números positivos. Los números negativos se detectan como un token `op(-)` seguido de un token de tipo número, ya que `'-'` puede ser interpretado como un signo negativo o como una operación de resta.

- **Int:**
  - *Reconoce:* Números enteros positivos.
  - *Devuelve:* El token `int(Integer)` donde `Integer` es el número entero positivo reconocido.
  - *Ejemplos:*
    - `1` ⇒ `int(1)`.
    - `10` ⇒ `int(10)`.
- **Fractional:**
  - *Reconoce:* Números decimales positivos de la forma parteEntera.parteDecimal.
  - *Devuelve:* El token `frac(Integer, Fractional)` donde `Integer` es la parte entera y `Fractional` la parte decimal.
  - *Ejemplos:*

- `3.14` ⇒ `frac(3,14)`.
- `.1` ⇒ `frac(0,1)`.

- **Float:**
  - *Reconoce:* Números positivos en formato exponencial, de la forma:
    - NumeroEnteroPositivo `[e|E]` NumeroEntero.
    - NumeroFraccionalPositivo `[e|E]` NumeroEntero.
  - *Devuelve:* El token `float(Integer, Fractional, Exponent)` donde `Integer` es la parte entera, `Fractional` es la parte decimal y `Exponent` es la parte exponencial.
  - *Ejemplos:*
    - `1e+1` ⇒ `float(1,0,1)`.
    - `1.1e-1` ⇒ `float(1,1,-1)`.

5.3.2. Strings

- *Reconoce:* Las cadenas de texto (strings) delimitadas por comillas simples (`'`). Para escapar una comilla simple dentro de la cadena se puede utilizar \ o duplicando la comilla simple.
- *Devuelve:* El token `str(String)` donde `String` es la cadena reconocida.
- *Ejemplos:*
  - `'2'` ⇒ `str('2')`.
  - `'ab'` ⇒ `str('ab')`.
  - `'O''Connell'` ⇒ `str('O\'Connell')`.

5.3.3. Identificadores delimitados

- *Reconoce:* Identificadores delimitados, i.e., las palabras encerradas por distintos delimitadores y que empiezan por una letra: comillas dobles (`"`), comillas invertidas (`` ` ``), y corchetes (`[ ]`). Como en el caso de las cadenas de texto, permite escapar el delimitador utilizando \ o duplicándolo. Los delimitadores pueden variar según el SGBD (Sistema de Gestión de Base de Datos). Por ejemplo, en MySQL se usan las comillas invertidas (`` ` ``), mientras que en Access se utilizan corchetes (`[ ]`).
- *Devuelve:* El token `delimited_id(Identifier)` donde `Identifier` es el identificador reconocido.
- *Ejemplos:*
  - `"nOmbre"` (Estándar SQL, Oracle, SQL Server) ⇒ `delimited_id(nOmbre)`.
  - `` `nOmbre` `` (MySQL) ⇒ `delimited_id(nOmbre)`.
  - `[nOmbre]` (Access) ⇒ `delimited_id(nOmbre)`.
  - `` `t\`` `` (MySQL) ⇒ `delimited_id('t`')`.
  - `[t+1]` (Access) ⇒ `delimited_id('t+1')`.

5.3.4. Comentarios de SQL

Detecta los comentarios de SQL.

- Una línea
  - *Reconoce:* Los comentarios de una línea en SQL, los que comienzan con dos guiones `--` y se extienden hasta el final de la línea.
  - *Devuelve:* El token `comment(Comment)`, donde `Comment` es el contenido del comentario de esa línea.
  - *Ejemplos:*
    - `-- comentario` ⇒ `comment(comentario)`.
    - `-- select -- Este * es + un_ "comentario" 'de' una linea` ⇒ `comment(select -- Este * es + un_ "comentario" 'de' una linea)`.
- Varias líneas
  - *Reconoce:* Los comentarios de varias líneas en SQL que están encerrados entre `/*` y `*/`. También se admite la anidación de comentarios.
  - *Devuelve:* El token `comment(Comment)`, donde `Comment` es el contenido del comentario.
  - *Ejemplo:*
    - ```
      /*
      Comentario
      /*de*/          ⇒ comment('\nComentario\nde\nvarias lineas\n').
      varias lineas
      */
      ```

5.3.5. Delimitadores

- **Operadores simbólicos**
  - *Reconoce:* Los operadores simbólicos. La lista de operadores simbólicos esta en el apéndice (A.3.1)
  - *Devuelve:* El token `op(Delimiter)` donde `Delimiter` es el operador simbólicos reconocido.
  - *Ejemplos:*
    - `-` ⇒ `op(-)`.
    - `>>` ⇒ `op(>>)`.
- **Operadores de comparación**
  - *Reconoce:* Los operadores de comparación. Los operadores de comparación en SQL de DES incluyen los siguientes:
    `!=` y `<>`: No es igual a.
    `=`: Es igual a.
    `>=`: Es mayor o igual a.
      ```

<=: Es menor o igual a.
>: Es mayor que.
<: Es menor que.

Es importante el orden de detección de estos operadores, ya que se realiza un corte después de reconocer cada token. Si se reconocen > o < antes que >=, <> o <=, se puede obtener un resultado incorrecto, interpretando <> como < y >, que significan menor y mayor respectivamente, en lugar de "no es igual a".

- *Devuelve:* El token `comparison_op(Delimiter)` donde `Delimiter` es el operador de comparación reconocido.
- *Ejemplo:*
  - < ⇒ `comparison_op(<)`.

- **Puntuaciones**
  - *Reconoce:* Las puntuaciones como por ejemplo paréntesis `'('` y `')'`, todas los posibles puntuaciones están en el apéndice A.4.
  - *Devuelve:* El token `punct(Delimiter)` donde `Delimiter` es la puntuación reconocida.
  - *Ejemplos:*
    - `Salto de línea` ⇒ `punct(nl)`.
    - , ⇒ `punct(',')`.

5.3.6. Palabras claves

Las palabras clave son las que tienen funcionalidades específicas en el lenguaje, como, por ejemplo: **create**, **avg**, **and**, etc... Estas palabras se reconocen independientemente de si están en mayúscula o minúscula. Todas ellas retornan `Lc_palabra`, que representa la palabra clave en minúscula para facilitar su reconocimiento en el analizador sintáctico. Además, retornan `Original`, que es la palabra en su formato original. Esto es útil para devolverla en la salida del analizador sintáctico cuando se detecta como un identificador (como nombre de columna, vista, tabla, etc.). Es importante mencionar que los identificadores de SQL en el DES pueden coincidir con una palabra clave de SQL y son sensibles a la distinción entre mayúsculas y minúsculas. Finalmente, este formato es útil para verificar si la primera letra es minúscula en caso de tratarse de un símbolo (utilizado para analizar constantes, funciones y relaciones, o predicados, de Datalog) en la cláusula INTO dentro de sentencias del lenguaje de consulta de datos (DQL - Data Query Language) de SQL.

Estas palabras clave se reconocen usando los patrones que aparecen en el listado 5.5:

```
command('select'/Original) -->>
    lc("select", Original), not_more_char, !, add_col(6).
function('concat'/Original) -->>
    lc("concat", Original), not_more_char, !, add_col(6).
textual_operator('xor'/Original) -->>
    lc("xor", Original),    not_more_char, !, add_col(3).
```

Listado 5.5: Reconocimiendo de las palabras claves

Para cada comando, función y operador textual, llamamos al predicado `lc`, pasando como entrada la palabra en minúscula, y comprobamos si coincide con la palabra clave que estamos analizando. Al mismo tiempo, almacenamos la palabra en su formato original. Luego, `not_more_char` verifica que no haya más letras, números o guiones bajos (_). Después del corte, actualizamos la posición actual del token reconocido utilizando `add_col` sumando al contador de columnas el tamaño de la palabra reconocida. Dado que los patrones para cada comando, función y operador textual son bastante similares, utilicé Python para generar automáticamente este código para cada uno de ellos (véase el apéndice B).

- **Comandos**
  - *Reconoce:* Las palabras claves de SQL, como por ejemplo las más usuales **select**, **from**, **group**, **by**, etc... la lista completa está en el apéndice A.1.
  - *Devuelve:* El token `cmd(Command/Original)` donde `Command` es el comando con letras en minúscula y `Original` se guarda la palabra original.
  - *Ejemplos:*
    - `delete` ⇒ `cmd(delete/delete)`.
    - `drOP` ⇒ `cmd(drop/drOP)`.
    - `CREATE` ⇒ `cmd(create/'CREATE')`.
- **Funciones**
  - *Reconoce:* Las palabras claves que se trata como funciones, como por ejemplo **avg**, **count**, **concat**, **sqrt**, etc... la lista completa está en el apéndice A.2.
  - *Devuelve:* El token `fn(Function/Original)` donde `Function` es el nombre de la función con letras en minúscula y `Original` se guarda la palabra original como lo que lee en la entrada.
  - *Ejemplos:*
    - `sin` ⇒ `fn(sin/sin)`.
    - `MIN` ⇒ `fn(min/'MIN')`.
- **Operadores textuales**
  - *Reconoce:* Los operadores textuales. La lista completa está en el apéndice A.3.3.
  - *Devuelve:* El token `textual_op(Operator/Original)` donde `Operator` es el operador con todas las letras en minúscula y `Original` ídem como comandos y funciones.
  - *Ejemplos:*
    - `or` ⇒ `textual_op(or/or)`.
    - `Not` ⇒ `textual_op(not/'Not')`.

5.3.7. Identificadores

- *Reconoce:* Los restos de palabra que empieza por una letra seguidos de caracteres alfanuméricos o guiones y las palabras que empieza por `$` seguidos del primer caso (letra `+` alfanuméricas o guiones), que en este caso son los identificadores generado por el sistema. Todos ellos son sensibles a mayúsculas y minúsculas.

- *Devuelve:* El token `id(Identifier/Case)` donde `Identifier` es el identificador reconocido y `Case` puede ser `u` o `l`, donde `l` indica que empieza por una letra minúscula y `u` en otros caso (`$` o mayúscula).

  La distinción dada por `Case` tiene relevancia en el contexto de los identificadores de Datalog, tal como se discutió en la sección de palabras claves (5.3.6)

- *Ejemplos:*

  - `Precio` ⇒ `id('Precio'/u)`.
  - `uno_` ⇒ `id(uno_/l)`.
  - `$Valor` ⇒ `id('$Valor'/u)`.

5.3.8. Algunas excepciones en analizador léxico

- **Mod:** Podía ser operador textual(`textual_op`) o funcion(`fn`). Al final lo tratamos como `fn`.

- **Or:** Podía ser operator textual(`textual_op`) o comando(`cmd`). Al final lo tratamos como `textual_op` ya que así crea menos dificultad en el analizador sintáctico (parser).

- **Replace**, **left**, **right** y **float:** Estos cuatros pueden ser comandos(`cmd`) o funciones(`fn`). Al final lo tratamos como `fn` ya que así crea menos dificultad en el analizador sintáctico (parser).

- El **comando** `Savepoint`: Durante el análisis sintáctico, observamos que después de la palabra clave `savepoint` sigue el término `Filename`. Este último puede ser cualquier palabra que no contenga el carácter punto y coma (`;`) o una cadena encerrada entre comillas simples sin que contenga comillas simples en su interior. Al principio mi idea es reconocer `Filename` como otra categoría nueva de token después de reconocer `Identificador` ya que si reconozco antes de `Identificador` ya no podré separar los `Identificador` con los `Filename` porque los `Identificadores` es un subconjunto de `Filename`. Aún así sigueteniendo dos problemas: primero tengo que separar la detección de String en dos tipos: una que no tiene comillas simples escapadas y otra que sí las tiene. La segunda complicación es que detecta token que en la mayoría de contextos resultaban innecesarios, como se mencionó anteriormente, `Filename` solo es relevante cuando se encuentra después del comando `Savepoint`, y estos problemas hacen que el programa sea menos eficiente.

  Para resolver estos problemas, decidí tratar a `Filename` como un delimitador del comando `Savepoint`. De esta manera, `Filename` solo se detecta cuando el token anterior es `cmd(savepoint/_)`.

5.4. Dificultades y desafíos en el analizador léxico

Antes de presentar el resultado final del analizador léxico, mencionaré algunas dificultades encontradas durante este proceso del proyecto.

- **EDCG:** No fue sencillo comprender el concepto de EDCG. Al comienzo del TFG tuve dificultades, pero poco a poco fui comprendiendo el concepto a medida que avanzaba el proyecto.
- **Savepoint filename:** Ya mencioné esta dificultad en la sección anterior 5.3.8.
- **Identificadores:** Al comienzo, no sabía que los identificadores eran sensibles a mayúsculas y minúsculas. En las primeras etapas,, solo devolvía las palabras clave (tales como `cmd`, `fn`, `textual_op`) y el `id(Identifier)` en el formato minúscula, sin conservar el formato original. Fue necesario realizar cambios significativos en el analizador léxico (lexer) y el analizador sintáctico (parser) para conservar las palabras también su formato original.
- **Categorías necesarias:** Al comienzo del proyecto, no estaba seguro de qué categorías léxicas serían necesarias para desarrollar el analizador sintáctico (parser) de SQL. Realicé varias modificaciones para corregir errores y completar las categorías requeridas o eliminar las que no son necesarias.

5.5. Resultados del análisis léxico

En la tabla 5.1 se realiza un repaso mediante ejemplos de todos los tipos de token que pueden aparecer como resultado del análisis léxico.

En la siguiente lista podemos ver todas las categorías léxicas que hemos mencionado en sección 5.3.

1. `int(Integer)`
2. `frac(Integer,Fractional)`
3. `float(Integer,Fractional,Exponent)`
4. `str(String)`
5. `delimited_id(Identifier)`
6. `comment(Comment)`
7. `op(Delimiter)`
8. `comparison_op(Delimiter)`
9. `punct(Delimiter)`
10. `cmd(Command/Original)`
11. `fn(Function/Original)`
12. `textual_op(Operator/Original)`
13. `id(Identifier/Case)`
14. `filename(Filename) (Solo se detecta después de cmd(savepoint/_))`

Entrada	Tokens
-1.2e3	op(-), float(1,2,3)
6*4.5	int(6), op(*), frac(4,5)
'a%b_c'	str('a%b_c')
"a[0]""\""	delimited_id(‘a[0]""’)
--select /* * FROM t */	comment(select), punct(nl), comment(‘ *\nFROM\nt ’)
a != 'b'	id(a/l), comparison_op(‘!=’), str(b)
true AnD C	cmd(true/true), textual_op(and/‘AnD’), id(‘C’/u)
sin(pi)	fn(sin/sin), punct(‘(’), fn(pi/pi), punct(‘)’)
savepoint fileName	cmd(savepoint/savepoint), filename(fileName)

Tabla 5.1: Ejemplos de tokens generados

Por último muestro algunos de los ejemplos para ver los resultados finales de cada token con sus posiciones.

- **Ejemplo I**

```
1 SELECT Id, age
2 FROM user1
3 WHERE age > 18;
```

Listado 5.6: Entrada (lexer) (Ejemplo I)

```
[cmd(select/‘SELECT’):pos(1,1),
id(‘Id’/u):pos(1,8),
punct(‘,’):pos(1,10),
id(age/l):pos(1,12),
punct(nl):pos(1,15),
cmd(from/‘FROM’):pos(2,1),
id(user1/l):pos(2,6),
punct(nl):pos(2,11),
cmd(where/‘WHERE’):pos(3,1),
id(age/l):pos(3,7),
comparison_op(>):pos(3,11),
int(18):pos(3,13),
punct(;):pos(3,15)]
```

Listado 5.7: Salida (lexer) (Ejemplo I)

- **Ejemplo II**

```
1 select * --select -- Este * es  + un_ "comentario" 'de' linea
      unica
2 from tabla
```

Listado 5.8: Entrada (lexer) (Ejemplo II)

```
[cmd(select/select):pos(1,1),
op(*):pos(1,8),
comment('select -- Este * es  + un_ "comentario" \'de\' linea
    unica '):pos(1,10),
punct(nl):pos(1,69),
cmd(from/from):pos(2,1),
id(tabla/l):pos(2,6)]
```

Listado 5.9: Salida (lexer) (Ejemplo II)

- **Ejemplo III**

```
1 alter table t1 add constraint primary key (a);
2
3 alter Table t2 drop constraint not null b;
4
```

Listado 5.10: Entrada (lexer) (Ejemplo III)

```
[cmd(alter/alter):pos(1,1),
cmd((table)/(table)):pos(1,7),
id(t1/l):pos(1,13),
cmd(add/add):pos(1,16),
cmd(constraint/constraint):pos(1,20),
cmd(primary/primary):pos(1,31),
cmd(key/key):pos(1,39),
punct('('):pos(1,43),
id(a/l):pos(1,44),
punct(')'):pos(1,45),
punct(;):pos(1,46),
punct(nl):pos(1,47),
punct(nl):pos(2,1),
cmd(alter/alter):pos(3,1),
cmd((table)/'Table'):pos(3,7),
id(t2/l):pos(3,13),
cmd(drop/drop):pos(3,16),
cmd(constraint/constraint):pos(3,21),
textual_op(not/not):pos(3,32),
cmd(null/null):pos(3,36),
id(b/l):pos(3,41),
punct(;):pos(3,42),
punct(nl):pos(3,43),
punct(nl):pos(4,1)]
```

Listado 5.11: Salida (lexer) (Ejemplo III)

Capítulo 6

Análisis sintáctico

En este capítulo se explican en detalle todos los pasos del análisis sintáctico del proyecto.

6.1. Estructura general del analizador sintáctico

La entrada inicial es procesada por el analizador léxico (lexer), que la convierte en una serie de tokens (para detalles, véase el capítulo 5). Posteriormente, estos tokens son filtrados para eliminar aquellos que no aportan al análisis sintáctico(los saltos de línea y los comentarios). Una vez que la entrada ha sido filtrada, se segmenta en declaraciones individuales, utilizando el punto y coma (`;`) como delimitador. Cada una de estas declaraciones será analizada individualmente en la etapa siguiente.

En el predicado `statement/3`, el fragmento `statement_type(Stmt)` verifica que `Stmt` corresponda a un tipo válido de sentencia. A continuación `call(Stmt, STs)` realiza una invocación dinámica del predicado `Stmt` con el argumento `STs`. Esto implica que se espera que `Stmt` sea uno de los predicados válidos para sentencias, tales como `dqlStmt, dmlStmt, etc`. Este predicado espera opcionalmente un punto y coma (`;`). Si detecta un cierre de paréntesis sin un correspondiente paréntesis de apertura, se genera un error. Si el comienzo no es ningún comienzo de las sentencia valida de SQL de DES se muestra un error indicando que se esperan las siguientes palabras claves.

```
statement(STs) -->
  {statement_type(Stmt)},
  call(Stmt, STs),
  optional_punct(';'),
  ([] ; [punct(')'):Pos], {set_error_with_parameter('Syntax',
      'opening parenthesis ''('' not found before', [], Pos)
    }).
statement(_) -->
  set_error('Syntax', 'valid SQL statement (SELECT, CREATE,
    DELETE, INSERT, UPDATE, DROP, RENAME, ALTER, SHOW,
    DESCRIBE, WITH, ASSUME, COMMIT, ROLLBACK, SAVEPOINT)').

statement_type(dqlStmt).
statement_type(dmlStmt).
```

```
statement_type(ddlStmt).
statement_type(islStmt).
statement_type(tmlStmt).
```

Listado 6.1: Predicado statement en el analizador sintáctico

En DES, el sistema se encarga de aislar cada instrucción con `read_input/3`. Además, en DES se reconocen dos modalidades de entrada: la entrada en una única línea y la entrada en múltiples líneas, identificadas como `multiline off` y `multiline on`, respectivamente. En el modo "multiline off", el punto y coma (`;`) es opcional y sirve para indicar el final de una sentencia SQL. Sin embargo, en el modo "multiline on", el uso del punto y coma (`;`) es obligatorio. En la versión que implementé, todas las entradas son tratadas como si estuvieran en modo "multiline on", es decir, es obligatorio utilizar el punto y coma (`;`) para delimitar las sentencias, salvo si es solo una sentencia.

El resultado final del analizador sintáctico es una lista que representan la estructura de la entrada. Esta lista es útil para las etapas posteriores del procesamiento, como la interpretación o la compilación.

Antes de abordar en detalle las cinco categorías de sentencias clasificadas en DES, primero se examinarán los tipos de datos, las constantes, los condiciones, las expresiones, los identificadores y las restricciones de columnas y tablas en SQL.

6.2. Reconocimiento de tipos de datos en SQL

El sistema reconoce una amplia variedad de tipos de datos SQL, que se pueden clasificar en las siguientes categorías:

1. **Tipos de Caracteres**: Estos incluyen `char(n)`, `varchar(n)`, `string`, y otras variantes. La longitud puede ser fija o variable.
2. **Tipos Numéricos**:
   - **Enteros**: Se identifican por términos como `integer`, `int` y `smallint`.
   - **Punto Flotante y Real**: Representados principalmente por `float`, `real`, y estructuras `numeric` con especificaciones opcionales de precisión.
3. **Tipos de Fecha y Hora**: Abarcan desde simples `date` y `time` hasta combinaciones más completas como `timestamp`.

Cabe destacar que, en algunos casos, hay sinónimos (por ejemplo, `varchar2` y `text` para `varchar`).

6.3. Reconocimiento de constantes en SQL

Los constantes en las sentencias SQL pueden ser de:

- **Número**: Reconoce un valor numérico. `int(), frac() o float()` devolviendo `cte(C,number(N))`, donde `C` es el valor de ese número y `N` es el tipo, en caso de que sea "Float" se pone `float` en otros caso se queda sin unificar.

- **Cadena de caracteres**: Identifica una cadena de texto. `str(String)`, devolviendo `cte(C,string(_S))` donde `C` es la cadena de texto.
- **Default**: Reconoce la palabra clave `default`, devuelve `default`.
- **Valor nulo**: Detecta la palabra clave `null` y generar un identificador para ese valor nulo.
- **Constante de fecha**: Regla que identifica constantes de tipo fecha. Que pueden ser `cte(D,datetime(date))`, `cte(D,datetime(time))` y `cte(D,datetime(datetime))` donde `D` es el valor.

6.4. Reconocimiento de condiciones en SQL

El sistema tiene la capacidad de procesar y analizar condiciones SQL, Estas condiciones son vitales, especialmente en cláusulas `WHERE`, `ON`, `HAVING`, ya que son componentes esenciales para filtrar y seleccionar información específica en una base de datos. Las condiciones pueden variar desde simples comparaciones entre valores hasta condiciones compuestas que involucran múltiples expresiones y operadores.

6.4.1. Condiciones generales (`sql_condition`)

El sistema aborda las condiciones desde un enfoque general que puede descomponerse en varios factores condicionales y operadores lógicos como `AND`, `OR`, entre otros. Características destacadas incluyen:

- Manejo de operadores tanto en formato *prefix* como *infix*.
- Capacidad para agrupar condiciones usando paréntesis, permitiendo estructurar lógicas complejas con diferentes prioridades.
- Definición de operadores lógicos.

6.4.2. Factores de condición (`cond_factor`)

Estas representan las condiciones atómicas que, al combinarse, forman las condiciones generales. Los factores pueden ser:

Valores Booleanos (`true` o `false`), `is_null(R)` o `is_not_null(R)`, `EXISTS`, `BETWEEN`, `NOT BETWEEN`, `IN` y `NOT IN`, `LIKE ... [ESCAPE]` y `NOT LIKE ... [ESCAPE]` o usando operadores básicos como `=`, `<`, `>`, etc.

Adicionalmente, el sistema realiza chequeos de tipos de datos para asegurar coherencia en las operaciones. Por ejemplo, en usos del operador `LIKE`, verifica que las expresiones sean cadenas.

6.5. Análisis de identificadores en SQL

Las consultas SQL a menudo involucran múltiples referencias a tablas, columnas y vistas. Las siguientes reglas describen cómo se procesan estos identificadores en un analizador sintáctico para SQL en DES:

- **untyped_column:** Se encarga de identificar una columna sin especificar un tipo asociado, redirigiendo simplemente al nombre de la columna.
- **p_ren_tablename y ren_tablename:** Estas reglas manejan nombres de tablas, posiblemente con un alias (indicado opcionalmente con `as` seguido de un identificador).
- **column:** Identifica una columna que puede estar prefijada con el nombre de una tabla o relación (por ejemplo, tabla.columna).
- **tablename, viewname, colname, relname:** Estas reglas reconocen identificadores genéricos (`sql_user_identifier`) que representan nombres de tablas, vistas, columnas y relaciones, respectivamente.
- **sql_user_identifier:** Reconoce diversos tipos de identificadores (delimitados por comillas dobles, corchetes, comillas invertidas, identificadores no delimitados, funciones, comandos o operadores textuales). Es importante mencionar que se excluye explícitamente la palabra clave `not` de `textual_op` como identificador.

6.6. Análisis de las expresiones SQL en Prolog

El análisis de expresiones SQL consiste en dividir la expresión SQL en sus componentes básicos.

1. **Expresión SQL básica**: Una expresión SQL puede empezar con un factor seguido de otra expresión SQL. También puede estar encerrada entre paréntesis o puede comenzar con operadores como `op`, `textual_op` o `fn(mod)`.
2. **Expresión SQL recursiva**: Describe cómo las operaciones binarias (infix) se manejan en una expresión SQL. Las operaciones pueden incluir operadores como `op`, `textual_op` o `fn(mod)`.
3. **Factor SQL**: Define cómo se analizan los componentes básicos de una expresión SQL. Puede ser una de las siguientes:
   - Una expresión encerrada entre paréntesis.
   - Una sentencia del lenguaje de consulta de datos (DQL - Data Query Language) (véase la sección 6.10).
   - Funciones agregadas especiales: Algunas funciones son tratadas de manera especial en SQL, por ejemplo, la función `COUNT(*)` y las funciones de agregación que utilizan `DISTINCT`. También verifica que no se combine `DISTINCT` con funciones como `MIN` o `MAX`.
   - Funciones con argumentos. Por ejemplo: `concat/2`, `times/1`, `round/1`, `round/2`, etc.
   - Función `extract` que se utiliza para extraer campos de fecha y hora.
   - Función `cast` para convertir un tipo de dato a otro.
   - Funciones `coalesce`, `greatest`, `least` e `iif` que toman múltiples argumentos.

- Función `case`. Estas son estructuras condicionales en SQL `if-then-else`, distinguiendo entre las formas `CASE-WHEN-THEN` y `CASE-VALUE-WHEN-THEN`.
- Constantes (véase la sección 6.3).
- Funciones sin argumentos. Por ejemplo: `pi`, `e`, `rand`.
- Columnas (véase la sección 6.5).

6.7. Análisis de restricciones de columnas y tablas en SQL

El sistema también se encarga de analizar las restricciones que se pueden aplicar tanto a columnas individuales o a tablas.

- Restricciones de Columna

  Las reglas de `column_constraint` manejan las siguientes restricciones para las columnas:

  Pueden ser **NOT NULL** (`not_nullables`), **NULL** (`true`), **PRIMARY KEY** (`primary_key`), **UNIQUE o CANDIDATE KEY** (`candidate_key`), **REFERENCES** (`foreign_key`), **DEFAULT** (`default`), **CHECK** (`CheckCtr`) y **DETERMINED** (`fd`).

- Restricciones de Tabla

  Las reglas de `table_constraint` manejan las restricciones aplicadas a nivel de tabla:

  Pueden ser **NOT NULL** (`not_nullables`), **PRIMARY KEY** (`primary_key`), **UNIQUE o CANDIDATE KEY** (`candidate_key`), **FOREIGN KEY ... REFERENCES** (`foreign_key`), **CHECK** (`CheckCtr`).

En las siguientes secciones se explica cada uno los cinco tipos de sentencias como fueron clasificadas en DES.

6.8. Sentencias del lenguaje del esquema de información (ISL - Information Schema Language)

Empezamos con las sentencias ISL ya que son los más fáciles de reconocer.
Son utilizada para consultar la información del esquema en DES. (Sáenz-Pérez, 2021a)
Dentro de DES, se implementan cuatros declaraciones principales de ISL. La gramática en sintaxis similar a EBNF se muestra detalladamente en el apéndice D:

1. `SHOW TABLES`
   - *Sintaxis EBNF*: `SHOW TABLES`.
   - *Devuelve*: `show_tables`
   - *Ejemplo*: `show tables` ⇒ `show_tables`

2. `SHOW VIEWS`
   - *Sintaxis EBNF*: `SHOW VIEWS.`
   - *Devuelve*: `show_views`
   - *Ejemplo*: `show Views` ⇒ `show_views`
3. `SHOW DATABASES`
   - *Sintaxis EBNF*: `SHOW DATABASES.`
   - *Devuelve*: `show_databases`
   - *Ejemplo*: `SHOW databases` ⇒ `show_databases`
4. `DESCRIBE`
   - *Sintaxis EBNF*: `DESCRIBE [TableName|ViewName].`
   - *Devuelve*: `describe(Name)` donde el `Name` es el nombre de la tabla o vista.
   - *Ejemplo*: `DESCRIBE t` ⇒ `describe(t)`

```
1 islStmt([show_tables|STs]/STs) -->
2     [cmd(show/_):_],
3     ([cmd(tables/_):_] -> {true} ; set_error('Syntax', '
        TABLES, VIEWS or DATABASES')).
```

Listado 6.2: Reconocimiento de sentencia de SHOW TABLES

El código del listado 6.2 primero reconoce el token `cmd(show/_):_` y después el token `cmd(tables/_):_` Si no reconoce el token `tables`, se actualiza el error sintáctico indicando que se espera la palabra tables, views o databases.

6.9. Sentencias del lenguaje de gestión de transacciones (TML - Transaction Management Language)

Las sentencias TML son esenciales para gestionar transacciones dentro de una base de datos (Sáenz-Pérez, 2021a). Una transacción se puede considerar como una serie de operaciones que actúan como una única unidad de trabajo. Su objetivo principal es garantizar la integridad de la base de datos, incluso si se produce un fallo en alguna de las operaciones.

Dentro de DES se implementan tres declaraciones principales de TML. La gramática en sintaxis similar a EBNF se muestra detalladamente en el apéndice D:

1. `COMMIT`
   - *Sintaxis EBNF*: `COMMIT [WORK].`
   - *Devuelve*: `commit`.
   - *Ejemplo*: `commit work` ⇒ `commit`
2. `ROLLBACK`

- *Sintaxis EBNF*: `ROLLBACK [WORK] [TO SAVEPOINT SavepointName]`.
- *Devuelve*: `rollback([SP])` o simplemente `rollback([])`, donde `SP` es el nombre del fichero.
- *Ejemplo*: `rollback to savepoint` ⇒ `rollback([sp1])`

3. `SAVEPOINT`

- *Sintaxis EBNF*: `SAVEPOINT SavepointName`.
- *Devuelve*: `savepoint([SP])`, donde `SP` es el nombre del fichero con extensión .ddb.
- *Ejemplo*: `SAVEPOINT sp1` ⇒ `savepoint([‘sp1.ddb’])`

A continuación se presenta en el listado 6.3 un fragmento de código que detalla cómo se reconoce la sentencia ROLLBACK TO SAVEPOINT en Prolog:

```
1  % ROLLBACK TO SAVEPOINT
2  tmlStmt([rollback([SP])|STs]/STs) -->
3      [cmd(rollback/_):_],
4      optional_cmd(work),
5      cmd(to/_)                  # ‘TO’,
6      cmd(savepoint/_)           # ‘SAVEPOINT’,
7      filename(SP)               # ‘savepoint name(id without
          semicolon or quoted id without unescaped quotes)’.
8
9  % ROLLBACK without specifying SAVEPOINT
10 tmlStmt([rollback([])|STs]/STs) -->
11     [cmd(rollback/_):_],
12     optional_cmd(work).
13
14 filename(FileName) -->
15   [filename(FileName):_Pos].
```

Listado 6.3: Reconocimiento de sentencia de ROLLBACK TO SAVEPOINT

Este fragmento define dos reglas: la primera trata el caso en que se especifica un `SAVEPOINT`, y la segunda maneja el caso en que se utiliza `ROLLBACK` sin especificar un `SAVEPOINT`.

La primera regla busca, en secuencia, token `cmd(rollback/_):_`, un token opcional `work`, seguido de los tokens `to` y `savepoint`. Finalmente, busca un token que represente el nombre del punto de guardado, denotado por denotado por el token `filename`. Y utilizamos `#` para gestionar los errores, indicando lo último que se espera. Véase el capítulo 7 para el detalle de la gestión de los errores.

La segunda regla, por otro lado, simplemente busca el token `cmd(rollback/_):_` y el token opcional `work`.

6.10. Sentencias del lenguaje de consulta de datos (DQL - Data Query Language)

La sentencia principal de DQL es la consulta SELECT, que se centra en la recuperación de datos de la base de datos basándose en criterios específicos, tales como condiciones, agrupaciones y ordenamientos.

La gramática en sintaxis similar a EBNF para las sentencias DQL se describe detalladamente en el apéndice D.

1. `SELECT`

   - *Sintaxis EBNF*:

     ```
     SELECT [TOP Integer [[ALL|DISTINCT]] SelectExpressionList
     [INTO SelectTargetList] [FROM Rels [WHERE WhereCondition] [GROUP BY
     Atts] [HAVING HavingCondition] [ORDER BY OrderDescription] [OFFSET
     Integer [LIMIT Integer]] [FETCH FIRST Integer ROWS ONLY]]
     ```

   - *Devuelve*:

     ```
     (select(DistinctAll,TopN,Offset,ProjList,TargetList,from(Relations),
     where(WhereCondition), group_by(GroupList), having(HavingCondition),
     order_by(OrderArgs,OrderSpecs)),_AS)
     ```

     - **DistinctAll**: distinct o all(valor por defecto). La cláusula `DISTINCT` asegura que el conjunto de resultados contenga solo filas únicas, excluyendo las duplicadas.
     - **TopN**: La cantidad de las primeras filas que se quieren recuperar.
     - **Offset**: Especifica cuántos registros se deben saltar.
     - **ProjList**: Es la lista de columnas o expresiones que se quiere seleccionar.
     - **TargetList**: Es una lista de nombres de variables del sistema o del usuario que reciben valores de `ProjList`.
     - **from(Relations)**: Representa la cláusula `FROM` en SQL y las relaciones o tablas de las cuales se extraerán los datos.
     - **where(WhereCondition)**: Representa la cláusula `WHERE`. Se utiliza para filtrar registros basándose en una o más condiciones.
     - **group_by(GroupList)**: Representa la cláusula `GROUP BY`. Se utiliza para agrupar registros que tienen los mismos valores en columnas específicas.
     - **having(HavingCondition)**: Representa la cláusula `HAVING`, que se utiliza para filtrar el resultado después de un `GROUP BY` basado en condiciones agregadas.
     - **order_by(OrderArgs,OrderSpecs)**: Representa la cláusula `ORDER BY` en SQL, que se utiliza para ordenar el conjunto de resultados. `OrderArgs` la lista de columnas por las cuales ordenar y `OrderSpecs` indica si el ordenamiento es ascendente o descendente.

   - *Ejemplo*:

     - **Entrada:**
       `SELECT a FROM taras GROUP BY a HAVING sum(b)=1`

- **Salida:**
  ```
  (select(all,top(all),no_offset,[expr(attr(_,a,_),_,_)],
   [],from([(taras,_)]),where(true),group_by([expr(attr(
   _,a,_),_,_)]),having(sum(attr(_,b,_))=cte(1,
   number(_))),order_by([],[])),_)
  ```

2. `SELECT sin FROM`

   - *Sintaxis EBNF*:
     ```
     SELECT [TOP IntegerExpression] [[ALL|DISTINCT]]
     SelectExpressionList [INTO SelectTargetList]
     ```
   - *Devuelve*:
     ```
     (select(DistinctAll,TopN,no_offset,ProjList,TargetList,
     from([(dual,_Ren)]),where(true),group_by([]),having(true),
     order_by([],[])),_AS)
     ```
   - *Ejemplo*:
     - **Entrada:**
       ```
       SELECT 1 a, a+1
       ```
     - **Salida:**
       ```
       (select(all,top(all),no_offset,[expr(cte(1,number(N)),
        a,number(N)),expr(attr(_,a,_)+cte(1,number(_)),_,
        number(_))],[],from([(dual,_)]),where(true),
        group_by([]),having(true),order_by([],[])),_)
       ```

3. `UNION`

   - *Sintaxis EBNF*: `DQLstmt UNION [ALL] DQLstmt`
   - *Devuelve*: `(union(D,R1,R2),_AS)`
     - **D**: distinct(valor por defecto) o all.
     - **R1**: Primera sentencia de DQL.
     - **R2**: Segunda sentencia de DQL.
   - *Ejemplo*:
     - **Entrada:**
       ```
       SELECT * FROM a union select * from b
       ```
     - **Salida:**
       ```
       (union(distinct,(select(all,top(all),no_offset,*,[],
        from([(a,_)]),where(true),group_by([]),having(true),
        order_by([],[])),_),(select(all,top(all),no_offset,
        *,[],from([(b,_)]),where(true),group_by([]),
        having(true),order_by([],[])),_)),_)
       ```

4. `EXCEPT/MINUS`

   - *Sintaxis EBNF*: `DQLstmt [EXCEPT|MINUS] [ALL] DQLstmt`
   - *Devuelve*: `(except(D,R1,R2),_AS)`
     - **D**: distinct(valor por defecto) o all.

- **R1**: Primera sentencia de DQL.
- **R2**: Segunda sentencia de DQL.

- *Ejemplo*:
```
SELECT * FROM a
MINUS
select * from b;
```
⇒ Ídem que `union`, solo cambiando `union` por `except`.

5. `INTERSECT`

- *Sintaxis EBNF*: `DQLstmt INTERSECT [ALL] DQLstmt`
- *Devuelve*: `(intersect(D,R1,R2),_AS)`
  - **D**: distinct(valor por defecto) o all.
  - **R1**: Primera sentencia de DQL.
  - **R2**: Segunda sentencia de DQL.

- *Ejemplo*:
```
SELECT * FROM a
Intersect
select * from b;
```
⇒ Ídem que `union`, solo cambiando `union` por `intersect`.

6. `WITH`

- *Sintaxis EBNF*:
  `WITH LocalViewDefinition {,LocalViewDefinition} DQLstmt`
- *Devuelve*: `(with(SQLst,SQLsts),_AS)`
  - **SQLst**: La consulta principal que sigue a la cláusula.
  - **SQLsts**: Una lista de definiciones de vistas temporales.
- *Ejemplo*:
  - **Entrada:**
    ```
    with p(a) as select 1 union select a+1 from p select top 10 a
    from p;
    ```
  - **Salida:**
    ```
    (with((select(all,top(expr(10,_,number(int))),no_offset,
     [expr(attr(_,a,_),_,_)],[],from([(p,_)]),where(true),
     group_by([]),having(true),order_by([],[])),_),[(union(
     distinct,(select(all,top(all),no_offset,[expr(cte(1,
     number(A)),_,number(A))],[],from([(dual,_)]),where(true),
     group_by([]),having(true),order_by([],[])),_),(select(all,
     top(all),no_offset,[expr(attr(_,a,_)+cte(1,number(_)),_,
     number(_))],[],from([(p,_)]),where(true),group_by([]),
     having(true),order_by([],[])),_)),p(a:_))]),_)
    ```

7. `ASSUME`

- *Sintaxis EBNF*: `ASSUME LocalAssumption {,LocalAssumption} DQLstmt`
- *Devuelve*: `(with(SQLst,SQLsts),_AS)`
  - **SQLst**: La consulta principal que sigue a la cláusula.

- **SQLsts**: Una lista de definiciones de vistas suposiciones.

- *Ejemplo*:
  - **Entrada:**
    ```
    assume select 1 in l(a) select * from l;
    ```
  - **Salida:**
    ```
    (with((select(all,top(all),no_offset,*,[],from([(l,_)]),
     where(true),group_by([]),having(true),order_by([],[])),_),
     [(select(all,top(all),no_offset,[expr(cte(1,number(A)),_,
     number(A))],[],from([(dual,_)]),where(true),group_by([]),
     having(true),order_by([],[])),l(a:_))]),_)
    ```

```
1  % SELECT
2  select_DQL([(select(DistinctAll,TopN,Offset,
3                  ProjList,TargetList,
4                  from(Relations),
5                  where(WhereCondition),
6                  group_by(GroupList),
7                  having(HavingCondition),
8                  order_by(OrderArgs,OrderSpecs)),
9                  _AS)|STs]/STs) -->
10     select_stmt(DistinctAll,TopN),
11     projection_list(ProjList),
12     target_clause(TargetList),
13     cmd(from/_)                    # 'a valid FROM clause',
14     {!},
15     opening_parentheses_star(N),
16     relations(Relations)           # 'a valid FROM clause',
17     where_clause_with_cut(WhereCondition),
18     group_by_clause(GroupList),
19     having_clause(HavingCondition),
20     order_by_clause(OrderArgs,OrderSpecs),
21     optional_offset_limit(Offset),
22     optional_fetch_first(TopN),
23     closing_parentheses_star(N),
24     {set_topN_default(TopN)}.
```

Listado 6.4: Reconocimiento de sentencia de SELECT

La regla `select_DQL/3` del listado 6.4 intenta analizar una consulta SQL `SELECT`. El fragmento de código que sigue a la regla describe los componentes de una consulta `SELECT`.

1. `select_stmt(DistinctAll,TopN)`: Analiza las partes `SELECT DISTINCT` y el número de filas a seleccionar.

2. `projection_list(ProjList)`: Analiza la lista de columnas o expresiones a seleccionar.

3. `target_clause(TargetList)`: Analiza la cláusula de destino.

4. `cmd(from/_)`: Espera encontrar la palabra `FROM`.
5. `opening_parentheses_star(N)` y `closing_parentheses_star(N)`: Identifican y manejan una cantidad variable de paréntesis que abren y cierran. La variable `N` mantiene un registro del número de paréntesis para garantizar que haya un equilibrio entre los que se abren y los que se cierran.
6. `relations(Relations)`: Analiza las tablas y relaciones de las que se extraen los datos.
7. `where_clause_with_cut(WhereCondition)`: Analiza la cláusula `WHERE`. Hace la deteccion del `cmd(where/_)` y después una condición SQL (véase la sección 6.4).
8. `group_by_clause(GroupList)`: Analiza la cláusula `GROUP BY`. Detecta primero la palabra `cmd(group/_)` seguido de `cmd(by/_)` después una secuencia de expresiones (véase la sección 6.6).
9. `having_clause(HavingCondition)`: Analiza la cláusula `HAVING`. Hace la deteccion del `cmd(having/_)` y después una condición SQL (véase la sección 6.4).
10. `order_by_clause(OrderArgs,OrderSpecs)`: Analiza la cláusula `ORDER BY`.

6.11. Sentencias del lenguaje de definición de datos (DDL - Data Definition Language)

El Lenguaje de Definición de Datos es esencial para establecer y modificar la estructura de las bases de datos en SQL. A continuación se presentan las sentencias DDL en DES:

La gramática en sintaxis similar a EBNF para las sentencias DDL se describe detalladamente en el apéndice D.

1. `CREATE TABLE`
   - *Sintaxis EBNF*:
     `CREATE [OR REPLACE] TABLE CompleteConstrainedSchema.`
   - *Devuelve*: `CRTSchema`. De forma `[CRT,Schema,Ctrs]` donde `CTR` es `create_table` o `create_or_replace_table`, `Schema` es el esquema de la tabla y `Ctrs` son restricciones de las columnas o tabla (véase la sección 6.7) es `true` si no hay restricciones.
   - *Ejemplo*:
     - **Entrada:**
       `create table t(a int)`
     - **Salida:**
       `create_table(t(a:number(integer)),[true])`
2. `CREATE TABLE AS`
   - *Sintaxis EBNF*:
     `CREATE [OR REPLACE] TABLE TableName [(] LIKE TableName [)].`

- *Devuelve*: `CRTSchema`. De forma `[CRT,(LSQLst,_AS),Schema]` donde `CTR` es `create_table_as` o `create_or_replace_table_as`, `LSQLst` es la sentencia DQL (véase la sección 6.10), y `Schema` es el esquema de la tabla.
- *Ejemplo*:
  - **Entrada:**
    ```
    create table t3(a3,b3,c3) as select a from n
    ```
  - **Salida:**
    ```
    create_table_as((select(all,top(all),no_offset,
     [expr(attr(_,a,_),_,_)],[],from([(n,_)]),where(true),
     group_by([]),having(true),order_by([],[])),_),
     t3(a3:_,b3:_,c3:_))
    ```

3. `CREATE TABLE LIKE`
   - *Sintaxis EBNF*:
     `CREATE [OR REPLACE] TABLE TableName [() AS DQLstmt ()]`.
   - *Devuelve*: `CRTSchema`. De forma `[CRT,TableName,ExistingTableName]` donde `CRT` es `create_table_like` o `create_or_replace_table_like`, `TableName` nombre de la tabla a crear y `ExistingTableName` nombre de la tabla existente.
   - *Ejemplo*:
     - **Entrada:**
       ```
       create or replace table t like s
       ```
     - **Salida:**
       ```
       create_or_replace_table_like(t,s)
       ```
4. `CREATE VIEW`
   - *Sintaxis EBNF*: `CREATE [OR REPLACE] VIEW Schema AS DQLstmt`.
   - *Devuelve*: `CRVSchema`. De forma `[CRVF,sql,(LSQLst,_AS),Schema]` donde `CRVF` es `create_view` o `create_or_replace_view`, `LSQLst` sentencia DQL de SQL (véase la sección 6.10) y `Schema` la estructura de la vista.
   - *Ejemplo*:
     - **Entrada:**
       ```
       create view v("a") as select b from "t";
       ```
     - **Salida:**
       ```
       create_view(sql,(select(all,top(all),no_offset,
        [expr(attr(_,b,_),_,_)],[],from([(t,_)]),where(true),
        group_by([]),having(true),order_by([],[])),_),v(a:_))
       ```
5. `CREATE DATABASE`
   - *Sintaxis EBNF*: `CREATE DATABASE DatabaseName`.
   - *Devuelve*: `create_database(DBName)` donde `DBName` es el nombre de la base de datos.
   - *Ejemplo*:

- **Entrada:**
  `create database x`
- **Salida:**
  `create_database(x)`

6. `ALTER TABLE [ADD|DROP]`

   - *Sintaxis EBNF*:
     `ALTER TABLE TableName [ADD|DROP] | [[COLUMN] Att | CONSTRAINT [ConstraintName] TableConstraint]`.
   - *Devuelve*: `alter_table(TableName,AD,Element)`
     - **TableName**: Nombre de la tabla.
     - **AD**: add o drop.
     - **Element**: Elemento a alterar con las restricciones de tabla.
   - *Ejemplo*:
     - **Entrada:**
       `alter table t1 add a int not null`
     - **Salida:**
       ```
       alter_table(t1,add,column(a:number(integer),
        [not_nullables([a])]))
       ```

7. `ALTER TABLE ALTER`

   - *Sintaxis EBNF*:
     `ALTER TABLE TableName ALTER [COLUMN] Att [AttDefinition | SET [DATA] TYPE Type]`.
   - *Devuelve*: `alter_table(TableName,AD,Element)`
     - **TableName**: Nombre de la tabla.
     - **AD**: alter.
     - **Element**: Elemento a alterar con los constricciones de la tabla.
   - *Ejemplo*:
     - **Entrada:**
       `alter table t1 alter column a1 set data type varchar(10)`
     - **Salida:**
       `alter_table(t1,alter,column(a1:string(varchar(10))))`

8. `RENAME TABLE`

   - *Sintaxis EBNF*: `RENAME TABLE TableName TO TableName`.
   - *Devuelve*: `rename_table(TableName,NewTableName)`
     - **TableName**: Nombre de la tabla.
     - **NewTableName**: Nuevo nombre de la tabla.
   - *Ejemplo*:
     - **Entrada:**
       `rename table t to s`

- **Salida:**
  `rename_table(t,s)`

9. `RENAME VIEW`
   - *Sintaxis EBNF*: `RENAME VIEW ViewName TO ViewName`.
   - *Devuelve*: `rename_view(Viewname,NewViewname)`
     - **Viewname**: Nombre de la vista.
     - **NewViewname**: Nuevo nombre de la vista.
   - *Ejemplo*:
     - **Entrada:**
       `rename view v to s`
     - **Salida:**
       `rename_view(v,s)`
10. `DROP TABLE`
    - *Sintaxis EBNF*:
      `DROP TABLE DropTableClauses TableName{,TableName} DropTableClauses`.
    - *Devuelve*: `drop_table(Name,Clauses)`
      - **Name**: Nombre de la tabla a eliminar.
      - **Clauses**: Las cláusulas opcionales identificadas antes y después del nombre de la tabla.
    - *Ejemplo*:
      - **Entrada:**
        `drop table if exists t`
      - **Salida:**
        `drop_table(t,[if_exists])`
11. `DROP VIEW`
    - *Sintaxis EBNF*: `DROP VIEW DropViewClauses ViewName DropViewClauses`.
    - *Devuelve*: `drop_view(Name,Clauses)`
      - **Name**: Vista a eliminar.
      - **Clauses**: Las cláusulas opcionales identificadas antes y después de la vista.
    - *Ejemplo*:
      - **Entrada:**
        `drop view v`
      - **Salida:**
        `drop_view(v,[])`
12. `DROP SCHEMA`
    - *Sintaxis EBNF*: `DROP DATABASE [DatabaseName]`.
    - *Devuelve*: `drop_database(DBName)` donde DBName es el nombre de base de datos.

- *Ejemplo*:
  - **Entrada:**
    `drop database db`
  - **Salida:**
    `drop_database(db)`

13. `HR-SQL CREATE VIEW syntax`

- *Sintaxis EBNF*: `CompleteSchema := DQLstmt`
  Añadido para soportar la sintaxis HR-SQL.
- *Devuelve*: `CRVSchema` de forma
  `[create_or_replace_view,hrsql,(SQLst,_AS),Schema]` donde `SQLst` es la sentencia DQL y `Schema` es el esquema por la parte izquierda de `:=`.
- *Ejemplo*:
  - **Entrada:**
    `my_view(age int) := SELECT age FROM my_table`
  - **Salida:**
    ```
    create_or_replace_view(hrsql,(select(all,top(all),
     no_offset,[expr(attr(_,age,_),_,_)],[],
     from([(my_table,_)]),where(true),group_by([]),having(true),
     order_by([],[])),_),my_view(age:number(integer)))
    ```

```
1  % CREATE TABLE AS
2  ddlStmt([CRTSchema|STs]/STs) -->
3      create_or_replace(CR),
4      cmd((table)/_)                          # 'TABLE or VIEW',
5      create_view_schema(Schema)              # 'opening parenthesis
           ''('' or LIKE or AS or column name',
6      % syntax_check_redef(Schema), %  If attempting to
         redefine a datalog keyword, exception is thrown.
7      opening_parentheses_star(N),
8      cmd(as/_)                               # 'AS',
9      dqlStmt([(LSQLst,Schema)|STs]/STs) # 'valid SQL DQL
         statement (SELECT, WITH or ASSUME)',
10     closing_parentheses_star(N),
11     {atom_concat(CR,'_table_as',CRT),
12     CRTSchema=..[CRT,(LSQLst,_AS),Schema]},
13     set_error_no_fail('Syntax', 'end of statement').
```

Listado 6.5: Reconocimiento de sentencia de CREATE TABLE AS

El fragmento de código en el listado 6.5 podemos ver como analiza e interpreta las sentencias `CREATE TABLE AS` en SQL.

1. `create_or_replace(CR)`: Esta regla identifica si la sentencia es simplemente `CREATE` o tiene la opción `OR REPLACE`. La variable `CR` captura esta elección.

2. `cmd((table)/_)`: Busca la palabra clave `TABLE`. Esta parte se asegura de que la sentencia esté tratando de crear una tabla y no otra estructura, como una vista.

3. `create_view_schema(Schema)`: Analiza y captura la estructura o esquema de la tabla que se va a crear.

4. `cmd(as/_)`: Reconoce la palabra clave `AS`.

5. `dqlStmt[(LSQLst,Schema)|STs]/STs)`: Analiza la sentencia DQL de SQL que proporciona los datos para la nueva tabla (véase la sección 6.10).

6. `{atom_concat(CR,'table_as',CRT), CRTSchema=..[CRT,(LSQLst,_AS),Schema]}`: Esta es una operación en tiempo de ejecución que construye dinámicamente el nombre de la operación de creación.

7. `set_error_no_fail('Syntax', 'end of statement')`: Esta regla establece un mensaje de error sin causar el fallo de todo el proceso de análisis para indicar que se espera un fin de sentencia.

6.12. Sentencias del lenguaje de manipulación de datos (DML - Data Manipulation Language)

En SQL, las operaciones relacionadas con la manipulación de los datos almacenados se llaman sentencias DML. A continuación se describen las sentencias más comunes y su funcionalidad:

La gramática en sintaxis similar a EBNF para las sentencias DML se describe detalladamente en el apéndice D.

1. `INSERT INTO(I)`

   - *Sintaxis EBNF*: `INSERT INTO TableName[(Att {,Att})] VALUES (ExprDef {,ExprDef}) {, (ExprDef {,ExprDef})}`.
   - *Devuelve*: `insert_into(TableName,Colnames,Vs)`.
     - **TableName**: Nombre de la tabla a insertar nuevos elementos.
     - **Colnames**: Nombre de las columnas de dicha tabla.
     - **Vs**: Lista de valores a insertar.
   - *Ejemplo*:
     - **Entrada:**
       ```
       insert into t1 values(1.5E2), ('2')
       ```
     - **Salida:**
       ```
       insert_into(t1,[a1],
        [[cte(150.0,number(float))],[cte('2',string(_))]])
       ```

2. `INSERT INTO(II)`

   - *Sintaxis EBNF*: `INSERT INTO TableName DEFAULT VALUES`.
   - *Devuelve*: Ídem a `INSERT INTO(I)`, los valores de `Vs` serán default.
   - *Ejemplo*:
     - **Entrada:**
       ```
       insert into t2 default values
       ```

- **Salida:**
  `insert_into(t2,[a2,b2],[[default,default]])`

3. `INSERT INTO(III)`

   - *Sintaxis EBNF*: `INSERT INTO TableName[(Att {,Att})] DQLstmt`.
   - *Devuelve*: Ídem a `INSERT INTO(I)`.
   - *Ejemplo*:
     - **Entrada:**
       `insert into t2 select a.a,b.b from a,b where a.a=b.b`
     - **Salida:**
       ```
       insert_into(t2,[a2,b2],(select(all,top(all),no_offset,
        [expr(attr(a,a,_),_,_),expr(attr(b,b,_),_,_)],[],
        from([(a,_),(b,_)]),where(attr(a,a,_)=attr(b,b,_)),
        group_by([]),having(true),order_by([],[])),_))
       ```

4. `DELETE FROM`

   - *Sintaxis EBNF*:
     `DELETE FROM TableName [[AS] Identifier] [WHERE Condition]`.
   - *Devuelve*: `delete_from(Table,WhereCondition)`
     - **Table**: Nombre de la tabla para eliminar.
     - **WhereCondition**: Condición que se debe cumplir para eliminar, `true` si no se especifica (véase la sección 6.4).
   - *Ejemplo*:
     - **Entrada:**
       `delete from t where b='a1'`
     - **Salida:**
       `delete_from((t,_),attr(_,b,_)=cte(a1,string(_)))`

5. `UPDATE`

   - *Sintaxis EBNF*: `UPDATE TableName [[AS] Identifier] SET Att=Expr {,Att=Expr} [WHERE Condition]`.
   - *Devuelve*: `update(Table,Assignments,WhereCondition)`
     - **Table**: Nombre de la tabla para actualizar.
     - **Assignments**: De forma `expr(ColumnName,_,string),Expression` donde `ColumnName` es el nombre de la columna a actualizar y `Expression` es la expresión para la actualización.
     - **WhereCondition**: Condición que se debe cumplir para actualizar, `true` si no se especifica (véase la sección 6.4).
   - *Ejemplo*:
     - **Entrada:**
       `update t1 set a=1`

- **Salida:**
  ```
  update((t1,_),[expr(a,_,string),
   expr(cte(1,number(A)),_,number(A))],true))
  ```

```
1  dmlStmt([insert_into(TableName,Colnames,Vs)|STs]/STs) -->
2      [cmd(insert/_):_],
3      cmd(into/_)                  # 'INTO',
4      tablename(TableName)         # 'table name',
5      current_position(Position),
6      punct('(')                   # 'opening parenthesis ''('' or DEFAULT',
7      column_name_list(Colnames) # 'a sequence of columns between parentheses',
8      punct(')')                   # 'comma or closing parenthesis '')''',
9      {my_remove_duplicates(Colnames,Colnames) -> true ;
10     set_error_with_parameter('Semantic', 'Column names must be different in ~w' , [Colnames], Position)},
11     {length(Colnames,L)},
12     insert_values_sql(L,Vs)      # 'VALUES, select statement, or DEFAULT VALUES',
13     set_error_no_fail('Syntax', 'end of statement').
```

Listado 6.6: Reconocimiento de sentencia de INSERT INTO Table(Columns)

El fragmento de código 6.6 ilustra cómo se analiza y se interpreta la sentencia SQL `INSERT INTO Table(Columns)`.

1. `[cmd(insert/_):_]` y `cmd(into/_)`: Estas reglas reconoce el comando `INSERT` y la palabra clave `INTO`.

2. `tablename(TableName)`: Esta regla detecta y captura el nombre de la tabla en la que se insertarán los datos.

3. `current_position(Position)`: Obtiene la posición actual en la sentencia SQL para señalar la ubicación exacta de un posible error.

4. `punct('(')` y `punct(')')`: Estas reglas identifican los paréntesis que encierran la lista de nombres de columnas.

5. `column_name_list(Colnames)`: Analiza la lista de nombres de columnas en las que se insertarán los valores. La lista está delimitada por paréntesis y los nombres están separados por comas.

6. El siguiente bloque verifica la unicidad de los nombres de las columnas:

   ```
   {my_remove_duplicates(Colnames,Colnames) -> true;
   set_error_with_parameter('Semantic', 'Column names must be different in ~w', [Colnames], Position)}
   ```

   Si hay columnas duplicadas, se genera un error semántico.

7. `length(Colnames,L)`: Calcula y captura la cantidad de columnas especificadas.

8. `insert_values_sql(L,Vs)`: Reconoce los valores que se insertarán en la tabla. Esta regla se adapta al número de columnas especificadas y verifica que la cantidad de valores coincida con el número de columnas.

9. `set_error_no_fail('Syntax', 'end of statement')`: Esta regla establece un mensaje de error sin hacer que el proceso de análisis completo falle.

6.13. Dificultades y desafíos del analizador sintáctico

- **Corte:** El uso y ubicación de los cortes ha sido esencial para asegurar la eficiencia del análisis sintáctico. Cualquier corte mal ubicado podría perjudicar la funcionalidad del sistema o hace que pierda algunas soluciones posibles que deberían ser consideradas.

- **Orden de reconocimiento de las sentencias:** Si se reconoce una regla antes que otra, debido a los cortes, el resultado podría variar significativamente. Es imprescindible que algunas sentencias se reconozcan después de otras para garantizar que el analizador sintáctico (parser) funcione adecuadamente.

- **Sentencias DQL:** El análisis de las sentencias DQL ha presentado dificultades, especialmente porque depende de numerosos predicados. En la cláusula `FROM`, es necesario hacer una previsualización de los tokens siguientes para identificar palabras clave relacionadas con JOIN. De no hacerlo, el programa entraría en un bucle infinito.

- **Gestión de errores:** Es imposible prever todos los errores que un usuario podría encontrar al usar el sistema. Como se ha mencionado anteriormente, los cortes complican la tarea de identificar qué se espera en cada momento. Además, no siempre se espera un único resultado; en ocasiones hay varias opciones posibles y es necesario considerar todas ellas.

Capítulo 7

Gestión de errores

En este capítulo se aborda la gestión de errores que se lleva a cabo durante las etapas de análisis léxico y sintáctico del proyecto.

7.1. Gestión de errores en el analizador léxico

A continuación se presentan algunos de los errores junto con ejemplos ilustrativos:

- **fractional:** Este error ocurre cuando la fracción de un número no está correctamente estructurada.

  Ejemplo: **-1.a** ⇒ Lexical Error: Incorrect fractional at Line 1, Column 4.

- **exponent:** Este error se presenta cuando el exponente de un número no sigue la estructura correcta.

  Ejemplo: **0.1E++2** ⇒ Lexical Error: Incorrect exponent at Line 1, Column 5.

- **unclosed delimited ID, separator, or unrecognized token:** Este tipo de error se genera cuando hay un identificador delimitado no cerrado, un separador incorrecto o un token no reconocido.

  Ejemplos:

  [t ⇒ Lexical Error: Unclosed delimited ID or incorrect separator or unrecognized token at Line 1, Column 1.

  t[⇒ Lexical Error: Unclosed delimited ID or incorrect separator or unrecognized token at Line 1, Column 2.

  1a ⇒ Lexical Error: Unclosed delimited ID or incorrect separator or unrecognized token at Line 1, Column 2.

7.2. Gestión de errores en el analizador sintáctico

La gestión de errores en el analizador sintáctico se realiza mediante el uso del operador `#`. El operando a la izquierda de este operador representa el predicado (Goal) o token terminal, mientras que el operando a la derecha especifica lo que se espera en la sentencia(por ejemplo que después de reconocer el token cmd(create/_) se espera `TABLE` o `VIEW`

o `OR REPLACE` o `DATABASE`). Si se encuentra un token terminal, el analizador sintáctico verifica si coincide con el token actual. En caso de coincidencia, continúa procesando el resto de la sentencia SQL. Si no coincide, busca alternativas mediante la vuelta atrás. Si no se encuentra ningún camino exitoso, se muestra el error que alcanzó la posición más avanzada en la entrada (véase el listado 7.1).

Por ejemplo:

- `cmd(set/_) # 'SET'`: Aquí se espera el token `cmd(set/_)`. Si no se encuentra, el sistema llamará al mecanismo de manejo de errores, representado por `# 'SET'`, para actualizar el mensaje de error correspondiente.
- `p_ren_tablename(Table) # 'table name'`: En este caso, se espera encontrar una definición válida para un nombre de tabla. Se invoca al mecanismo `Goal # Error` para actualizar el mensaje de error y luego se continúa con la ejecución de `p_ren_tablename(Table)`.

```
1  % +Token # +Error//
2  % Set errors for expected tokens
3  Token # Error -->
4    {terminal(Token),
5     !},
6    [CurrentToken:Position],
7    {set_error('Syntax', Error, Position),
8     Token = CurrentToken},
9    !.
10 Goal # Error -->
11   current_position(Position),
12   {set_error('Syntax', Error, Position)},
13   Goal.
```

Listado 7.1: Gestión de errores en analizador sintáctico

Los errores se indican de forma:

`Syntax Error:El ERROR DETECTADO at Line L, Column C.` [1]

`Syntax Error:El ERROR DETECTADO at the end of the program`. Si el error ocurre al final del programa.

Entrada	Syntax Error: Expected ...	at Posición (Line, Column)
DESCRIBE 2	table name or view name	1, 10
show t	TABLES, VIEWS or DATABASES	1, 6

Tabla 7.1: Errores sintácticos en consultas ISL

[1] En los ejemplos, todas las sentencias están en una única línea, por lo que siempre es la línea 1 donde ocurre el error.

Entrada	**Syntax Error: Expected ...**	**at Posición (Line, Column)**
ROLLBACK WORK TO point sp1	SAVEPOINT	1, 13
ROLLBACK TO SAVEPOINT "s""p1"	savepoint name(id without semicolon or quoted id without escaped quotes)	1, 23

Tabla 7.2: Errores sintácticos en consultas TML

Entrada	**Syntax Error: Expected ...**	**at Posición (Line, Column)**
select 2 1 from t	comma or FROM clause or end of SELECT statement	1, 10
SELECT * FROM t WHERE a	a comparison operator	the end of the program
select * from 1 right join t2	a valid FROM clause	1, 15
select * from a union b	SELECT statement	1, 23
with v(a) select 1 select * from v	AS	1, 11
with 2 select 1 select * from v	assume/with schema	1, 6

Tabla 7.3: Errores sintácticos en consultas DQL

Entrada	**Syntax Error: Expected ...**	**at Posición (Line, Column)**
create or table t(a int)	REPLACE	1, 11
create or replace able t(a int)	TABLE or VIEW	1, 19
create table t('a' intiger)	LIKE or AS or column name	1, 16
create table t	opening parenthesis (or LIKE or AS or column name	1, 14
create or replace table t(a int primary kye)	KEY	1, 41
create or replace table t(a int, unique s,)	valid table constraint (NOT, PRIMARY, UNIQUE, FOREIGN, CHECK, CANDIDATE)	1, 43
create table t like 2	table name	1, 21
create view	view schema	the end of the program

Tabla 7.4: Errores sintácticos en consultas CREATE de DDL

Entrada	**Syntax Error: Expected ...**	**at Posición (Line, Column)**
alter t1 add constraint primary key(a)	TABLE after ALTER	1, 7
alter table t1 ad a int	ALTER, ADD or DROP	1, 16
alter table t1 alter column a1 et data type varchar(10)	SET	1, 32
alter table t1 alter column a1 set ata type varchar(10)	DATA TYPE or TYPE	1, 36
rename table t(a) to s	TO	1, 15
drop t	TABLE or VIEW or DATABASE	1, 6
drop table if exist t	EXISTS after IF	1, 15
drop view 2 v	view name or optional drop view clauses(IF EXISTS, CASCADE)	1, 11

Tabla 7.5: Errores sintácticos en otras consultas DDL

Entrada	**Syntax Error: Expected ...**	**at Posición (Line, Column)**
insert into t1 default v	VALUES after DEFAULT	1, 24
insert into t()	a sequence of columns between parentheses	1, 15
insert into t	VALUES, select statement, or DEFAULT VALUES	the end of the program
delete * from t	FROM	1, 8
update t a=1	SET	1, 10
update t set	sequence of column assignments Col=Expr	the end of the program

Tabla 7.6: Errores sintácticos en consultas DML

Entrada	**Semantic Error**	**at Posición (Line, Column)**
alter table t drop a	Unknown table t	1, 13
alter table t alter column a1 set data type varchar(10)	unknown_column(t, a1)	1, 28
alter table t1 alter column a string default ''	unknown_column(t1, a)	1, 29
select top 1 distinct * from t fetch first 1 rows only	Only one TOP/LIMIT/FETCH specification is allowed	void, void[2]
insert into t3(a2,a3) values (1,2, 'a')	Unmatching number of values => 3 (must be 2)	1, 30
insert into t2(a3,b3,a3) values (1,2,'a')	Column names must be different in [a3,b3,a3]	1, 15

[2] En este caso especial de análisis sintáctico, no indica la posición exacta donde se produce al usuario.

Tabla 7.7: Errores semánticos en el proceso de analizador sintáctico

Capítulo 8

Pruebas y comparaciones

8.1. Metodología de prueba

Para la prueba, como se mencionó en el apartado 6, por cada cambio que se ha realizado en el lexer.pl o parser.pl, se ejecuta lexer:test (35 tests) y parser:test (141 tests) para asegurarnos de que la salida es lo que se espera. Estos casos de prueba se derivaron directamente del archivo de tests presente en DES.

8.2. Comparaciones de resultados

El principal objetivo de las comparaciones es asegurar que la salida del analizador sintáctico de nuestra nueva implementación y la salida actual de DES produzcan resultados idénticos, facilitando así la integración del nuevo analizador sintáctico en DES para los procesos subsiguientes relacionados con el lenguaje SQL. Además, es esencial la importancia de evaluar la mejora en términos de tiempo de ejecución que la nueva implementación aporta.

En general, para las sentencias `DML`, `DDL`, `ISL` y `TML`, ambos sistemas producen resultados en un tiempo inferior a 5ms, sin presentar diferencias significativas. Sin embargo, las sentencias que involucran `DQL` muestran una discrepancia notable a medida de que la consulta sea más compleja. Por ejemplo, para la sentencia `SELECT first_name, last_name FROM employees WHERE department_id = 10;`, DES tarda aproximadamente 8ms, mientras que nuestra nueva implementación lo realiza en 2ms.

A continuación, se mostrarán unos fragmentos de código del analizador sintáctico del sistema actual de DES, y posteriormente se detallarán las diferencias con la nueva implementación.

```
1 parse_sql_query(SQLst) -->
2   my_sql_blanks_star, % The ODBC driver may return blanks
3   {reset_semantic_error,
4    my_retract_all_facts(dictionary(_)),
5    assertz(dictionary([]))}, % WARNING: ONLY FOR TEST CASE
       GENERATION
6   push_syntax_error([‘Expected valid SQL statement (SELECT,
      CREATE, DELETE, INSERT, UPDATE, DROP, RENAME, ALTER,
      SHOW, DESCRIBE, WITH, ASSUME, COMMIT, ROLLBACK,
```

```
      SAVEPOINT)'],Old),
7   my_SQL(SQLst),
8   pop_syntax_error(Old),
9   my_sql_blanks_star,
10  my_optional(";"),
11  my_sql_blanks_star, % The ODBC driver may return blanks
12  !.
13
14 % DQL Statement
15 my_SQL(SQLst) -->
16   my_DQL(SQLst).
17 % DML Statement
18 my_SQL(SQLst) -->
19   my_DML(SQLst).
20 % DDL Statement
21 my_SQL(SQLst) -->
22   my_DDL(SQLst).
```

Listado 8.1: Analizador sintáctico en la versión 6.7 de DES

El predicado `parse_sql_query` del fragmento de código del listado 8.1 comienza buscando espacios en blanco, comentarios de SQL, saltos de línea, etc. Luego realiza un reseteo de los errores. Posteriormente emplea `push_syntax_error` y `pop_syntax_error` para manejo de errores. Concretamente, el primero se utiliza para registrar un error y guardar en una lista utilizando una variable, mientras que el segundo se emplea para eliminar dicho error tras haber reconocido los tokens o textos esperados. Finalmente, se analiza la sentencia que podría ser uno de los cinco tipos mencionados en el capítulo 6, se verifica de manera opcional el punto y coma (`;`), y se concluye con otro `my_sql_blanks_star`.

Se pueden identificar dos grandes diferencias con la nueva implementación:

- Podemos ver que se gestionan de manera diferente los errores sintácticos comparado con la nueva implementación (véase la sección 7.2), y se guardan en una lista de errores todos los valores posibles hasta ese momento. Una vez reconocido dicho token se elimina de la lista de errores y no se guarda la posición de las sentencias.

- En la versión actual (6.7) de DES, el analizador sintáctico se encarga de reconocer y descartar espacios en blanco, saltos de línea y comentarios de SQL directamente dentro de las reglas del analizador sintáctico, como se evidencia con el uso repetido de `my_sql_blanks_star`. Esto puede complicar el código y hacer que el reconocimiento de las sentencias SQL sea menos claro. En la nueva implementación, estos tokens se identifican en analizador léxico y se filtran antes de empezar con el analizador sintáctico.

  Esta separación tiene varias ventajas: hace que el código del analizador sintáctico sea más limpio y centrado en la estructura lógica de las sentencias SQL, y simplifica el mantenimiento y extensión del código, ya que el filtrado se realiza antes del análisis sintáctico.

```
1 % DQL Statements
2 my_DQL(SQLst) -->
```

```
3     my_b_DQL(SQLst).
4   my_DQL(SQLst) -->
5     my_ub_DQL(SQLst).
6
7   my_b_DQL(SQLst) -->
8     "(",
9     my_sql_blanks_star,
10    my_DQL(SQLst),
11    my_sql_blanks_star,
12    ")".
13
14  % SELECT
15  my_ub_DQL(SQLst) -->
16    my_select_DQL(SQLst),
17    push_syntax_error([‘Expected end of SELECT statement’]).
18
19  % SELECT
20  my_select_DQL((select(DistinctAll,TopN,Offset,
21                  ProjList,TargetList,
22                  from(Relations),
23                  where(WhereCondition),
24                  group_by(GroupList),
25                  having(HavingCondition),
26                  order_by(OrderArgs,OrderSpecs)),_AS)) -->
27    my_select_stmt(DistinctAll,TopN),
28    push_syntax_error([‘Invalid SELECT list’],Old1),
29    my_projection_list(ProjList),
30    pop_syntax_error(Old1),
31    my_sql_blanks,
32    my_target_clause(TargetList),
33    push_syntax_error([‘Expected FROM clause’],Old2),
34    my_kw("FROM"),
35    pop_syntax_error(Old2),
36    my_opening_parentheses_star(N),
37    my_optional_sql_blanks(N),
38    {!}, % 23-01-2021
39    push_syntax_error([‘Invalid relation’],Old3),
40    my_relations(Relations),
41    pop_syntax_error(Old3),
42    my_where_clause(WhereCondition),
43    my_group_by_clause(GroupList),
44    my_having_clause(HavingCondition),
45    my_order_by_clause(OrderArgs,OrderSpecs),
46    my_optional_offset_limit(Offset),
47    my_optional_fetch_first(TopN),
48    my_closing_parentheses_star(N),
49    {set_topN_default(TopN)},
50    push_syntax_error([‘Expected end of SELECT statement’]).
```

Listado 8.2: Predicado para analizar sentencias DQL en version 6.7 de DES

En este otro fragmento de código, representado en el listado 8.2, es posible identificar las diferencias previamente mencionadas. Por ejemplo, el predicado `my_b_DQL` es ligeramente diferente en comparación con la nueva implementación, como se muestra en el listado 8.3.

```
1 b_DQL([STs1|STs]/STs) -->
2   [punct('('):_],
3   dqlStmt([STs1|STs]/STs),
4   [punct(')'):_].
```

Listado 8.3: Predicado b_DQL en la nueva implementación

Como se observa en el listado 8.3, ya no se requieren comprobaciones para espacios en blanco, saltos de línea o comentarios. En su lugar, simplemente se consume el token `punct('('):_`, se sigue con una sentencia select y, finalmente, se concluye con `punct(')'):_`.

El predicado `my_select_DQL` difiere del código mostrado en el listado 6.4. Además de las diferencias previamente señaladas, se pueden identificar dos discrepancias adicionales:

- En cuanto a las posiciones en las que se actualizan los errores esperados:

  En el sistema actual, los errores sintácticos se actualizan antes de `my_projection_list`, `my_kw("FROM")`, `my_relations`, y al finalizar el reconocimiento de la sentencia `select`. Sin embargo, en la nueva implementación, la actualización solo ocurre en `cmd(from/_)`, y en `relations`, dejando la gestión de otros errores a otros predicados.

- Respecto a la posición del corte:

  En la nueva implementación, este debe estar ubicado antes de `opening_parentheses_star(N)` para permitir regresar si el número de paréntesis de cierre no coincide con el de apertura. En cambio, en la versión actual de DES, el corte se encuentra después de `my_opening_parentheses_star(N)` y `my_optional_sql_blanks(N)`.

8.3. Resultados en sentencias complejas del DQL

Para las sentencias más complicadas como por ejmplo: `SELECT DISTINCT - + 34 + + - 26 + - 34 + - 34 + + COALESCE ( 93, COUNT ( * ) + + 44 - 16, - AVG ( + 86 ) + 12 ) / 86 * + 55 * + 46;` en nuestra nueva implementación, esta consulta se procesa en aproximadamente 50.595 segundos. En el sistema actual no pudo completar el análisis después de 30 minutos.

En la tabla 8.1 se muestran las comparaciones de tiempos para sentencias complejas.

Entrada	**Tiempo actual en DES**	**Tiempo en la nueva implementación**
`SELECT - 34 - 26 + - 34 - 34 + 93 + 44, - 86, count(*), pi, e;`	7.147 s	0.113 s
`SELECT DISTINCT count(*), pi, e, count(*), pi, e, count(*), pi, e, count(*), pi, e;`	13.163 s	0.005 s
`SELECT x.a - 2, x.d, y.c FROM (SELECT a, CASE a WHEN 1 THEN 'Uno' WHEN 2 THEN 'Dos' ELSE 'O' END d, (a + 12) * 3 c FROM (SELECT a FROM l1 UNION ALL SELECT a FROM l2) u) x JOIN (SELECT a, CASE a WHEN 1 THEN 'Uno' WHEN 2 THEN 'Dos' ELSE 'O' END d, (a + 12) * 3 c FROM l3) y ON x.a <> y.a ORDER BY x.a DESC;`	33.534 s	2.357 s
`SELECT - 34 - 26 + - 34 - 34 + COALESCE ( 93, COUNT ( * ) + 44, - 86 );`	38.612 s	0.197 s
`SELECT a1.a, a2.c FROM (SELECT a, (a + 12) * 3 c FROM (SELECT a FROM l1 UNION ALL SELECT a FROM l2) AS u) a1 JOIN (SELECT a, (a + 12) * 3 c FROM (SELECT a FROM l3 UNION ALL SELECT a FROM l4) AS u) a2 ON a1.a <> a2.a ORDER BY a1.a DESC;`	43.679 s	2.737 s

Tabla 8.1: Comparaciones en el tiempo de ejecución

Podría parecer que un análisis realizado en una sola pasada sería más eficiente porque reduce la necesidad de revisar la misma información múltiples veces. Sin embargo, la ventaja de separar el análisis léxico del sintáctico, como se hace en nuestra nueva implementación, radica en poder especializar y optimizar cada etapa por separado. Esta división evita repetir costosas operaciones de fases anteriores debido a la vuelta atrás (backtracking) y ha demostrado mejorar el rendimiento en consultas complejas.

Capítulo 9

Conclusiones y trabajo futuro

La tabla 8.1 demuestra las significativas mejoras en eficiencia que ofrece la nueva implementación en comparación con el sistema actual de DES. A medida que las sentencias SQL se vuelven más complejas, la diferencia en los tiempos de procesamiento se amplía aún más, favoreciendo claramente nuestra nueva solución. En términos de rendimiento, la nueva implementación supera al sistema actual, en particular cuando se trata de sentencias SQL de alta complejidad. La elección de separar el análisis léxico del sintáctico ha aportado claridad y organización al proyecto, lo que facilitará las futuras modificaciones y ajustes.

Aunque se han logrado todos los objetivos previstos para el proyecto, se han identificado adicionalmente diversas tareas para abordar en el futuro. Para trabajos futuros, se contempla la integración de la nueva implementación en DES, con las siguientes modificaciones y adiciones propuestas:

- **Gestión de errores:** Se necesitan ajustes en el manejo de errores, ya que DES ya cuenta con un sistema de gestión de errores que difiere de nuestra implementación reciente. Además de SQL, DES también procesa otros lenguajes de consulta, lo que requiere considerar errores específicos de esos lenguajes. Además, los errores semánticos que pueden surgir durante el análisis sintáctico son tratados de manera distinta en DES, por lo que es esencial abordarlos con precisión.

  Consideremos el siguiente ejemplo:

  ```
  select 2 1 from t
  ```

  La respuesta de nuestra nueva implementación sería:

  ```
  Syntax Error: Expected comma or FROM clause or end of SELECT statement
  at Line 1, Column 10.
  ```

  Por otro lado, el sistema actual proporcionaría:

  ```
  Error: (SQL) Expected comma or (SQL) Expected FROM clause or (SQL)
  Expected end of SELECT statement or (RA) Expected NOT or (RA) Expected
  string type in 2 or (RA) Expected comparison operator near "select 2 "
  ```

  A simple vista, podemos observar el ejemplo que la nueva implementación señala con precisión la línea y columna donde se encuentra el error, limitándose a indicar los posibles errores en SQL. En contraposición, el sistema actual combina errores léxicos y sintácticos en un solo mensaje, y además muestra la sentencia hasta el punto donde podría surgir el error.

Nueva implementación:

- **Ventajas:** Indica con precisión la línea y columna donde se encuentra el error, facilitando su identificación, especialmente si la sentencia provenga de un fichero.
- **Desventajas:** Si la sentencia es ingresada directamente, puede ser más difícil identificar la ubicación exacta del error, habrá que contar la columna hasta donde está el error.

Sistema actual:

- **Ventajas:** Al mostrar la sentencia hasta el punto donde podría surgir el error, es más fácil de identificar en caso de que la sentencia se escribe directamente por entrada.
- **Desventajas:** La combinación de errores léxicos y sintácticos en un solo mensaje puede generar confusión. Además, si se manejan muchas sentencias simultáneamente, puede complicarse la identificación del error específico.

- **Implementación de predicados:** Las funciones como `syntax_check_redef`, `syntax_check_same_types`, `allowed_with_schemas` y `create_or_replace _view()` ya están presentes en DES. Sin embargo, no se han incluido en la nueva implementación porque no afectan directamente la salida del paso de análisis sintáctico.

- **Gestión de constantes temporales:** DES cuenta con un sistema detallado para gestionar constantes relacionadas con el tiempo (`DATE`, `TIME`, `DATETIME` y `TIMESTAMP`), que trata diferentes formatos y conversiones. Se ha implementado una versión simplificada, será necesario reemplazarla por la que se encuentra en DES.

- **Atributos exist_att y exist_table:** Aunque estos atributos ya están integrados, será necesario reemplazarlos por las versiones que ya existen en DES. Estos atributos, que indican la existencia de un atributo o tabla, son fundamentales para garantizar la integridad de las consultas y prevenir errores de referencia.

- **Separación de sentencias:** Como se mencionó anteriormente en sección 6.1, DES posee dos modos de entrada: `multiline_on` y `multiline_off`. En la actualidad, se asume que todas las entradas están en el modo `multiline_on`. Por tanto, se filtran primero los tokens(saltos de línea y comentarios) y, posteriormente, se separan las entradas por el token `punct(';')`. Para integrar en DES, en el caso del modo `multiline_off`, primero se debería dividir la lista de tokens por el carácter `punct(nl)` y, después, realizar el filtro.

- **Implementación de nuevo analizador sintáctico para otros lenguajes:** En futuros trabajos, uno de los objetivos principales será abordar la implementación separada del análisis léxico y sintáctico para los otros lenguajes que DES soporta.

- **Adaptación para SICStus Prolog:** Asegurar que todas las mejoras y modificaciones implementadas en este TFG sean plenamente operativas en SICStus Prolog. Esta adaptación permitirá a DES operar sobre diferentes plataformas de Prolog.

Este proyecto ha sido un esfuerzo significativo en la búsqueda de optimizar el rendimiento de DES, un sistema de bases de datos deductivas ya de por sí poderoso. Las mejoras que hemos logrado con nuestra nueva implementación tienen el potencial de tener un impacto sustancial en la eficiencia y la capacidad del sistema para manejar consultas complejas.

Introduction

Nowadays, database systems, as essential components of computing, have permeated every aspect of our lives, from online shopping, medical records, to social networks. All of these rely on the backbone of databases. DES (http://des.sourceforge.net/), as a deductive database system, can perform complex inferences on relational data and supports multiple query languages.

9.1. Motivation

Within the theme of language processors, various techniques and tools are explored. Among the most classic are the use of finite automata, context-free grammars, lexical and syntactic analyzers, among others. These methodologies have proven efficient in processing a wide variety of programming languages (see section 3.1).

Within these methodologies, Prolog's Definite Clause Grammars (DCG) have emerged as an essential tool, especially in systems like DES. DCGs are tools commonly used in Prolog for language processing. Unlike traditional techniques, DCGs provide a declarative representation of grammars, allowing for a more intuitive and concise definition (see the operation of DCGs in section 3.2.3).

However, despite the advantage of using DCG in DES, there's room for further improvements. With the constant growth in volume and complexity of data, database query efficiency has become a focal point. Even though DES has an edge in supporting multiple query languages, its language processor might face bottlenecks in performance when handling some complex queries. Optimizing its language processor is essential to fully harness the potential of DES.

9.2. Objectives

This project focuses on examining and improving DES's current processor. Our main objectives are:

- Design and develop a new SQL processor in Prolog that is not only more efficient but also more structured and maintainable. By carrying out the lexical analysis and syntactic analysis separately for the SQL language, the aim is not only exceptional performance but also more organized code. This separate structuring facilitates future changes, debugging, and the incorporation of new features, by providing a clear distinction between the different phases of analysis.

- Develop a test framework that allows comparing performance between the old processor and the newly developed one. This comparison will ensure that the results produced by the new processor are consistent and equivalent to those of the previous processor, thus validating its proper functionality.

9.3. Work Plan

During the project development, we will use SWI-Prolog, a popular implementation of the Prolog programming language (see section 2.2). Below is a detailed work plan:

1. **Familiarization with tools and techniques**:
   a) Get familiar with DES: A deductive database system that allows inferences on relational data and supports multiple query languages (see section 2.1).
   b) Explore the SWI-Prolog platform: Familiarize oneself with the development environment, its syntax, and key features (see section 2.2).
   c) Learn Prolog and DCG: Acquire programming skills in Prolog and use of DCG (Definite Clause Grammar) (see section 3.2.3).
2. **Processor Design**:
   a) Design the lexical analysis (lexer): Develop the lexical analyzer responsible for tokenizing the input.
   b) Design the syntactic analysis (parser): Create the syntactic analyzer that will construct the abstract syntax tree.
   c) Adjust the lexical analysis (lexer): Make adjustments and improvements to the lexical analysis based on the developed syntactic analyzer.
3. **Testing and error management**:
   a) Test the lexical analyzer (lexer): Validate the functionality of the lexical analysis.
   b) Test the syntactic analyzer (parser): Validate the functionality of the syntactic analysis.
4. **Testing**:
   a) Conduct general tests: Compare the results of the new processor with DES's current system.
   b) Compare execution times: Assess the efficiency of the new processor in comparison with the current processor.
5. **Documentation**:
   a) Compile the project report.

9.4. Structure of the Report

In this Bachelor's Thesis (TFG), the following chapters are presented and described below:

- **State of the Art:** This chapter provides a review of previous works and research related to DES, the platform where we program with Prolog (SWI-Prolog), and the importance of SQL.
- **Preliminary Concepts:** Fundamental concepts are introduced to understand the developed work.
  - *Language Processors:* The function of language processors, and the phases to follow.
  - *Prolog and Logical Programming:* An introduction to Prolog is offered, briefly describing its basic components, operation, and its application in language processing.
- **General Project Structure:** This chapter details the project's architecture, describing each of the generated .pl files and their specific function within SQL analysis using Prolog.
- **Lexical Analysis:** The first level of analysis, the lexical, is tackled, explaining its structure and the lexical categories considered in SQL. Furthermore, challenges encountered and the results obtained from this analysis are discussed.
- **Syntactic Analysis:** This chapter breaks down syntactic analysis, explaining how different SQL statements and structures are processed. The identification of various components, such as data types, constants, and conditions, among others, is discussed. Challenges and results of syntactic analysis are also addressed.
- **Error Management:** This chapter describes how error management is conducted during the analysis process, how errors are detected, and how they are reported to the user.
- **Tests and Comparisons:** This chapter presents the tests conducted and the comparison of execution times.
- **Conclusions and Future Work:** The work carried out is summarized, conclusions are presented, and potential future work is suggested.
- **Appendices:**
  - **A. Keywords in DES:** Enumeration of the keywords used in DES.
    - A.1. Commands: Listing of available commands.
    - A.2. Functions: The functions used.
    - A.3. Operators: The operators (Symbolic Operators, Comparison Operators, and Text Operators).
    - A.4. Punctuation Symbols: Punctuation signs.

- **B. Prolog code generation for keyword detection in the lexical analyzer (Python):** Explanation of the code generation process for keyword detection in the lexical analyzer using Python.
- **C. More examples of lexical analysis:** An expanded set of examples related to lexical analysis.
- **D. DES V6.7 SQL Grammar:** Detailed description of the specific SQL grammar for version V6.7 of DES.
- **E. More examples of error management in the syntactic analyzer:** Extended presentation of examples illustrating error handling in syntactic analysis.

Conclusions and future work

Table 8.1 showcases the substantial efficiency improvements offered by the new implementation compared to DES's current system. As SQL statements become more intricate, the difference in processing times further widens, decisively favoring our new solution. In performance terms, the new implementation outperforms the current system, particularly when handling complex SQL statements. The decision to separate lexical from syntactic analysis has brought clarity and organization to the project, simplifying future modifications and adjustments.

While all the anticipated objectives for the project have been achieved, additional tasks have been identified for future consideration. For upcoming work, the integration of the new implementation into DES is planned, along with the following proposed modifications and additions:

- **Error management:** Adjustments in error handling are needed since DES already has an error management system that differs from our recent implementation. Besides SQL, DES also processes other query languages, requiring the consideration of language-specific errors. Furthermore, the semantic errors that might arise during syntactic analysis are handled differently in DES, making it essential to address them accurately.

  Consider the following example:

  ```
  select 2 1 from t
  ```

  Our new implementation's response would be:

  ```
  Syntax Error:  Expected comma or FROM clause or end of SELECT statement
  at Line 1, Column 10.
  ```

  In contrast, the current system would provide:

  ```
  Error:  (SQL) Expected comma or (SQL) Expected FROM clause or (SQL)
  Expected end of SELECT statement or (RA) Expected NOT or (RA) Expected
  string type in 2 or (RA) Expected comparison operator near "select 2 "
  ```

  At first glance, the new implementation precisely indicates the line and column of the error, focusing solely on potential SQL errors. Conversely, the current system merges lexical and syntactic errors into one message and also displays the statement up to where the error might arise.

  New implementation:

- **Advantages:** Clearly indicates the error's line and column, easing its identification, especially if the statement comes from a file.
- **Disadvantages:** If the statement is input directly, pinpointing the exact error location might be challenging, requiring column counting to find the error.

Current system:

- **Advantages:** Displaying the statement up to the error point aids identification when the statement is directly entered.
- **Disadvantages:** Merging lexical and syntactic errors into one message can be confusing. Handling multiple statements can further obscure specific error identification.

- **Implementation of predicates:** Functions like `syntax_check_redef`, `syntax_check_same_types`, `allowed_with_schemas`, and `create_or_replace_view()` are already present in DES. They haven't been included in the new implementation since they don't directly impact the syntactic analysis output.

- **Temporal constants management:** DES has a detailed system for managing time-related constants (`DATE`, `TIME`, `DATETIME`, and `TIMESTAMP`), which deals with various formats and conversions. A simplified version has been implemented, necessitating its replacement with DES's version.

- **Attributes exist_att and exist_table:** While these attributes are integrated, they need replacement with the versions available in DES. These attributes, indicating the existence of an attribute or table, are vital to ensuring query integrity and preventing reference errors.

- **Statement separation:** As previously mentioned in section 6.1, DES has two input modes: `multiline_on` and `multiline_off`. Currently, all inputs are assumed to be in the `multiline_on` mode. Thus, tokens (line breaks and comments) are filtered first, and inputs are then separated by the `punct(';')` token. For DES integration, in the `multiline_off` mode, the token list should first be split by the `punct(nl)` character, followed by filtering.

- **Implementation of a new syntactic analyzer for other languages:** In future work, one of the main objectives will be to address the separate implementation of lexical and syntactic analysis for the other languages that DES supports.

- **Adaptation for SICStus Prolog:** Ensure that all enhancements and modifications implemented in this TFG are fully operational in SICStus Prolog. This adaptation will allow DES to operate on different Prolog platforms.

This project has been a considerable endeavor towards enhancing DES's performance, an inherently potent deductive database system. The advancements achieved with our new implementation have the potential to considerably impact the system's efficiency and its capability to manage intricate queries.

Bibliografía

Datalog educational system (des). `https://des.sourceforge.io/index.html`, 2023. Accedido el 26-07-2023.

Swi-prolog. `https://www.swi-prolog.org/`, 2023. Accedido el 27-07-2023.

Aho, A. V., Lam, M. S., Sethi, R. y Ullman, J. D. *Compilers: principles, techniques, and tools*. Addison-Wesley, 2006.

Apt, K. R. *From Logic Programming to Prolog*. Prentice Hall, 1st edición, 1996. ISBN 9780132303682. A unique publication that provides an introduction to the theory of logic programming and its application to Prolog programs. Covers programming issues such as termination, occur-check freedom, partial correctness and absence of runtime errors.

Bratko, I. *Prolog programming for artificial intelligence (4th edition)*. Pearson, 2011.

Melton, J. y Simon, A. *SQL:1999: Understanding Relational Language Components*. Morgan Kaufmann, 2002.

O'Keefe, R. A. *The Craft of Prolog*. MIT Press, 1990.

Pereira, F. C. N. y Shieber, S. M. *Prolog and Natural-Language Analysis*. Microtome, digital edition, revision of october 5, 2005 edición, 2005. Hardbound edition available with ISBN 0-9719777-0-4.

Scott, M. L. *Programming language pragmatics*. Morgan Kaufmann, 2009.

Sterling, L. y Shapiro, E. Y. *The Art of Prolog (2nd Edition): Advanced Programming Techniques*. The MIT Press, 1994.

Sáenz-Pérez, F. *Datalog Educational System V6.7: User's Manual*. Formal Analysis and Design of Software Systems (FADoSS), Departamento de Ingeniería del Software e Inteligencia Artificial (DISIA), Universidad Complutense de Madrid (UCM), 2021a. Sección 4.2 SQL.

Sáenz-Pérez, F. *Datalog Educational System V6.7: User's Manual*. Formal Analysis and Design of Software Systems (FADoSS), Departamento de Ingeniería del Software e Inteligencia Artificial (DISIA), Universidad Complutense de Madrid (UCM), 2021b. Sección 4.2.13 SQL Grammar.

Van Roy, P. L. Extended dcg notation: A tool for applicative programming in prolog. Informe Técnico UCB/CSD-90-583, EECS Department, University of California, Berkeley, 1990.

Apéndice A

Palabras claves en DES

A.1. Comandos

add	all	alter	any
ascending	asc	assume	as
between	bc	by	candidate
cascade	character	char	check
column	commit	constraints	constraint
create	databases	database	data
datetime	date	decimal	default
delete	descending	desc	describe
determined	distinct	division	drop
else	end	escape	except
exists	extract	false	fetch
first	foreign	from	full
group	having	if	inner
insert	intersect	into	integer
int	in	is	join
key	like	limit	minus
natural	no	null	number
numeric	offset	only	on
order	outer	primary	real
recursive	references	rename	restrict
rollback	rows	savepoint	select

set	show	smallint	some
string	tables	table	text
then	timestamp	time	type
top	to	true	union
unique	update	using	values
varchar2	varchar	views	view
when	where	with	work

A.2. Funciones

sqrt	ln	log	exp
sin	cos	tan	cot
asin	acos	atan	acot
abs	mod	float	integer
sign	gcd	min	max
truncate	trunc	float_integer_part	float_fractional_part
round	floor	ceiling	rand
power	avg	avg_distinct	count
count_distinct	sum	sum_distinct	times
times_distinct	pi	e	length
concat	instr	left	lower
lpad	ltrim	replace	repeat
reverse	rpad	right	rtrim
space	substr	trim	upper
year	month	day	hour
minute	second	last_day	to_char
to_date	sysdate	current_date	current_time
current_datetime	datetime_add	datetime_sub	add_months
cast	coalesce	greatest	least
nvl	nvl2	nullif	iif
case			

A.3. Operadores

A.3.1. Operadores simbólicos

`<<`	`>>`	`\/`	`+`
`/\`	`//`	`/`	`^`
`|`	`**`	`*`	`-`

A.3.2. Operadores de comparación

`!=`	`=`	`>=`	`<=`
`<>`	`>`	`<`	

A.3.3. Operadores textuales

and	or	not	xor
rem	div		

A.4. Símbolos de puntuación

`(`	`)`	`,`	`.`
`;`	`::`	`:`	`nl (\n)`

Apéndice B

Generación de código Prolog para detección palabras claves en el analizador léxico (Python)

```
template = "command('{0}'/Original) -->> {1}lc(\"{0}\",
   Original),{2}not_more_char, !, add_col({3})."

# Encuentra la longitud del comando mas largo para alinear.

max_len = max([len(comando) for comando in lista_comandos])

for comando in lista_comandos:
    # para alinear 'lc'
    num_spaces_lc = max_len - len(comando) + 2

    # para alinear 'not_more_char'.
    num_spaces_not_more_char = max_len - len(comando) + 2

    # Crea el codigo Prolog.
    prolog_code = template.format(comando, ' ' *
       num_spaces_lc, ' ' * num_spaces_not_more_char, len(
       comando))
    print(prolog_code)
```

Listado B.1: Generación de código Prolog para detección de comandos

Para las funciones solo hay que cambiar la plantilla. Igual para operadores textuales solo cambiar la palabra por: `textual_operator`.

```
template2 = "function('{0}'/Original) -->>{1}lc(\"{0}\",
   Original),{2}not_more_char, !, add_col({3})."
```

Listado B.2: Plantilla de generación de código Prolog para funciones

Apéndice C

Más ejemplos de análisis léxico

- **Ejemplo IV**

```
1 INSERT INTO t2 VALUES (TIME '12:00:01', -2.5)
```

Listado C.1: Entrada (lexer) (Ejemplo IV)

```
[cmd(insert/‘INSERT’):pos(1,1),
cmd(into/‘INTO’):pos(1,8),
id(t2/l):pos(1,13),
cmd(values/‘VALUES’):pos(1,16),
punct(‘(’):pos(1,23),
cmd(time/‘TIME’):pos(1,24),
str(‘12:00:01’):pos(1,29),
punct(‘,’):pos(1,39),
op(-):pos(1,41),
frac(2,5):pos(1,42),
punct(‘)’):pos(1,45)]
```

Listado C.2: Salida (lexer) (Ejemplo IV)

- **Ejemplo V**

```
1 Select count(*) from t group by a
```

Listado C.3: Entrada (lexer) (Ejemplo V)

```
[cmd(select/‘Select’):pos(1,1),
fn(count/count):pos(1,8),
punct(‘(’):pos(1,13),
op(*):pos(1,14),
punct(‘)’):pos(1,15),
cmd(from/from):pos(1,17),
id(t/l):pos(1,22),
cmd(group/group):pos(1,24),
cmd(by/by):pos(1,30),
id(a/l):pos(1,33)]
```

Listado C.4: Salida (lexer) (Ejemplo V)

Apéndice D

Gramática SQL de DES V6.7

Esta gramática sigue una sintaxis de tipo EBNF. (Sáenz-Pérez, 2021b)

```
SQLstmt ::=
 DDLstmt[;]
 |
 DMLstmt[;]
 |
 DQLstmt[;]
 |
 ISLstmt[;]
 |
 TMLstmt[;]
```

1. Sentencias DDL(Lenguaje de Definición de Datos)

```
 DDLstmt ::=
 CREATE [OR REPLACE] TABLE CompleteConstrainedSchema
 |
 CREATE [OR REPLACE] TABLE TableName [(] LIKE TableName [)]
 |
 CREATE [OR REPLACE] TABLE TableName [(] AS DQLstmt [)]
 |
 CREATE [OR REPLACE] VIEW Schema AS DQLstmt
 |
 CREATE DATABASE DatabaseName % Unsupported up to now
 |
 ALTER TABLE TableName [ADD|DROP] | [[COLUMN] Att |
    CONSTRAINT
[ConstraintName] TableConstraint]
 |
 ALTER TABLE TableName ALTER [COLUMN] Att [AttDefinition |
    SET
[DATA] TYPE Type]
 |
 RENAME TABLE TableName TO TableName
```

```
 |
 RENAME VIEW ViewName TO ViewName
 |
 DROP TABLE DropTableClauses TableName{,TableName}
DropTableClauses % Extended syntax following MySQL, SQL
   Server and others
 |
 DROP VIEW DropViewClauses ViewName DropViewClauses
 |
 DROP DATABASE [DatabaseName]
 |
 CompleteSchema := DQLstmt % Addition to support HR-SQL
    syntax

DropTableClauses ::=
 [IF EXISTS] [CASCADE [CONSTRAINTS]]
DropViewClauses ::=
 [IF EXISTS] [CASCADE]
Schema ::=
 RelationName
 |
 RelationName(Att,...,Att)

CompleteConstrainedSchema ::=
 RelationName(AttDefinition {,AttDefinition} [,
TableConstraintDefinitions])

AttDefinition ::=
 Att Type [ColumnConstraintDefinition {ColumnConstraint}]
CompleteSchema ::=
 RelationName(Att Type {,...,Att Type})
Type ::=
 CHAR(n) % Fixed-length string of n characters
 |
 CHARACTER(n) % Equivalent to CHAR(n)
 |
 CHAR % Fixed-length string of 1 character
 |
 VARCHAR(n) % Variable-length string of up to n characters
 |
 VARCHAR2(n) % Oracle's variable-length string of up to n
characters
 |
 TEXT(n) % MS Access' variable-length string of up to n
characters
 |
 VARCHAR % Variable-length string of up to the maximum length
of the underlying Prolog atom
```

```
 |
 STRING % Equivalent to VARCHAR
 |
% CHARACTER VARYING(n) % Equivalent to the former
% |
 INT
 |
 INTEGER % Equivalent to INT
 |
 SMALLINT
 |
 NUMERIC(p,d) % A total of p digits, where d of those are in
    the decimal place
 |
 NUMERIC(p) % An integer with a total of p digits
 |
 NUMERIC % An integer
 |
 DECIMAL(p,d) % Synonymous for NUMERIC
 |
 DECIMAL(p) % Synonymous for NUMERIC
 |
 DECIMAL % Synonymous for NUMERIC
 |
 NUMBER(p,d) % Synonymous for NUMERIC. For supporting Oracle
NUMBER
 |
 NUMBER(p) % Synonymous for NUMERIC
 |
 NUMBER % Synonymous for NUMERIC
 |
 REAL
 |
 FLOAT % Synonymous for REAL
% |
% DOUBLE PRECISION % Equivalent to FLOAT
% |
 FLOAT(p) % FLOAT with precision of at least p digits
 |
 DECIMAL % Synonymous for REAL (added to support DECIMAL
    LogiQL
Type). Not SQL standard
 |
 DATE % Year, month and day
 |
 TIME % Hours, minutes and seconds
 |
 TIMESTAMP % Combination of date and time
```

```
ConstraintNameDefinition ::=
 CONSTRAINT ConstraintName
ColumnConstraintDefinition ::=
 [ConstraintNameDefinition] ColumnConstraint
ColumnConstraint ::=
 [NOT] NULL % NULL is not in the standard
 |
 PRIMARY KEY
 |
 UNIQUE
 |
 CANDIDATE KEY % Not in the standard, but
supported in DB2 for functional dependencies
 |
 REFERENCES TableName[(Att)]
 |
 DEFAULT Expression
 |
 CHECK CheckConstraint

TableConstraintDefinitions ::=
 TableConstraintDefinition{,TableConstraintDefinition}
TableConstraintDefinition ::=
 [ConstraintNameDefinition] TableConstraint
TableConstraint ::=
 NOT NULL Att % Not in the standard
 |
 UNIQUE (Att {,Att})
 |
 CANDIDATE KEY (Att {,Att}) % Not in the standard
 |
 PRIMARY KEY (Att {,Att})
 |
 FOREIGN KEY (Att {,Att}) REFERENCES TableName[(Att {,Att})]
 |
 CHECK CheckConstraint

CheckConstraint ::=
 WhereCondition
 |
 (Att {,Att}) DETERMINED BY (Att {,Att}) % Not in the
    standard, but supported in DB2 for functional
    dependencies
RelationName is a user identifier for naming tables, views
   and aliases
TableName is a user identifier for naming tables
ViewName is a user identifier for naming views
Att is a user identifier for naming relation attributes
```

2. Sentencias DML(Lenguaje de Manipulación de Datos)

```
DMLstmt ::=
 INSERT INTO TableName[(Att {,Att})] VALUES (ExprDef
{,ExprDef}) {, (ExprDef {,ExprDef})}
|
 INSERT INTO TableName DEFAULT VALUES
 |
 INSERT INTO TableName[(Att {,Att})] DQLstmt
 |
 DELETE FROM TableName [[AS] Identifier] [WHERE Condition]
 |
 UPDATE TableName [[AS] Identifier] SET Att=Expr {,Att=Expr}
[WHERE Condition]
% ExprDef is either a constant or the keyword DEFAULT
```

3. Sentencias DQL(Lenguaje de Consulta de Datos)

```
DQLstmt ::=
 (DQLstmt)
 |
 UBSQL
UBSQL ::=
 SELECTstmt
 |
 DQLstmt UNION [ALL|DISTINCT] DQLstmt
 |
 DQLstmt EXCEPT [ALL|DISTINCT] DQLstmt
 |
 DQLstmt MINUS [ALL|DISTINCT] DQLstmt
 |
 DQLstmt INTERSECT [ALL|DISTINCT] DQLstmt
 |
 WITH LocalViewDefinition {,LocalViewDefinition} DQLstmt
 |
 ASSUME LocalAssumption {,LocalAssumption} DQLstmt % Not in
    the standard
LocalViewDefinition ::=
 [RECURSIVE] Schema AS DQLstmt
 |
 [RECURSIVE] DQLstmt NOT IN Schema
LocalAssumption ::=
 DQLstmt [NOT] IN Schema
SELECTstmt ::=
 SELECT [TOP Integer] [[ALL|DISTINCT]] SelectExpressionList
 [INTO SelectTargetList]
 [FROM Rels
 [WHERE WhereCondition]
```

```
 [GROUP BY Atts]
 [HAVING HavingCondition]
 [ORDER BY OrderDescription]
 [OFFSET Integer [LIMIT Integer]]
 [FETCH FIRST Integer ROWS ONLY]]
Atts ::=
 Att {,Att}
OrderDescription ::=
 Att [OrderDirection] {,Att [OrderDirection]}
OrderDirection ::=
 ASC|DESC|ASCENDING|DESCENDING
SelectExpressionList ::=
 *
 |
 SelectExpression {,SelectExpression}
SelectExpression ::=
 UnrenamedSelectExpression
 |
 RenamedExpression
UnrenamedSelectExpression ::=
 Att
 |
 RelationName.Att
 |
 RelationName.*
 |
 Expression
 |
 DQLstmt
RenamedExpression ::=
 UnrenamedExpression [AS] Identifier
Expression ::=
 Op1 Expression
 |
 Expression Op2 Expression
 |
 Function(Expression{, Expression})
 |
 Att
 |
 RelationName.Att
 |
 Cte
 |
 DQLstmt
Op1 ::=
 - | \
Op2 ::=
```

```
 ^ | ** | * | / | // | rem | \/ | # | + | - | /\ | << | >> |
    div
Function ::=
 sqrt/1 | ln/1 | log/1 | log/2 | sin/1 | cos/1 | tan/1 | cot
    /1 | asin/1 | acos/1 | atan/1 | acot/1 | abs/1 | power/2
    | exp/1 | float/1 | integer/1 | sign/1 | gcd/2 | min/2 |
    max/2 | mod/2 | trunc/1 | truncate/1 | trunc/2 | truncate
    /2 | float_integer_part/1 | float_fractional_part/1 |
    round/1 | round/2 | floor/1 | ceiling/1 | rand/1 | rand/2
     | concat/2 | length/1 | like-escape | lower/1 | lpad/2 |
     lpad/3 | rpad/2 | rpad/3 | instr/2 | replace/3 | reverse
    /1 | substr/3 | upper/1 | left/2 | ltrim/1 | rtrim/1 |
    trim/1 | repeat/2 | right/2 | space/1 | year/1 | month/1
    | day/1 | hour/1 | minute/1 | second/1 | datetime_add/2 |
     datetime_sub/2 | add_months/2 | current_time/0 |
    current_date/0 | current_datetime/0 | sysdate/0 | extract
    -from | to_char/1 | to_char/2 | to_date/1 | to_date/2 |
    cast/2 | coalesce/N | greatest/N | iif/3 | least/N | nvl
    /2 | nvl2/3 | nullif/2 | case-when-then-end
SelectTargetList ::=
 HostVariable {, HostVariable}
% Aggregate Functions:
% The argument may include a prefix "distinct" for all but "
   min" and "max":
% avg/1 | count/1 | count/0 | max/1 | min/1 | sum/1 | times/1
ArithmeticConstant ::=
 pi | e
Rels ::=
 Rel {,Rel}
Rel ::=
 UnrenamedRel
 |
 RenamedRel
UnrenamedRel ::=
 TableName
 |
 ViewName
 |
 DQLstmt
 |
 JoinRel
 |
 DivRel
RenamedRel ::=
 UnrenamedRel [AS] Identifier
JoinRel ::=
 Rel [NATURAL] JoinOp Rel [JoinCondition]
JoinOp ::=
```

```
 INNER JOIN
 |
 LEFT [OUTER] JOIN
 |
 RIGHT [OUTER] JOIN
 |
 FULL [OUTER] JOIN
JoinCondition ::=
 ON WhereCondition
 |
 USING (Atts)
DivRel ::=
 Rel DIVISION Rel % Not in the standard
WhereCondition ::=
 BWhereCondition
 |
 UBWhereCondition
HavingCondition
 % As WhereCondition, but including aggregate functions
BWhereCondition ::=
 (WhereCondition)
UBWhereCondition ::=
 TRUE
 |
 FALSE
 |
 EXISTS DQLstmt
 |
 NOT (WhereCondition)
 |
 (AttOrCte{,AttOrCte}) [NOT] IN [DQLstmt|(Cte{,Cte})|((Cte{,
    Cte}){,(Cte{,Cte})})] % Extension for lists of tuples
 |
 WhereExpression IS [NOT] NULL
 |
 WhereExpression [NOT] IN DQLstmt
 |
 WhereExpression ComparisonOp [[ALL|ANY]] WhereExpression
 |
 WhereCondition [AND|OR|XOR] WhereCondition
 |
 WhereExpression BETWEEN WhereExpression AND WhereExpression
WhereExpression ::=
 Att
 |
 Cte
 |
 Expression
```

```
 |
 DQLstmt
AggrExpression ::=
 [AVG|MIN|MAX|SUM]([DISTINCT] Att)
 |
 COUNT([*|[DISTINCT] Att])
AttOrCte ::=
 Att
 |
 Cte
ComparisonOp ::=
 = | <> | != | < | > | >= | <=
Cte ::=
 Number
 |
 'String'
 |
 DATE 'String' % String in format '[BC] Int-Int-Int'
 |
 TIME 'String' % String in format 'Int:Int:Int'
 |
 TIMESTAMP 'String' % String in format '[BC] Int-Int-Int Int:
    Int:Int'
 |
 NULL
% Number is an integer or floating-point number
% Int is an integer number
```

4. Sentencias ISL(Lenguaje del Esquema de Información)

```
ISLstmt ::=
 SHOW TABLES
 |
 SHOW VIEWS
 |
 SHOW DATABASES
 |
 DESCRIBE [TableName|ViewName]
```

5. Sentencias TML(Lenguaje de Gestión de Transacciones)

```
TMLstmt ::=
 COMMIT [WORK]
 |
 ROLLBACK [WORK] [TO SAVEPOINT SavepointName]
 |
 SAVEPOINT SavepointName
```

Apéndice E

Más ejemplos de gestión de errores en el analizador sintáctico

Entrada	**Syntax Error: Expected ...**	**at Posición (Line, Column)**
create table t(a)	valid type	1, 17
create or replace table t(a char())	a positive integer	1, 34
create table t(a) as	valid SQL DQL statement (SELECT, WITH or ASSUME)	the end of the program
create or replace table t(a int references s a)	valid column constraint (NOT, NULL, PRIMARY, UNIQUE, REFERENCES, DEFAULT, CHECK, CANDIDATE, DETERMINED)	1, 46
create table t like sa)	opening parenthesis "(" not found before	1, 23
create view v(a) s	AS	1, 18
create view v() as select * from a	column sequence separated by commas	1, 15
my_view(age int) : SELECT age FROM my_table	equals "="	1, 20
insert into t2 values(1 '2')	comma or closing parenthesis ")"	1, 25

Tabla E.1: Otros errores sintácticos (I)

Entrada	**Syntax Error: Expected ...**	**at Posición (Line, Column)**
insert into t values 1	opening parenthesis "("	1, 22
update t set a=1, w=	an expression	the end of the program
update t set a=1 where	valid WHERE condition	the end of the program
select extract(hur from time '22:05:31')	valid datetime field (year, month, day, hour, minute, second)	1, 16
select extract(hour from 3)	valid datetime expression	1, 26
select case when 1=1 then 'a' else 'b'	END	the end of the program
select 2 into V	a word starting with a lowercase letter or within single quotes	1, 15
update table set a=1	a different word after UPDATE, TABLE is not allowed	1, 8

Tabla E.2: Otros errores sintácticos (II)