Instituto Tecnológico de Buenos Aires

Trabajo Práctico Especial - PiBot

72.39 Autómatas, Teoría de Lenguajes y Compiladores

Integrantes:

Aramburu, Paz - 62556

Neimark, Luciano - 62086

Tordomar, Nicolás - 62250

Fecha de entrega: 23 de noviembre de 2023

Tabla de contenidos

1. Introducción

El objetivo del trabajo consiste en la elaboración de un compilador que posea la capacidad de transformar programas provenientes de un subconjunto específico del lenguaje SQL hacia su equivalente en álgebra relacional. Este lenguaje diseñado permitirá la recepción de archivos con extensión .sql para posteriormente generar un documento en formato LaTeX que exhiba la fórmula correspondiente en álgebra relacional. Es una idea sumamente innovadora y estimulante, dado que refleja el funcionamiento interno fundamental de un motor de bases de datos relacionales. Esta habilidad para traducir las consultas SQL en álgebra relacional destaca una gran relevancia práctica en el contexto de la gestión y manipulación eficiente de bases de datos (y también para aquellos que estén cursando la materia Base de Datos I).

2. Desarrollo del proyecto

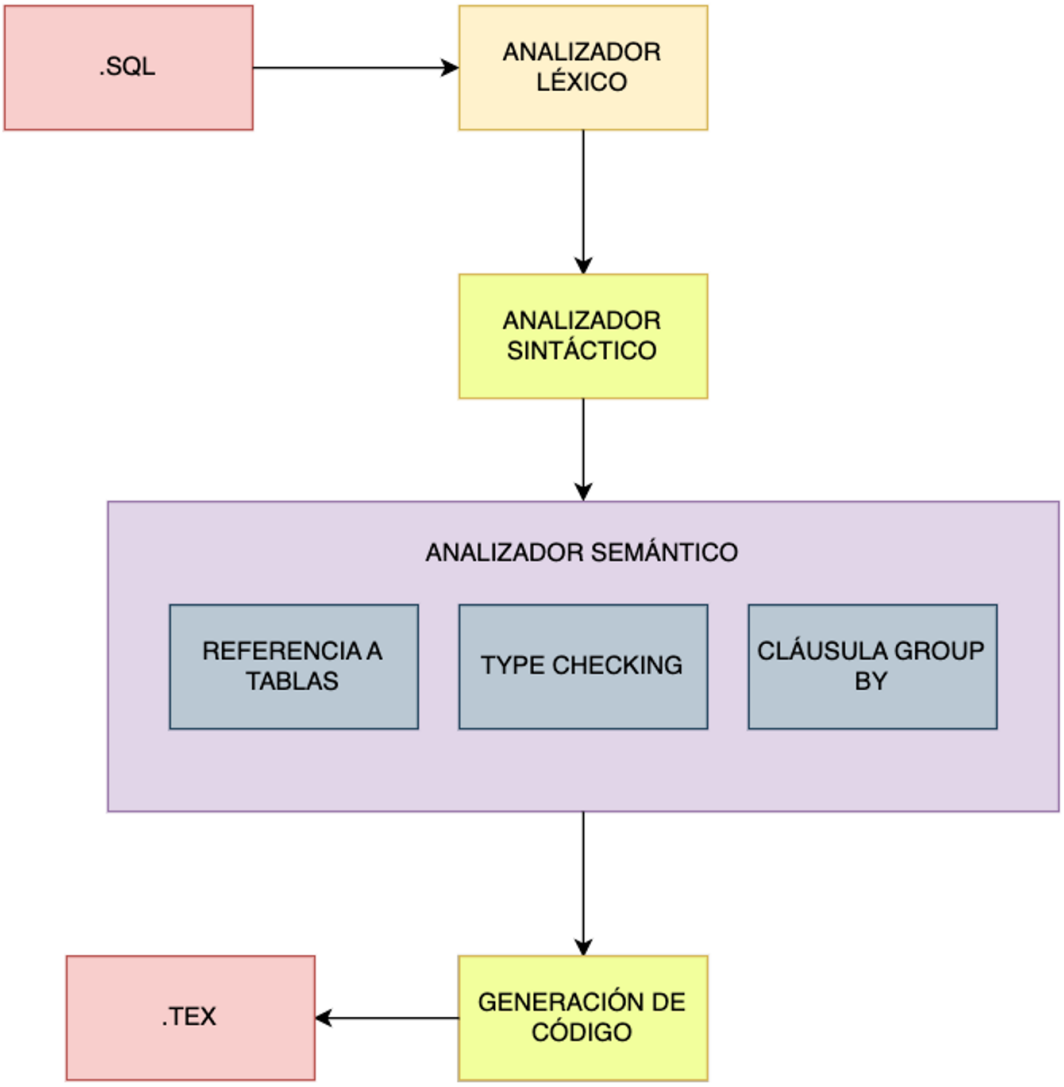

Figura 1: Flujo del desarrollo del lenguaje y compilador

En la figura 1 se puede observar el flujo de trabajo para generar el lenguaje y el compilador. Las primeras dos fases corresponden a la sección frontend, mientras que las otras dos corresponden a la sección backend.

2.1 Frontend

En el desarrollo del frontend, se llevaron a cabo dos fases distintas: en primer lugar, se procedió con la construcción del alfabeto a ser empleado, seguido por la definición de la gramática. En líneas generales, no hubo mayores dificultades en esta instancia, puesto a que al ser SQL un lenguaje preexistente, su gramática y alfabeto ya se encuentran documentados, y muy bien descritos.

Durante la elaboración de la gramática, surgieron conflictos del tipo *"reduce-reduce"*. Los problemas fueron solucionados al reconsiderar y ajustar su estructura. Asimismo, se eligió qué se iba a aceptar y qué no dentro de las consultas SQL, debido a que tomar el lenguaje SQL entero iba a ser muy complejo. Se limitó el lenguaje para garantizar un desarrollo exitoso. A partir de esto se realizaron los casos de aceptación y los casos de rechazo dentro de los testeos, los cuales se basaban en la gramática definida.

2.2 Backend

El desarrollo del backend fue un poco más dificultoso. El primer paso fue generar un árbol de sintaxis abstracta. En este árbol se crearon nodos representando las partes del programa. El nodo principal es Program, el cual contiene los seis tipos de cláusulas. El nodo de cada cláusula varía ya que sus gramáticas son distintas. Por ejemplo, en SELECT se declaran columnas y el FROM se declaran tablas. Tener el programa en el árbol facilitó mucho las siguientes etapas. El siguiente paso fue generar las bison actions, cuyo objetivo es popular el árbol a partir de la gramática y, más adelante, construir la tabla de símbolos.

Una vez terminado el árbol, se avanzó con el análisis semántico. En este paso se verificó que:

1. Para toda referencia a una tabla del estilo table.column_i, esa tabla "table" pertenezca a la FROM clause.
2. En caso de que incluya la cláusula GROUP BY, se debe verificar que toda columna que esté en el SELECT, también esté en el GROUP BY.
3. Validar los tipos de datos de cada columna.

Para garantizar la referencia a las tablas, se creó una lista que se popula con todas las tablas de la FROM clause, o con su alias. Para todas las demás partes de la query (WHERE,

GROUP BY, HAVING, ORDER BY), se valida la referencia a la tabla de manera dinámica. Esto evita iterar al final por todo el árbol buscando las referencias a la tabla. El manejo de la cláusula SELECT representa una situación particular dentro del programa, ya que al momento de validar el SELECT, la lista que contiene las tablas no está completa. Por ende, es necesario llevar a cabo la validación del SELECT una vez que se haya finalizado la construcción completa del árbol del programa.

En segundo lugar, se tuvo que verificar que si existe la cláusula GROUP BY, entonces toda columna referenciada en el SELECT tiene que estar en el GROUP BY. Esto también se debe verificar una vez terminado el árbol, ya que el SELECT es la primera cláusula que se encuentra.

La validación del tipo de dato de las columnas es probablemente el desafío más grande del analizador semántico. Como las tablas que se están consultando no existen en una base de datos, no es posible saber qué tipo de dato tiene cada columna. Existían dos posibles caminos. Primer camino: no hacer nada. Segundo camino: resolver dinámicamente el tipo de dato de cada columna. No hacer nada no era una opción.

Por lo tanto, se construyó un mapa donde la clave era una estructura que tenía: como clave el nombre de la columna y el nombre de la tabla en caso de que se haga referencia a ella (table.column_i), y como valor el tipo de dato de la columna. Para el caso en el que haya dos columnas con el mismo nombre pero de distintas tablas es necesario tener el nombre de la tabla, puesto a que table1.alias puede ser de tipo INT, y table2.alias puede ser de tipo TEXT. Lo mismo ocurre en cualquier consulta SQL, si se referencian en el FROM dos tablas que contienen columnas con el mismo nombre, es necesario especificar el nombre de la tabla para referenciar esa columna. De lo contrario, se produce un error de ambigüedad.

La tabla de símbolos se va generando dinámicamente en la cláusula WHERE. Cada vez que se compara una columna con un valor, y esa columna no pertenece a la tabla de símbolos, se agrega en ella el nombre de la columna (y la tabla si es referenciada), seguido del tipo de dato al que se está comparando. Por ejemplo, si se hace un id > 9, se agrega que la columna id es de tipo INT. Pero si se hiciese un id = “nombre”, se especifica en la tabla que la columna id es de tipo STRING.

En caso de que se intente hacer una comparación, y la columna ya exista en la tabla de símbolos con su respectivo tipo de dato, se valida que el tipo de dato de la comparación sea el mismo que el de la tabla de símbolos.

En caso de que se haga una comparación entre dos columnas, como por ejemplo: name = user_name, hay 3 casos posibles:

1. Ninguna de las columnas tiene tipo asignado en la tabla de símbolos → no se hace nada.
2. Solo una de ellas tiene tipo asignado → a la otra se le asigna el mismo tipo que ella.
3. Las dos tienen tipo asignado → se pregunta si son del mismo tipo. De no serlo, se produce un error.

Es importante que el usuario tenga una experiencia óptima usando el compilador. En el caso de que surjan errores, se espera que el programa no se detenga con el primer inconveniente, sino que sea capaz de terminar de compilar y devolver todos los errores identificados. Esta característica se puede encontrar con la funcionalidad de PiBot. Se implementó un arreglo que registra todos los errores detectados durante el proceso, para posteriormente presentarlos a través de la salida estándar (stdout).

Finalmente, para asegurar liberar toda la memoria reservada, se creó una librería que hace un wrapper de las funciones *malloc* y *calloc*, y guarda en una lista todos los punteros que se hayan pedido. Al finalizar el programa, libera todos los punteros evitando así *memory leaks*.

3. Dificultades encontradas

Una de las dificultades más grandes fue implementar el type checking. Al no tener la declaración de las columnas, hubo que averiguar los tipos de datos de manera dinámica, lo que conllevo a una lógica interesante y distinta a la que se vio en clase. Puesto a que en nuestro caso no teníamos la declaración de las tablas, en ningún momento es conocido el tipo de dato de cada columna.

Otra de las dificultades que encontramos fue el caso de los operadores IN y ALL. El álgebra relacional puro no tiene operadores análogos a ellos. Es decir, σ es un operador que verifica 1:N. Sin embargo, los operadores IN y ALL son de tipo N:M. Por cuestiones de tiempo y complejidad, se decidió utilizar un álgebra relacional extendido donde σ pueda hacer comparaciones de orden N:M.

4. Futuras extensiones

La primera extensión a desarrollar sería aceptar todo el lenguaje de SQL. Estamos satisfechos con toda la cantidad de SQL que aceptamos. Sin embargo, resultaría interesante incorporar operadores como UNION, los cuales implementan una lógica distinta a la actual. A su vez, se podría incorporar la función CAST, lo cual implicaría reinventar el "*type checking"*.

Otra extensión pensada es hacer la transformación de álgebra relacional a SQL. Resulta desafiante, ya que habría que pensar cómo recibir el input del álgebra relacional, y como hacer el camino inverso. Creemos que este producto puede ser beneficioso para aquellas personas que están aprendiendo SQL mediante álgebra relacional, para empezar a ver las transformaciones entre álgebra y SQL.

Por otro lado, otra funcionalidad que quedó pendiente fue la optimización de código. En SQL hay muchas maneras de hacer una query. Por lo tanto, para optimizar la respuesta es necesario primero optimizar el código de entrada.

Finalmente, una extensión que consideramos necesaria es integrar el compilado del LaTeX al proyecto, y que el retorno del programa sea un .pdf que contenga el LaTeX compilado.

5. Conclusión

Habiendo terminado el proyecto, estamos sorprendidos con lo que hemos hecho. Desde que estamos en primer año que vemos lenguajes de programación y sus respectivos compiladores. Haber diseñado y compilado un lenguaje como SQL nos parece un gran logro. No solo por el manejo que obtuvimos sobre flex y bison y el aprendizaje acerca de los compiladores, sino que también por nuevas cosas que aprendimos acerca de C y LaTeX.

Terminamos este trabajo con un gran entendimiento del tema, y mucha motivación por haber conocido estás herramientas que estamos seguros que nos servirán para el futuro. Asimismo, aunque nunca fue el objetivo de esta materia, nos vamos con mucho conocimiento nuevo sobre el lenguaje SQL y el álgebra relacional, lo cual también nos va a ser muy útil para los próximos años.

6. Bibliografía

[1] Overleaf. (2023). Overleaf latex documentation. Recuperado de https://www.overleaf.com/learn

[2] PostgreSQL Global Development Group. (2023). PostgreSQL SQL Syntax. Recuperado de https://www.postgresql.org/docs/8.1/sql-syntax.html