Manual técnico

Pasos para realizar el interprete

1. Generar una gramática por medio de Jison que reconozca el lenguaje.
2. Armar un AST(Arbol de sintaxis abstracta) para cada una de las sentencias/instrucciones en el archivo fuente por medio de las acciones de Jison.
3. Interpretar cada sentencia/instrucción de forma recursiva.

1. Generar una gramática por medio de Jison

- Se puede escribir la gramática para el analizador léxico y sintáctico por separado
- se utiliza el siguiente comando para compilar ambos en un solo script de JS:

```
#jison <ruta_sintactico> <ruta_lexico> -o <ruta_para_guardar_script_generado>
#ejemplo:
jison src/parser/parser.jison src/parser/lexer.flex -o src/parser/query_parser.cjs
```

1.1 Estructura archivo Léxico

La sintaxis para el archivo léxico es de la siguiente forma:

```
%options <opciones>

%%

<regex>             (return '<nombre_terminal>' | /* comentario */)
<regex>             (return '<nombre_terminal>' | /* comentario */)
<regex>             (return '<nombre_terminal>' | /* comentario */)
<<EOF>>             return 'EOF'
.                   return 'INVALID'

%%

<codigo javascript personalizado>
```

Ejemplo de archivo léxico

```
%options flex case-insensitive

%%

\s+                 /* no hacer nada */
"--".*              /* no hacer nada */
"true"              return 'TRUE'
"false"             return 'FALSE'
[a-z][a-z0-9_-]*    return 'ID'
<<EOF>>             return 'EOF'
.                   return 'INVALID'
```

Notas de archivo léxico

- Para expresiones regulares de cosas que queremos ignorar (ej. comentarios, espacios en blanco), se debe escribir un comentario en lugar de devolver un nombre de token. De lo contrario, el arhivo no podrá ser compilado por Jison
- Las producciones:

```
<<EOF>>               return 'EOF'
.                     return 'INVALID'
```

son obligatorias

- El último par de %% puede ser obviado si no se piensa escribir código personalizado. Personalmente yo lo uso para poner comentarios que muestran de donde saqué algunas regex en internet.

2. Armar un AST para cada sentencia

2.1 Estructura de un archivo fuente

Un archivo fuente es simplemente una lista de sentencias.
Cada archivo fuente está compuesto por dos tipos de instrucciones básicas:

1. Sentencias: Son instrucciones que realizan algún proceso. No devuelven ningún valor (a excepción de sentencias de expresión, pero eso es otro rollo).

- ejemplos: print, select, delete, etc.

2. Expresiones: Son instrucciones que realizan una operacion entre operandos. Devuelven un valor al terminar la operación.

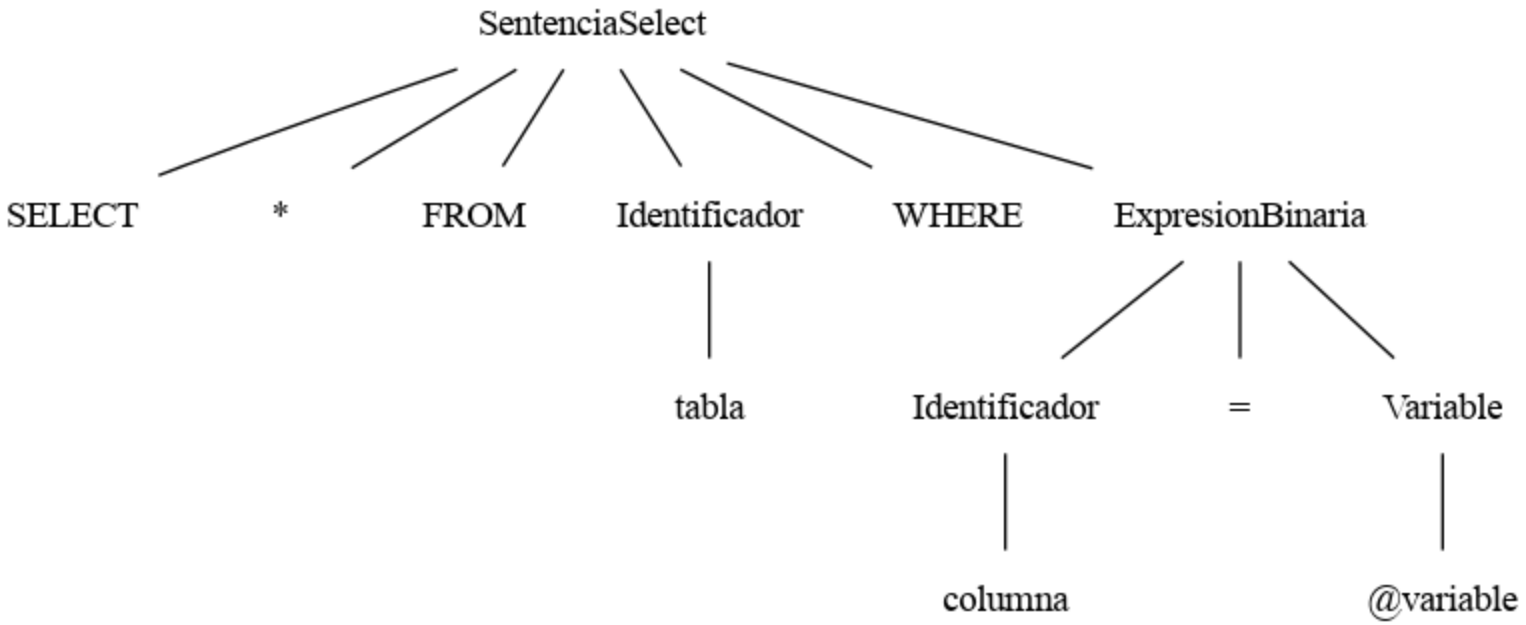
- ejemplos: 1 + 1, a OR b, @variable, etc.

2.2 ¿Qué es un AST?

Un AST es un estructura de datos en forma de arbol que representa la estructura de un archivo fuente.
Por ejemplo, la instrucción:

```
SELECT * FROM tabla WHERE columna = @variable;
```

Convertido en AST se vería algo así:



Como mencionaba en la primera diapositiva, el primer paso del interprete es poder reconocer el lenguaje por medio de una gramática hecha en Jison.
Dentro de nuestra gramática, tendremos producciones para reconocer cada parte del lenguaje fuente.
Tomando cómo ejemplo la instrucción anterior, veamos cómo generar el AST.

1. Dentro del archivo de gramática sintactica, tenemos producciones como estas:

```
Inst_Select:
    SELECT '*' FROM ID where
    {$$ = new Stmt.SelectFrom($4, $2, $5)}
;

where:
    WHERE lista_condiciones
    {$$ = $2}
    |
    /* epsilon */
;
```

El código encerrado en {} (llamado acción semántica) debajo de la producción se ejecutará cuando el analizador sintactico reconozca esa derivación.
Es decir, cuando esta cadena venga en el archivo fuente:

```
SELECT * FROM tabla WHERE columna = @variable;
```

el analizador la reconocerá, y luego ejecutará la acción semántica.

2. Como podemos darnos cuenta, lo que hace la acción semántica es retornar una nueva instancia de 'Stmt.SelectFrom'. Esta instancia será un nuevo nodo en nuestro AST.
   Es decir, cada vez que reconozcamos una instrucción en nuestro archivo fuente,
   en la acción semántica de esa producción generaremos un nuevo nodo del AST.
   De esa forma, iremos armando el AST paso a paso.

Las producciones de más alto nivel en nuestra gramática representarán una lista de instrucciones, visto en forma de gramática sería así:

```
querys:
    stmts EOF
    {
        $$ = $1
        return $$
    }
;

stmts:
    stmts stmt ';'
    {
        $1.push($2)
        $$ = $1
    }
    |
    stmt ';'
    {
        $$ = []
        $$.push($1)
    }
;
```

3. Por último, al terminar de reconocer nuestro archivo fuente tendremos una lista de instrucciones, representadas cada una por un AST. Solamente resta llamar el método interpretar() de cada instrucción, y eso interpretará nuestro lenguaje. Aquí está el controlador en el backend del proyecto que recibe una entrada de texto y la interpreta. Los [...] es código escondido.

```
export const interpret = (req, res) => {
    const {input} = req.body
    const stmts = parser.parse(input)
    const global = new Context('Global')
    [...]
    for(const stmt of stmts){
        stmt.interpret(global)
    }
    [...]
    res.status(200).json(output)
}
```

3. interpretar sentencia SELECT-FROM-WHERE

La cosa con esta sentencia es que hay que evaluar una expresión por cada fila en la tabla. Entendamos que implica esto.

3.1 Estructura de condición

la condición después del WHERE es de la forma:

```
condiciones:    condiciones ( AND | OR ) condicion
                |
                condicion

condicion:      IDENTIFICADOR ( = | != | < | > | <= | >= ) expr

expr:           expr ( + | - | * | / | % ) expr
                |
                ENTERO
                |
                DECIMAL
                |
                VARCHAR
                |
                FECHA
                |
                NULL
                |
                llamada_variable
```

basicamente, el lado izquierda de la condición es siempre un identificador,
el operador es relacional, y el lado derecho es una expresion aritmetica.
Pueden venir más condiciones usando AND u OR.

3.2 Interpretar la condición del WHERE

Hay que almacenar el AST de la condición, e interpretarlo
dentro del método que usamos para seleccionar filas de la tabla.
Antes de interpretar la condición, debemos crear un nuevo contexto que
tenga el valor de cada celda en la fila actual para las columnas seleccionadas.
Por ultimo, ese contexto se pasa al metodo interpretar() de la condición.

3.3 orden de ejecución de la sentencia Select

Para seguir un orden en la ejecución de la sentencia Select,
tomé inspiración de la forma en que funcionan los verdaderos
interpretes de SQL.
Cómo en el caso de este lenguaje solo ejecutamos tres pasos,
el orden sería:

1. FROM: Se selecciona la tabla correcta
2. WHERE: Si existe, se evalua la condición por cada fila en la tabla
3. SELECT: Se ejecutan las expresiones de retorno especificadas en select.
   Si la expresión es "*", simplemente se devuelve la fila entera sin cambios.
   Estos tres pasos se ejecutan para cada fila en la tabla.
4. AS: Finalmente, si las expresiones en SELECT tienen alias, se utilizan esos nombres
   en las tablas.

El resultado de aplicar ese orden es el siguiente método:

```
/* ./backend/src/database/Table.js */
select(selection, condition, context){
    const records = []
    for(let i = 0; i < this.cardinality; i++){
        const row = this.getRowAtIndex(i)
        const header = this.getHeaderContext(row, context)
        if(condition){
            const result = condition.interpret(header).valueOf()
            if(result){
                records.push(this.applyExpressions(selection, row, header))
            }
        }
        else{
            records.push(this.applyExpressions(selection, row, header))
        }
    }
    return {
        header: this.getAlises(selection),
        records
    }
}
```

Notas

- 'this.cardinality' es el número de filas en la tabla.
- la idea es que al momento de seleccionar las filas en una tabla, tengamos disponibles:

1. La lista de columnas que pide la consulta.
2. El AST de la condición del WHERE
3. El contexto en el que se hizo la consulta (Esto porque si en las expresiones viene una llamada a variable, debemos tener el contexto para trauducirla)

estas tres cosas son los argumentos (selection, condition, context) en la firma del método select()