MANUAL TÉCNICO PRÁCTICA 1, COMPILADORES I

2023

Erick Daniel Morales Xicara

Carné: 201930699

Manual Técnico

El software puede ser utilizado en cualquier sistema operativo ya que es un proyecto web, solamente se necesita acceder a internet y poder tener un dominio para que el proyecto sea funcional en cualquier lugar. Se utilizó NPM versión: 8.19.3, Angular versión 15.2.4 y Nodejs versión 16.19.0,a su vez se utilizó Webstorm como editor de texto, el cual es de IntelliJ.

El software consiste en el manejo de un lenguaje determinado, en cual es un pequeño intérprete donde se puede definir tablas de una base de datos y a su vez realizar consultas de las mismas, también tomamos en cuenta la declaración de variables las cuales pueden ir cambiando. tenemos palabras reservadas como DECLARE, IF,ELSEIF, END, SET entre otras mas

El software a su vez contiene varias clases, modelos y manejadores que permiten iniciar, ejecutar y realizar peticiones a la base de datos.

Lenguaje MiniSql

Paquete Models: este paquete contiene todos los modelos utilizados para poder manejar el lenguaje MiniSql

Las clases:

1. assignment
2. condition_select
3. declare
4. else_state
5. if_state
6. input
7. instruction
8. limit
9. off-set
10. operacion_binaria
11. print
12. select
13. settear
14. tabla_simbolos
15. where

Contienen un método run, el cual permite ir realizando las verificaciones y las operaciones determinadas por cada clase.

La clase Value: es la que nos da el valor de las variables por medio de su tabla run.

En el paquete Parser/ParserSql contiene el jison con las producciones y lenguaje léxico definido:

El siguiente método es el que nos ayuda a poder recopilar los errores léxicos.

```
%{
let errores_lexicos=[];
  function addEr(linea, columna,simbolo){
    var n={
      linea: linea,
      columna:columna,
      type: "Lexico",
      des: "simbolo no reconocido => "+simbolo
    }
    errores_lexicos.push(n);
  }
%}
```

También definimos las reglas léxicas

```
%lex
%%
\s+                     {/*ignore*/}
("--".*)              {/* ignore*/}
"INT"               return 'INT';
"DECIMAL"           return 'DECIMAL';
"TEXT"              return 'TEXT';
"BOOLEAN"           return 'BOOLEAN';
"TRUE"              return 'TRUE';
"FALSE"             return 'FALSE';
"DECLARE"           return 'DECLARE';
"AS"                return 'AS';
"SET"               return 'SET';
"AND"               return 'AND';
"OR"                return 'OR';
"INPUT"             return 'INPUT';
"PRINT"             return 'PRINT';
"IF"                return 'IF';
"ELSEIF"            return 'ELSEIF';
"ELSE"              return 'ELSE';
"END"               return 'END';
"THEN"              return 'THEN';
"NOT"               return 'NOT';
"SELECT"            return 'SELECT';
"FROM"              return 'FROM';
"WHERE"             return 'WHERE';
"LIMIT"             return 'LIMIT';
"OFFSET"            return 'OFFSET';
"("                 return 'LPARENT';
")"                 return 'RPARENT';
```

```
"+"                 return 'MAS';
"-"                 return 'MENOS';
"*"                 return 'POR';
"/"                 return 'DIVIDE';
";"                 return 'PUNTO_COMA';
","                 return `COMA`;
"<>"                return 'NO_IGUAL';
">="                return 'MAYOR_IGUAL_QUE';
"<="                return 'MENOR_IGUAL_QUE';
"<"                 return 'MENOR_QUE';
">"                 return 'MAYOR_QUE';
"="                 return 'IGUAL';
("@"[a-zA-Z][a-zA-Z0-9_]*) return 'VARIABLE';
([0-9]+("."[0-9]+)\b)   return 'NUM_DECIMAL';
([0-9]+\b)              return 'ENTERO';
(\"[^\"]*\")        { yytext = yytext.substr(1,yyleng-2); return 'CADENA'; }
(\'[^\']*\')        { yytext = yytext.substr(1,yyleng-2); return 'CADENA'; }
([a-zA-Z][a-zA-Z0-9_]*) %{
                        return 'LITERAL';
                    %}
<<EOF>>             %{
                        console.log('fin de archivo');
                        return "EOF";
                    %}
.                   %{
                        addEr(yylloc.first_line, yylloc.first_column, yytext)
                        return 'INVALID';
                    %}
/lex
```

La explicación creo que es entendible, definimos primero las palabras reservadas.
La expresión “--”.* => es la encargada de ignorar los comentarios de una sola linea y las expresiones :

```
(\"[^\"]*\")
-Esta expresión nos dice que al venir una comilla puede venir adentro cualquier cosa mientras no se otra comilla, y al final debe venir una comilla.

(\'[^\']*\')
-Esta expresión nos dice que al venir una comilla simple puede venir adentro cualquier cosa mientras no se otra comilla simple, y al final debe venir una comilla simple.
```

La gramática utilizada:

```
inic
 : declare_variables statements EOF
 ;

statements
 : statements  state_op
 | state_op

 ;



declare_variables : declare_variables  declare_prod
                  | declare_prod
;

declare_prod
          : DECLARE variablePro AS type IGUAL a PUNTO_COMA
          | DECLARE variablePro  AS type PUNTO_COMA
;
variablePro: variablePro COMA VARIABLE
           | VARIABLE
;
type: INT
 | DECIMAL
```

```
 | TEXT
 | BOOLEAN


state_op: print_stmt {$$=$1}
         | SET set_stmt PUNTO_COMA
         | if_stmt
         | select_stmt PUNTO_COMA


;
print_stmt
       : PRINT LPARENT expr RPARENT PUNTO_COMA
;

expr : expr COMA a
      | a
;

set_stmt
      : set_stmt COMA setPro
      | setPro
;

setPro: VARIABLE IGUAL a
;

if_stmt
 : IF a THEN statements else_statement END IF PUNTO_COMA}
   |IF a THEN  else_statement END IF PUNTO_COMA
;

else_statement
 : ELSEIF a THEN statements else_statement
 | ELSE statements
 |
 ;

select_stmt
```

```
         : SELECT name_select FROM LITERAL select
;
select
    : where_pro limit_pro off_set_pro
;
where_pro
       : WHERE a
       |
;

limit_pro
       : LIMIT a
       |
;

off_set_pro
         : OFFSET a
         |
;

name_select: POR
         | names_select
;

names_select: names_select COMA LITERAL
           |LITERAL
;

a
 : INPUT LPARENT CADENA RPARENT
 |a OR b
 | b
 ;

b
 : b AND c
 | c
 ;
```

```
c : NOT c
  | d
  ;

d: d NO_IGUAL e
 |d MENOR_QUE e
 |d MENOR_IGUAL_QUE e
 |d MAYOR_QUE e
 |d MAYOR_IGUAL_QUE e
 |d IGUAL e
 | e
;

e: e MAS f
 | e MENOS f }
 | f
;
f: f POR g
 | f DIVIDE g }
 | g
;

g: MENOS h
 | MAS h
 | h
;

h:  ENTERO
   | NUM_DECIMAL
   | CADENA
   | FALSE
   | TRUE
   | VARIABLE
   | LITERAL
   | LPARENT a RPARENT
;
```

A simple vista podemos ver que primero se deben declarar las variables seguidas del otro tipo de instrucciones reconocidas, en la declaración de variables:

- Se acepta DECLARE @variable as TEXT = “hola” es decir se declara una variable y se le puede asignar algo.
- Se acepta DECLARE @variable, @variable2 as TEXT es decir se declara dos o más variables y se le asigna un tipo de variable
- No se acepta DECLARE @variable, @variable2 as TEXT = “hola” es decir se declara más de una variable y se le quiere asignar algo.

A su vez la gramática para los statements es bastante práctico, ya que podemos recibir instrucciones como:

- PRINT
- SET
- SELECT
- IF

que son como las principales,

- El PRINT sus producciones serán el formato establecido pero que contiene la parte “expr” donde pueden venir variables, operaciones, cadenas a mostrar en consola.

- El SET es el que permite asignarle un valor o nuevo valor, este puede asignarle un valor a una variable o varias variables por eso la producción es SET set_stmt, donde el nuevo valor puede ser una operación, o un terminal.

- El IF, se controla por medio del if_stmt , el cual se modificó para tener el formato de un if común, donde el else_statement es el que indica que puede venir else if * , else o ninguno de estos.

- El SELECT, está controlado por el select_stmt, donde este cumplirá el formato SELECT … WHERE.. LIMIT… OFFSET donde el where, limit y offset pueden venir o no, esto nos permite controlar de mejor manera estas producciones

Definición de tablas:

Para esto, utilizamos el paquete objects, el cual tiene los siguientes objetos:

- Atributo
- Base De Datos
- Consulta
- DataB
- DBTable
- Propiedad
- Propiedad Stmt
- Stmt
- Type Propiedad
- TypeProStmt

Los cuales nos permite crear las tablas por medio del parser, cual los definimos de la siguiente manera:

Parte Lexica:

para errores lexicos:

```
%{
var myObject;
var ex=false;
var tamaño=0;
let errores_lexicos=[];
  function addError(linea, columna,simbolo){
    var n={
      linea: linea,
      columna:columna,
      type: "Lexico",
      des: "simbolo no reconocido => "+simbolo
    }
    errores_lexicos.push(n);
  }
```

Reglas léxicas

```
%lex
TRUE              "true"| "TRUE"
FALSE             "false"| "FALSE"
whitespace          [\n\t]+
ignore_line         \s+
space               \n
%%

(\#[^\r\n]*)            {/* ignore*/}
\s+                      {/*ignore*/}
"INT"                   return 'INT';
"DECIMAL"               return 'DECIMAL';
"STRING"                return 'STRING';
"BOOLEAN"               return 'BOOLEAN';
{TRUE}                  return 'TRUE';
{FALSE}                 return 'FALSE';
":"                     return 'DOS_PUNTOS';
";"                     return 'PUNTO_COMA';
","                     return `COMA`;
"&&"                    return 'AND';
"||"                    return 'OR';
"<="                    return 'MENOR_IGUAL_QUE';
">="                    return 'MAYOR_IGUAL_QUE';
"=="                    return 'DOBLE_IGUAL';
"!="                    return 'NO_IGUAL';
"<"                     return 'MENOR_QUE';
">"                     return 'MAYOR_QUE';
"="                     return 'IGUAL';
"!"                     return 'NOT';
"("                     return 'LPARENT';
```

```
"!"                     return 'NOT';
"("                     return 'LPARENT';
")"                     return 'RPARENT';
"+"                     return 'MAS';
"-"                     return 'MENOS';
"*"                     return 'POR';
"/"                     return 'DIVIDE';
(\"[^\"]*\")        { yytext = yytext.substr(1,yyleng-2); return 'CADENA'; }
(\'[^\']*\')        { yytext = yytext.substr(1,yyleng-2); return 'CADENA'; }
([0-9]+("."[0-9]+)\b)   return 'NUM_DECIMAL';
([0-9]+\b)                return 'ENTERO';
([a-zA-Z][a-zA-Z0-9_]*) %{
                          return 'LITERAL';
                        %}
<<EOF>>             %{
                        return 'EOF';
                    %}
.                   %{
                        addError(yylloc.first_line, yylloc.first_column, yytext)
                        return 'INVALID';
                    %}
/lex
```

Y la gramática:

```
inic : instruc EOF
     | EOF

;

instruc : instruc data_base
;

data_base
 : def_table
 | def_table_of PUNTO_COMA
   | error
 ;

def_table
       : name_table LPARENT def_values RPARENT PUNTO_COMA
;

name_table
       : LITERAL
;

def_values: def_values COMA properties
   | properties
;

properties: LITERAL type

type: INT
 | DECIMAL
 | STRING
 | BOOLEAN

def_table_of
          : def_table_of  def_values_of PUNTO_COMA
          | def_values_of
;
```

```
def_values_of: def_values_of COMA table_value
   | table_values
;

table_values: LITERAL IGUAL a
;

a
 : a OR b
 | b
 ;

b
 : b AND c
 | c
 ;

c : NOT c
  | d
  ;

d: d DOBLE_IGUAL e
 |d NO_IGUAL e
 |d MENOR_QUE e
 |d MENOR_IGUAL_QUE e
 |d MAYOR_QUE e
 |d MAYOR_IGUAL_QUE
 | e
;

e: e MAS f}
 | e MENOS f
 | f
;

f: f POR g
 | f DIVIDE g
 | g
;
```

```
g: MENOS h
 | MAS h
 | h
;

h:  ENTERO
   | NUM_DECIMAL
   | CADENA
   | FALSE
   | TRUE
   | LPARENT a RPARENT
;
```

Esta gramática tiene un pequeño error , es la parte donde se define la tabla seguida de sus registros, ya que el comienzo de un registro y de una tabla, en un LITERAL entonces el jison comprende que sigo analizando definición de tablas y no registro.

Respecto a lo demás es muy clara en cómo se derivan las producciones