Escuela Técnica Superior de Ingeniería
Universidad de Huelva

Grado en Ingeniería Informática

Generador de Analizadores Sintácticos Ascendentes

Procesadores de Lenguajes

Autora:
Alba Márquez-Rodríguez

Profesor:
Francisco José Moreno Velo

Huelva, 7 de Junio 2023

Índice

1. Introducción

El proyecto *Generador de Analizadores Sintácticos Ascendentes* tiene como objetivo desarrollar una aplicación para el análisis sintáctico ascendente de gramáticas, basada en la notación BNF (Forma Normal de *Backus-Naur*). La aplicación se encargará de generar los ficheros necesarios para el correcto funcionamiento del analizador sintáctico, así como implementar el algoritmo SLR para construir la tabla de desplazamiento/reducción.

El análisis sintáctico es una etapa crucial en el proceso de compilación de un lenguaje de programación. Se encarga de verificar la estructura gramatical de un programa, determinando si cumple con las reglas sintácticas establecidas por la gramática del lenguaje. Para este proyecto en específico, se utiliza una técnica de análisis ascendente, donde se construye un árbol de sintaxis abstracta que representa la estructura jerárquica del programa.

La notación BNF es utilizada para describir la sintaxis de los lenguajes formales. En ella, las reglas gramaticales se definen en términos de:

- Símbolos Terminales
- Símbolos No Terminales
- Flechas que indican la producción de una regla (*::=*)
- Símbolos especiales (—, *;*)

Esta notación permite definir la gramática de un lenguaje de manera precisa y concisa.

En este proyecto, se implementará un analizador léxico que filtrará los componentes léxicos no relevantes como los espacios en blanco y comentarios. Posteriormente, se desarrollará el analizador sintáctico/semántico utilizando el algoritmo SLR, que permitirá construir una tabla de desplazamiento/reducción para determinar la acción a realizar en cada paso del análisis.

Además, se generará el código correspondiente a las clases que conforman el Árbol de Sintaxis Abstracta, una estructura de datos que representa la estructura sintáctica del programa. Asimismo, se implementará la generación de los ficheros *"TokenConstants.java"*, *"SymbolConstants.java"* y *"Parser.java"*, los cuales serán fundamentales para el funcionamiento del analizador sintáctico.

En esta memoria se presentan los diferentes componentes del proyecto, desde el autómata finito determinista utilizado en el análisis léxico, hasta el algoritmo SLR y la generación de ficheros y clases. Además, se proporcionarán ejemplos de pruebas del funcionamiento de la aplicación. A través de este proyecto, se espera lograr una mejor comprensión y aplicación de los conceptos teóricos relacionados con el análisis sintáctico y la generación de compiladores.

Se puede encontrar más información sobre el proyecto que se describe en esta memoria en la página web de la asignatura [1].

1.1. Fichero de prueba `Main.txt`

El fichero introducido para las pruebas contiene lo que se podría encontrar en un fichero de entrada al proyecto junto a algunas otras posibilidades para realizar la prueba:

```
/* Definici n de una expresi n aritm tica como una suma o resta de t rminos */
Expr ::= Term
    | Expr <PLUS> Term
    | Expr <MINUS> Term
  ;

/* Los t rminos son productos o divisiones de factores */
Term ::= Factor
    | Term <PROD> Factor
    | Term <DIV> Factor
  ;

/* Los factores son constantes, expresiones entre par ntesis o llamadas a funciones */
Factor ::= <NUM>
    | <LPAREN> Expr <RPAREN>
  | <IDENTIFIER> <LPAREN> Args <RPAREN>
  ;

/* Las reglas lambda se crean con secuencias vac as, como la segunda l nea de esta
    definici n */
Args ::= ArgumentList
  |
  ;

/* Los argumentos son una lista de expresiones separadas por coma */
ArgumentList ::= Expr
  | ArgumentList <COMMA> Expr
  ;
```

Listing 1: Ejemplo de gramática en notación BNF

2. Autómata Finito Determinista en el que se basa el analizador léxico.

El analizador léxico descompone el flujo de caracteres de entrada en una secuencia de componentes léxicos o tokens, que representan unidades significativas en el lenguaje. Para lograr esto, se utiliza un **Autómata Finito Determinista (AFD)** que se basa en la especificación léxica de la gramática en notación BNF.

2.1. Especificación Léxica

La especificación léxica de las gramáticas en notación BNF se compone de diferentes categorías, cada una de ellas asociada a una expresión regular que define su estructura y patrón de reconocimiento. Estas categorías son:

Especificación	Expresión regular
blanco	(" " | "\r" | "\n" | "\t")
comentario	"/*" (("*")* ~["*", "/"] | "/")* ("*")+ "/"
NOTERMINAL	["_","a"-"z","A"-"z"] (["_","a"-"z","A"-"Z","0"-"9"])*
TERMIINAL	"<" ["_","a"-"z","A"-"z"] (["_","a"-"z","A"-"Z","0"-"9"])* ">"
EQ	"::="
BAR	"|"
SEMICOLON	";"

Figura 1: Especificación léxica

- **blanco**: se refiere a espacios en blanco, tabuladores, saltos de línea y otros caracteres de espacio, los cuales deben ser filtrados y no forman parte de los componentes léxicos relevantes.
- **comentario**: representa los comentarios multilínea que siguen la convención de C, iniciando con /ƨ finalizando con /". Estos comentarios deben ser filtrados y no se consideran como componentes léxicos.
- **NOTERMINAL**: está asociada a identificadores y se utiliza para referenciar a los símbolos no terminales de la gramática. Estos identificadores pueden comenzar con un carácter de subrayado ("_") o una letra del alfabeto (mayúscula o minúscula), seguidos opcionalmente de letras, dígitos o subrayados.
- **TERMINAL**: se refiere a los símbolos terminales de la gramática, los cuales se escriben entre los signos menor («") y mayor (»"). Estos símbolos terminales también deben seguir las mismas reglas que los identificadores no terminales.
- **EQ**: representa la flecha que separa las partes izquierda y derecha de una regla en la gramática. Esta flecha se denota con el símbolo "::=".
- **BAR**: es el separador utilizado entre diferentes reglas de un mismo símbolo no terminal. Se representa con el carácter "—".
- **SEMICOLON**: corresponde al punto y coma que se utiliza como finalizador de las reglas de un símbolo no terminal.

2.2. Autómata Finito Determinista

El AFD utilizado en el analizador léxico se construye considerando estas categorías y sus correspondientes expresiones regulares. Los estados del AFD representan las etapas del proceso de reconocimiento léxico, y las transiciones se definen según los caracteres de entrada y las reglas léxicas establecidas por la gramática.

En este caso, el AFD se construye considerando la especificación léxica de la gramática en notación BNF. Los caracteres de entrada son procesados uno a uno y el AFD se desplaza de un estado a otro de acuerdo con la función de transición, determinando así el tipo de token que se está analizando.

La correcta definición y construcción del AFD es fundamental para asegurar un análisis léxico preciso y eficiente. Al aplicar el AFD al flujo de caracteres de entrada, el analizador léxico podrá identificar y clasificar cada componente léxico, proporcionando la base necesaria para el posterior análisis sintáctico y semántico del programa.

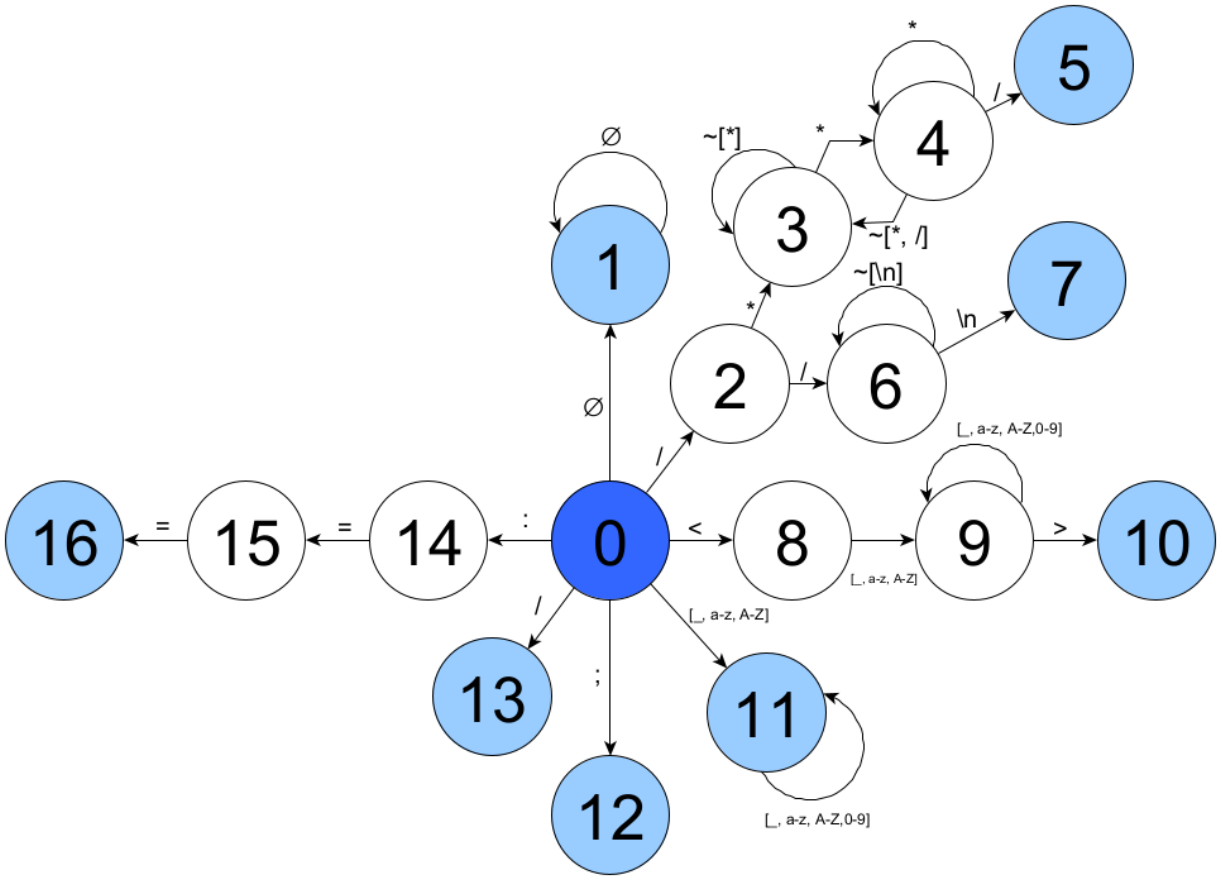

Figura 2: Autómata Finito Determinista

3. Código de la clase que desarrolla el analizador léxico

A partir del **Automata Finito Determinista** anterior, se ha elaborado el código de la clase Lexer que se encarga del analizador léxico.

La clase `GrammarLexer` extiende la clase Lexer, que es la encargada de implementar el análisis léxico. La función `transition` define las transiciones del autómata del analizador léxico, donde se establecen las reglas para reconocer los diferentes componentes léxicos. La función `isFinal` verifica si un estado es final, es decir, si corresponde a un componente léxico válido. La función `getToken` genera el componente léxico correspondiente al estado final y al lexema encontrado. En esta última función además se puede indicar que tokens se quieren considerar para el análisis léxico y qué tokens se quieren ignorar. En esta implementación se ha decidido ignorar los Tokens de comentarios y blancos desde el analizador léxico.

```
/**
 * Clase que desarrolla el analizador lexico
 */
public class GrammarLexer extends Lexer {

  /**
   * Transiciones del automata del analizador lexico
   *
   * @param state Estado inicial
   * @param symbol Simbolo del alfabeto
   * @return Estado final
   */
  protected int transition(int state, char symbol) {
      switch (state) {
          case 0:
              if (symbol == ' ' || symbol == '\r' || symbol == '\n' || symbol == '\t')
    return 1;
              else if (symbol == '/') return 2;
              else if (symbol == '_' || (symbol >= 'a' && symbol <= 'z') || (symbol >= 'A'
    && symbol <= 'Z')) return 11;
              else if (symbol == '<') return 8;
              else if (symbol == ':') return 14;
              else if (symbol == '|') return 13;
              else if (symbol == ';') return 12;
              else return -1;
          case 1:
              if (symbol == ' ' || symbol == '\r' || symbol == '\n' || symbol == '\t')
    return 1;
              else return -1;
          case 2:
              if (symbol == '*') return 3;
              else if (symbol == '/') return 6;
              else return -1;
          case 3:
              if (symbol == '*') return 4;
              else return 3;
          case 4:
              if (symbol == '*') return 4;
              else if (symbol == '/') return 5;
              else return 3;
          case 6:
              if (symbol == '\n') return 7;
              else return 6;
          case 8:
            if (symbol == '_' || (symbol >= 'a' && symbol <= 'z') || (symbol >= 'A' &&
    symbol <= 'Z')) return 9;
              else return -1;
          case 9:
            if (symbol == '_' || (symbol >= 'a' && symbol <= 'z') || (symbol >= 'A' &&
    symbol <= 'Z') || (symbol >= '0' && symbol <= '9')) return 9;
            else if (symbol == '>') return 10;
            else return -1;
          case 11:
```

```
            if (symbol == '_' || (symbol >= 'a' && symbol <= 'z') || (symbol >= 'A' &&
    symbol <= 'Z') || (symbol >= '0' && symbol <= '9')) return 11;
            else return -1;
          case 14:
            if (symbol == ':') return 15;
            else return -1;
          case 15:
            if (symbol == '=') return 16;
            else return -1;
          default: return -1;
      }
  }

  /**
   * Verifica si un estado es final
   *
   * @param state Estado
   * @return true, si el estado es final
   */
  protected boolean isFinal(int state) {
      switch (state) {
          case 1:
          case 5:
          case 7:
          case 10:
          case 11:
          case 12:
          case 13:
          case 16:
              return true;
          default:
              return false;
      }
  }

  /**
   * Genera el componente lexico correspondiente al estado final y
   * al lexema encontrado. Devuelve null si la accion asociada al
   * estado final es omitir (SKIP).
   *
   * @param state Estado final alcanzado
   * @param lexeme Lexema reconocido
   * @param row Fila de comienzo del lexema
   * @param column Columna de comienzo del lexema
   * @return Componente lexico correspondiente al estado final y al lexema
   */
  protected Token getToken(int state, String lexeme, int row, int column) {
      switch (state) {
        // Comentados llos que no se quieren mostrar n contemplar como token
        //case 1: return new Token(TokenKind.BLANCO, lexeme, row, column);
          //case 5: return new Token(TokenKind.COMENTARIO, lexeme, row, column);
          //case 7: return new Token(TokenKind.COMENTARIO, lexeme, row, column);
          case 10: return new Token(TokenKind.TERMINAL, lexeme, row, column);
          case 11: return new Token(TokenKind.NOTERMINAL, lexeme, row, column);
          case 12: return new Token(TokenKind.SEMICOLON, lexeme, row, column);
          case 13: return new Token(TokenKind.BAR, lexeme, row, column);
          case 16: return new Token(TokenKind.EQ, lexeme, row, column);
          default: return null;
      }
  }

  public GrammarLexer(File file) throws IOException
  {
    super(file);
  }
}
```

Listing 2: Código clase GrammarLexer

3.1. Prueba Analizador Léxico

Para probar si su funcionamiento es correcto se ha elaborado un método main. Este método main permite probar la funcionalidad del analizador léxico *GrammarLexer* al analizar el archivo de entrada. Se pueden observar los tokens reconocidos en la salida y verificar si se corresponden con lo esperado según la especificación léxica y las reglas definidas en *GrammarLexer*.

```
1
2  public static void main(String[] args) {
3      try {
4          File inputFile = new File("Main.txt");
5          GrammarLexer lexer = new GrammarLexer(inputFile);
6
7          Token token = lexer.getNextToken();
8          while (token.getKind() != TokenKind.EOF) {
9              if (token.getKind() != TokenKind.COMENTARIO) {
10                 System.out.println(token);
11             }
12             token = lexer.getNextToken();
13         }
14
15         lexer.close();
16     } catch (IOException e) {
17         e.printStackTrace();
18     }
19 }
```

Listing 3: Código Test del GrammarLexer

La salida recibida para el fichero de entrada descrito en 1 es la siguiente:

```
1  Expr
2  ::=
3  Term
4  |
5  Expr
6  <PLUS>
7  Term
8  |
9  Expr
10 <MINUS>
11 Term
12 ;
13
14 ...
15
16 ArgumentList
17 ::=
18 Expr
19 |
20 ArgumentList
21 <COMMA>
22 Expr
23 ;
```

Listing 4: Extracto de la salida recibida del Test

La salida obtenida se corresponde con la salida esperada, pues se muestran todos los tokens que se han encontrado que no pertenecen al tipo ni comentario ni blanco ya que estos deben ser ignorados.

4. Gramática BNF en la que se basa el analizador sintáctico

Para realizar el análisis sintáctico, se emplea una gramática **BNF (Backus-Naur Form)** que define la estructura sintáctica del lenguaje. La gramática BNF proporciona reglas claras y precisas que determinan cómo se pueden combinar los símbolos no terminales y terminales para formar las construcciones válidas en el lenguaje.

```
Gramatica ::= ( Definicion )*
Definicion ::= NOTERMINAL EQ ListaReglas SEMICOLON
ListaReglas ::= Regla ( BAR Regla )*
Regla ::= ( NOTERMINAL | TERMINAL )*
```

Listing 5: Gramática BNF en la que se basa el analizador sintáctico

En esta especificación, **Gramatica** representa la gramática en su conjunto, que se compone de una o más **Definicion**. Cada **Definicion** se compone de un **NOTERMINAL** (símbolo no terminal) seguido de **EQ ('::=')**, seguido de una **ListaReglas** y finalmente un **SEMICOLON (';')** para indicar el final de la definición.

La **ListaReglas** se compone de una o más **Regla**, separadas por el símbolo de barra **BAR**. Una **Regla** puede contener uno o más **NOTERMINAL** o **TERMINAL**, que representan los símbolos no terminales y terminales de la gramática, respectivamente.

En resumen, esta especificación sintáctica define la estructura de una gramática en términos de sus definiciones, reglas y símbolos terminales y no terminales. El analizador sintáctico utilizará esta especificación para verificar la validez sintáctica del código fuente y construir un árbol de análisis sintáctico correspondiente.

4.1. Gramática LR(1)

Para obtener los conjuntos de predicción de la gramática es necesario definir primero una gramática LR(1) equivalente a la especificación sintáctica indicada y calcular los conjuntos de predicción para cada no terminal de la gramática.

A partir de la gramática extendida, se debe simplificar a gramática LL(1) y BNF. Para convertir la gramática en LL(1) hay que eliminar la factorización y la recursividad izquierda. Y para transformarla en BNF hay que eliminar las disyunciones, cláusulas, claúsulas positivas y opcionalidad.

Primero se ha realizado la representación LL(1), La gramática quedaría de la siguiente manera:

```
Gramatica -> Definiciones
Definiciones -> Definicion Definiciones
Definiciones -> \lambda
Definicion -> NOTERMINAL EQ ListaReglas SEMICOLON
ListaReglas -> Regla ListaReglasPrime
ListaReglasPrime -> BAR Regla ListaReglasPrime
ListaReglasPrime -> \lambda
Regla -> Elemento Regla'
Regla' -> BAR Elemento Regla' | \lambda
Elemento -> NOTERMINAL | TERMINAL
```

Listing 6: Gramática LL(1)

En esta gramática, se han introducido los símbolos no terminales adicionales *Definiciones*, *ListaReglasPrime*, *ReglaPrime* y *Elemento* para manejar la recursividad a izquierda en la gramática original. Además, para representar la producción nula (es decir, una regla vacía) se ha utilizado λ.

Eliminando claúsulas y disyunciones quedaría la siguiente representación:

```
Gramatica -> Definicion Gramatica
Gramatica -> \lambda
Definicion -> NOTERMINAL EQ ListaReglas SEMICOLON
ListaReglas -> Regla ListaReglasPrime
```

```
ListaReglasPrime -> BAR Regla ListaReglasPrime
ListaReglasPrime -> \lambda
Regla -> Elemento Regla
Regla -> \lambda
Elemento -> NOTERMINAL
Elemento -> TERMINAL
```

Listing 7: Gramática BNF

5. Conjuntos de predicción de dicha gramática

Para calcular los conjuntos de predicción de la gramática, se deben analizar las producciones de cada símbolo no terminal y determinar los conjuntos *FIRST* y *FOLLOW* correspondientes. A continuación, se muestran los conjuntos *FIRST* y *FOLLOW* para cada símbolo no terminal de la gramática:

```
Conjuntos FIRST:
    FIRST(Gramatica) = {NOTERMINAL,   }
    FIRST(Definicion) = {NOTERMINAL}
    FIRST(ListaReglas) = {NOTERMINAL,   }
    FIRST(ListaReglasPrime) = {BAR,   }
    FIRST(Regla) = {NOTERMINAL,   }
    FIRST(Elemento) = {NOTERMINAL, TERMINAL}
Conjuntos FOLLOW:
    FOLLOW(Gramatica) = {EOF}
    FOLLOW(Definicion) = {EOF, NOTERMINAL}
    FOLLOW(ListaReglas) = {SEMICOLON}
    FOLLOW(ListaReglasPrime) = {SEMICOLON}
    FOLLOW(Regla) = {BAR, SEMICOLON}
    FOLLOW(Elemento) = {BAR, SEMICOLON}
```

Estos conjuntos de predicción indican los símbolos terminales que se pueden derivar a partir de cada símbolo no terminal en cada posición. Los conjuntos de *FIRSTS* indican los primeros símbolos que pueden aparecer en una producción, mientras que los conjuntos de *FOLLOWS* indican los símbolos que pueden aparecer inmediatamente después de cada producción.

Ahora podemos usar estos conjuntos para determinar los conjuntos de predicción de cada producción. Los conjuntos de predicción se definen como la unión de los conjuntos FIRST de las producciones y, en caso de que las producciones deriven en , se incluye el conjunto *FOLLOW*.

```
Gramatica ::= Definicion Gramatica
    FIRST(Definicion) = {NOTERMINAL}
    PRED(Gramatica) = {NOTERMINAL}
Gramatica ::= \lambda
    FOLLOW(Gramatica) = {EOF}
    PRED(Gramatica) = {EOF}
Definicion ::= NOTERMINAL EQ ListaReglas SEMICOLON
    FIRST(NOTERMINAL) = {NOTERMINAL}
    PRED(Definicion) = {NOTERMINAL}
ListaReglas ::= Regla ListaReglasPrime
    FIRST(Regla) = {NOTERMINAL, \lambda}
    PRED(ListaReglas) = {NOTERMINAL}
ListaReglasPrime ::= BAR Regla ListaReglasPrime
    FIRST(BAR) = {BAR}
    PRED(ListaReglasPrime) = {BAR}
ListaReglasPrime ::= \lambda
    FOLLOW(ListaReglasPrime) = {SEMICOLON}
    PRED(ListaReglasPrime) = {SEMICOLON}
Regla ::= Elemento Regla
    FIRST(Elemento) = {NOTERMINAL, TERMINAL}
    PRED(Regla) = {NOTERMINAL, TERMINAL}
Regla ::= \lambda
    FOLLOW(Regla) = {BAR, SEMICOLON}
    PRED(Regla) = {BAR, SEMICOLON}
Elemento ::= NOTERMINAL
    FIRST(NOTERMINAL) = {NOTERMINAL}
    PRED(Elemento) = {NOTERMINAL}
Elemento ::= TERMINAL
    FIRST(TERMINAL) = {TERMINAL}
    PRED(Elemento) = {TERMINAL}
```

6. Código de la clase que desarrolla el analizador sintáctico/-semántico

A continuación se muestra el código de la clase que implementa el analizador sintáctico/semántico:

```
public class GrammarParser {
    private Lexer lexer;
    private Token currentToken;

    public GrammarParser(String filePath) {
        try {
            File file = new File(filePath);
            lexer = new GrammarLexer(file);
            currentToken = lexer.getNextToken();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void match(int expectedKind) throws ParseException {
        // si es comentario
        if (currentToken.getKind() == TokenKind.COMENTARIO) {
            // no hacer nada
            currentToken = lexer.getNextToken();
        }
        if (currentToken.getKind() == expectedKind) {
            currentToken = lexer.getNextToken();
        }
        else {
            throw new ParseException("Expected " + expectedKind + " but found " +
    currentToken.getKind());
        }
    }

    public void parse() throws ParseException {
        while (currentToken.getKind() != TokenKind.EOF){
            definicion();
        }
    }

    private void definicion() throws ParseException {
        match(TokenKind.NOTERMINAL);
        match(TokenKind.EQ);
        rulesList();
        match(TokenKind.SEMICOLON);
    }

    private void rulesList() throws ParseException {
        rule();
        while (currentToken.getKind() == TokenKind.BAR){
            match(TokenKind.BAR);
            rule();
        }
    }

    private void rule() throws ParseException {
        while (currentToken.getKind() == TokenKind.NOTERMINAL || currentToken.getKind() ==
    TokenKind.TERMINAL){
            if (currentToken.getKind() == TokenKind.NOTERMINAL) {
                match(TokenKind.NOTERMINAL);
            } else {
                match(TokenKind.TERMINAL);
            }
        }
    }
}
```

Listing 8: Código de la clase GrammarParser

En esta clase, se utiliza un objeto `Lexer (GrammarLexer)` para obtener los tokens del archivo de entrada. El analizador sintáctico/semántico implementa métodos que siguen la gramática definida para analizar la estructura del archivo.

El método `parse` inicia el análisis y procesa todas las definiciones presentes en el archivo. Llama al método `definicion` para analizar cada definición.

El método `definicion` verifica la sintaxis de una definición, asegurándose de que comience con un **NOTERMINAL** seguido de un **EQ** y luego una **lista de reglas** (`rulesList`), finalizando con un **SEMICOLON**.

El método `rulesList` analiza una lista de reglas, llamando al método `rule` para cada regla encontrada. Si se encuentra un símbolo **BAR**, indica que hay más de una regla definida.

El método `rule` analiza una regla, asegurándose de que esté compuesta por uno o más elementos, que pueden ser **NOTERMINAL** o **TERMINAL**.

En caso de que se produzca alguna discrepancia sintáctica durante el análisis, se lanza una excepción `ParseException` indicando el tipo de token esperado y el token encontrado.

Para probar si su funcionamiento es correcto se ha elaborado un método main. Este método main permite probar la funcionalidad del analizador sintáctico `GrammarParser` al analizar el archivo de entrada. Si el análisis es correcto, se mostrará por pantalla *Análisis sintáctico completado exitosamente*, en caso contrario se mostrará *Error de análisis sintáctico: - e.getMessage()* donde `e.getMessage()` es el error descrito antes.

```
public static void main(String[] args) {
        String filePath = "Main.txt";

        GrammarParser parser = new GrammarParser(filePath);

        try {
            parser.parse();
            System.out.println("An lisis sint ctico completado exitosamente.");
        } catch (ParseException e) {
            System.err.println("Error de an lisis sint ctico: " + e.getMessage());
        }
}
```

Listing 9: Código Test del GrammarParser

7. Código de las clases que desarrollan el Árbol de Sintaxis Abstracta.

A continuación se presentan las clases relevantes que participan en la construcción del Árbol de Sintaxis Abstracta (AST) en el proyecto. Estas clases son fundamentales para la representación y el análisis de la estructura sintáctica del lenguaje. A través del análisis léxico y sintáctico, se construye el AST, que captura la jerarquía y las relaciones entre los elementos del código fuente.

Clase Rule

La clase `Rule` representa una regla gramatical en el AST. Contiene un identificador y una lista de expresiones que conforman la producción de la regla. Las reglas gramaticales son elementos fundamentales en la definición de la estructura y las relaciones sintácticas del lenguaje en un AST.

```
public class Rule {

  String identifier;
  List<Expression> production;
  ArrayList<String> firsts;
  ArrayList<String> follows;

  public Rule(String identifier) {
    this.identifier = identifier;
    production = new ArrayList<>();
    firsts = new ArrayList<>();
    follows = new ArrayList<>();
  }

  public Rule(String identifier, List<Expression> produccionRegla) {
    this.identifier = identifier;
    this.production = produccionRegla;
  }


  public boolean equalRules(Rule ruleAnalizar) {
      return this.identifier.equals(ruleAnalizar.identifier) &&
              this.production.equals(ruleAnalizar.production);
  }

  public void addExpressions(String expression, boolean terminal) {
    Expression aux = new Expression(expression, terminal);
    production.add(aux);
  }

  public void addFirsts(String token) {
    if(!firsts.contains(token))
      firsts.add(token);
  }

  public void addFollows(String token) {
    if(!follows.contains(token))
      follows.add(token);
  }

  public String toString() {
      StringBuilder builder = new StringBuilder();
      builder.append(identifier).append(" ->").append(production.toString());
      return builder.toString();
  }
}
```

Listing 10: Clase Rule

Clase Expression

La clase `Expression` representa una expresión en el AST. Cada expresión puede ser un símbolo terminal o no terminal. Diferenciar entre símbolos terminales y no terminales es esencial para la construcción y el análisis sintáctico del lenguaje.

```
public class Expression {

  String expression;
  boolean terminal;

  public Expression(Expression expression) {
    this.expression = expression.expression;
    this.terminal = expression.terminal;
  }

  public Expression(String expresion, boolean terminal) {
    this.expression = expresion;
    this.terminal = terminal;
  }

  public String getExpression() {
    return expression;
  }

  public void setExpression(String expression) {
    this.expression = expression;
  }

  public boolean isTerminal() {
    return terminal;
  }

  public String toString(){
    return expression + " ";
  }
}
```

Listing 11: Clase Expression

Clase Grammar

La clase `Grammar` representa la gramática en el AST. Contiene una lista de reglas gramaticales que definen la estructura sintáctica del lenguaje. Esta clase es crucial para la construcción del AST, ya que la gramática es la base fundamental para el análisis y la generación de la estructura sintáctica del código fuente.

Se ha simplificado el código, al ser una clase muy extensa, para ver todo su contenido se puede acceder al código del proyecto adjuntado junto a la memoria.

```
public class Grammar {

  ArrayList<Rule> rules;

  public Grammar() {
    rules = new ArrayList<>();
  }

  public Grammar(Grammar grammar) {
    rules = new ArrayList<>();
    rules.addAll(grammar.rules);
  }

  public void addRule(Rule rule) {
    rules.add(rule);
  }

  private void calcularFirsts() {
```

```
    for (Rule reg : rules) {
      Expression analizar = reg.production.get(0);
      calculaFirstsRule(reg, analizar);
    }
    unificarFirsts();
  }

  private void calcularFollows() {
    ArrayList<String> identificadores = getIdentifiers();

    for (Rule reg : rules) {
      if (reg.identifier.equals(identificadores.get(0))) {
        reg.addFollows("<EOF>");
      }
    }

    for (String id : identificadores) {
      ArrayList<String> aux = calcularNextIdentifier(id);

      for (Rule reg : rules) {
        if (reg.identifier.equals(id)) {
          reg.follows = unionConjuntos(reg.follows, aux);
        }
      }
    }
  }
}
```

Listing 12: Clase Grammar

Clase Automaton

La clase `Automaton` representa el autómata en el AST. Un autómata es una representación formal del análisis sintáctico que ayuda a determinar si una secuencia de símbolos cumple con las reglas gramaticales del lenguaje. El autómata desempeña un papel crucial en la validación y el análisis sintáctico del código fuente.

Se ha simplificado el código, al ser una clase muy extensa, para ver todo su contenido se puede acceder al código del proyecto adjuntado junto a la memoria.

```
public class Automaton {
    private List<State> states;
    private Grammar grammar;

    public Automaton(Grammar grammar) {
        states = new ArrayList<>();
        this.grammar = grammar;
    }

    public Grammar getGrammar() {
    return grammar;
  }

    public void createState0() throws FileNotFoundException {
        Rule firstRule = grammar.rules.get(0);

        List<Expression> initProduction = new ArrayList<>();
        initProduction.add(new Expression(firstRule.identifier, false));
        GrammarElement firstElement = new GrammarElement("Inicio", initProduction, false);
        List<GrammarElement> stateElements = new ArrayList<>();
        stateElements.add(firstElement);

        createStateFromElement(stateElements);

        System.out.println("Final Alcanzado\n");

        try (PrintStream stream = new PrintStream(new FileOutputStream(new File("src/
    generated/Automata.txt")))) {
            stream.println(states.toString());
```

```
        }
    }

    public int createStateFromElement(List<GrammarElement> startingElements) {
        State newStates = new State();

        int newAddedElements = 0;
        for (GrammarElement startingElement : startingElements) {
            newStates.addElement(startingElement);
            int i = 0;
            do {
                List<GrammarElement> toAddElements = newElementsFromMarkedElement(newStates.
    getElements().get(i + newAddedElements));
                for (GrammarElement toAddElement : toAddElements) {
                    newStates.addElement(toAddElement);
                }
                i++;
            } while (newStates.getElements().size() - newAddedElements > i);
            newAddedElements += i;
        }

        int statePos = existsStates(newStates);
        if (statePos != -1) {
            return statePos;
        }

        states.add(newStates);
        int addedStatePos = states.size() - 1;

        List<Transition> stateTransitions = possibleTransitionsInState(states.get(
    addedStatePos));
        for (Transition transition : stateTransitions) {
            states.get(addedStatePos).addTransition(transition);
        }

        return addedStatePos;
    }


    public List<GrammarElement> newElementsFromMarkedElement(GrammarElement elementToAnalyze
    ) {
        List<GrammarElement> returnElements = new ArrayList<>();
        int markerIdx = elementToAnalyze.markerIdx();

        if (!elementToAnalyze.markerAtEnd()) {
            Expression nextMarkerProduction = elementToAnalyze.getProduction().get(markerIdx
     + 1);

            if (nextMarkerProduction.terminal) {
                return returnElements;
            } else {
                String identifier = nextMarkerProduction.expression;

                for (Rule reg : grammar.rules) {
                    if (reg.identifier.equals(identifier)) {
                        List<Expression> newElementProduction = new ArrayList<>(reg.
    production);
                        GrammarElement newElement = new GrammarElement(reg.identifier,
    newElementProduction, false);
                        returnElements.add(newElement);
                    }
                }
            }
        }

        return returnElements;
    }
}
```

Listing 13: Clase Automaton

Clase State

La clase `State` representa un estado en el AST. Cada estado del autómata tiene transiciones hacia otros estados y puede contener elementos gramaticales específicos. Los estados del autómata ayudan a definir la estructura y las relaciones sintácticas del lenguaje.

```
public class State {
    private ArrayList<Transition> transitions;
    //Reglas punteadas * para ver por donde va la lectura...
    private ArrayList<GrammarElement> grammarElements;

    public State() {
        transitions = new ArrayList<>();
        grammarElements = new ArrayList<>();
    }

    public ArrayList<Transition> getTransiciones() {
        return transitions;
    }

    public ArrayList<GrammarElement> getElements() {
        return grammarElements;
    }

    public void addElement(GrammarElement grammarElement) {//A adimos elementos excluyendo
    estados repetidos
        boolean existeElemento = false;

        for(GrammarElement elementoExistente : grammarElements) {
            if(grammarElement.isEqualElement(elementoExistente))
                existeElemento = true;
        }
        if(!existeElemento)
            grammarElements.add(grammarElement);
    }

    public void addTransition(Transition transition) {
        boolean transitionExists = false;

        // A adir transiciones excluyendo repetidas
        for(Transition existingTransition : transitions) {
            if(transition.areTransitionsEqual(existingTransition))
                transitionExists = true;
        }

        if(!transitionExists)
            transitions.add(transition);
    }
}
```

Listing 14: Clase State

Clase Transition

La clase `Transition` representa una transición en el AST. Las transiciones conectan estados en el autómata y se activan al leer un símbolo específico. Las transiciones permiten el recorrido y la validación de la estructura sintáctica del código fuente.

```
public class Transition {

    private Expression source;
    private int destination;
    private boolean isReduce;

    public Transition() {
        this.isReduce = false;
    }

    public Transition(Expression expr, int destination) {
        this.source = expr;
        this.destination = destination;
        this.isReduce = false;
    }

    public void setSource(Expression source) {
        this.source = source;
    }

    public Expression getSource() {
        return this.source;
    }

    public void setDestination(int destination) {
        this.destination = destination;
    }

    public void addReduce() {
        this.isReduce = true;
    }

    public boolean isReduce() {
        return this.isReduce;
    }

    public boolean areTransitionsEqual(Transition transition) {
        return this.destination == transition.getDestination();
    }

    public int getDestination() {
        return this.destination;
    }
}
```

Listing 15: Clase Transition

Clase GrammarParser

La clase `GrammarParser` es la clase principal que genera el AST a partir del código fuente. Utiliza un `Lexer` para analizar el código fuente y construir el AST basado en la gramática y las reglas sintácticas definidas. El GrammarParser orquesta el proceso de análisis sintáctico y genera la estructura sintáctica del código fuente en forma de árbol.

Su código se puede encontrar en 8

8. Código de la clase que desarrolle el algoritmo SLR y explicación de su funcionamiento.

La clase `Generator` representa el generador del analizador sintáctico SLR. Contiene una serie de métodos y variables para leer la gramática, generar las constantes de tokens y símbolos, crear el autómata y generar el analizador.

El algoritmo **SLR (Simple LR)** utiliza las tablas de acción y transición generadas a partir del autómata LR(0) para decidir entre las acciones de reducción y desplazamiento durante el análisis sintáctico. Estas acciones determinan cómo se procesa la entrada y se reconocen las estructuras sintácticas válidas.

Cuando el analizador sintáctico se encuentra en un estado y recibe un símbolo de entrada, tiene dos posibles acciones a considerar:

- **Desplazamiento (Shift)**: El analizador sintáctico realiza un desplazamiento cuando mueve el símbolo de entrada a la pila y avanza al siguiente símbolo de entrada. Esto implica que el analizador sintáctico espera encontrar más símbolos que formen una estructura sintáctica válida en el futuro.

  La acción de desplazamiento ocurre cuando en la tabla de acción (actionTable) se encuentra una entrada que indica un desplazamiento. Por lo general, esto se representa mediante el número del siguiente estado al que se debe mover el analizador sintáctico.

  El desplazamiento permite procesar la entrada incrementalmente y mantener un seguimiento de las estructuras sintácticas parciales. Al realizar desplazamientos, el analizador sintáctico construye una pila de símbolos que representa la secuencia de símbolos vistos hasta el momento.

- **Reducción (Reduce):** El analizador sintáctico realiza una reducción cuando aplica una regla de producción para reemplazar una secuencia de símbolos en la pila por un símbolo no terminal. Esto implica que el analizador sintáctico ha reconocido una estructura sintáctica válida y está simplificando la pila.

  La acción de reducción ocurre cuando en la tabla de acción (actionTable) se encuentra una entrada que indica una reducción. Por lo general, esto se representa mediante el número de la regla de producción que se debe aplicar.

  La reducción permite simplificar la pila y reemplazar una secuencia de símbolos por un símbolo no terminal, que representa una estructura sintáctica más grande y compleja. Al realizar reducciones, el analizador sintáctico combina las estructuras sintácticas parciales en estructuras más grandes.

La decisión de realizar una reducción o desplazamiento se basa en el contenido de la tabla de acción y en el estado actual del analizador sintáctico. Si la entrada actual y la acción en la tabla de acción indican una reducción, se aplica la regla de producción correspondiente. Si la entrada actual y la acción en la tabla de acción indican un desplazamiento, se mueve la entrada a la pila y se avanza al siguiente símbolo.

En el código, el método `generateOutput` es el punto de entrada para generar el resultado. Recibe como parámetro la ruta del archivo de salida y realiza las siguientes tareas:

1. Lee el archivo de salida que contiene la gramática del lenguaje.
2. Genera las constantes de tokens y símbolos llamando a los métodos `generateTokenConstants` y `generateSymbolConstants`.
3. Modifica la gramática para eliminar las producciones lambda llamando al método `removeLambda`.
4. Crea el estado inicial del autómata llamando al método `createState0`.
5. Genera el analizador llamando al método `generateParser`.
6. Genera un archivo de texto que muestra la gramática, los conjuntos `First` y `Follow` de cada regla llamando al método `generateGrammar`.

El método `generateParser` genera la clase **Parser**, que implementa el **analizador sintáctico SLR**. Este método declara e importa las clases necesarias, implementa el constructor de la clase Parser, inicializa las reglas y las tablas de acción (*actionTable*) y de transición (*gotoTable*). Los métodos `generateActionTable` y `generateGotoTable` se encargan de generar las tablas de acción y de transición respectivamente, basándose en los estados y transiciones del autómata.

1. Declara e importa las clases necesarias.
2. Implementa el constructor de la clase Parser.
3. Inicializa las reglas, las tablas de acción (actionTable) y las tablas de transición (gotoTable).

```
1
2  public class Generator {
3      ArrayList<String> symbols;
4      ArrayList<String> tokens;
5      Grammar grammar;
6      Automaton automaton;
7
8      public Generator() {
9          symbols = new ArrayList<String>();
10         tokens = new ArrayList<String>();
11         grammar = new Grammar();
12         automaton = new Automaton(grammar);
13     }
14
15     public void generateOutput(String ruta) throws IOException {
16         // ...
17
18         grammar.removeLambda();
19         automaton.createState0();
20         generateParser();
21         generateGrammar();
22     }
23
24     // ...
25
26     private void generateParser() {
27         // ...
28
29         // Declarar InitRules
30         stream.println("\tprivate void initRules() {\n\t\t" + "int[][] initRule = {\n" + "\t
       \t\t\t{ 0, 0 },");
31         // Reglas
32         for (Rule reg : grammar.rules) {
33             stream.println("\t\t\t\t{ " + reg.identifier + ", " + reg.production.size() + "
       },");
34         }
35         // Cerrar InitRule
36         stream.println("\t\t};\n\n\t\t" + "this.rule = initRule;\n\t}\n");
37
38         // ActionTable
39         stream.println("\tprivate void initActionTable(){");
40         // Declarar ActionTable (size)
41         stream.println("\t\tactionTable = new ActionElement[" + automaton.getStates().size()
        + "][" + tokens.size() + "];");
42         // Informaci n ActionTable a generar
43         for (int i = 0; i < automaton.getStates().size(); i++) {
44             stream.print(generateActionTable(automaton.getStates().get(i), i));
45         }
46         // Cerrar ActionTable
47         stream.println("\t}\n");
48
49         // GotoTable()
50         stream.println("\tprivate void initGotoTable() {");
51         // Declarar GotoTable (size)
```

```
52          stream.println("\t\tgotoTable = new int[" + automaton.getStates().size() + "][" +
    symbols.size() + "];");
53          // Informaci n GotoTable a generar
54          for (int i = 0; i < automaton.getStates().size(); i++) {
55              stream.print(generateGotoTable(automaton.getStates().get(i), i));
56          }
57          // Cerrar GoToTable
58          stream.println("\t}\n");
59
60          // ...
61      }
62
63      private String generateActionTable(State state, int numEstado) {
64          String devolver = "";
65          ArrayList<Transition> transicionesEstado = state.getTransiciones();
66
67          for (Transition transition : transicionesEstado) {
68              if (transition.getSource().terminal) {
69                  // Reducci n
70                  if (transition.isReduce()) {
71                      devolver += generateReduceActionTable(transition, numEstado);
72                  // Desplazamiento
73                  } else {
74                      devolver += "\n\t\tactionTable[" + numEstado + "][" + eliminaCuadrilla(
    transition.getSource().expression) + "] = new ActionElement(ActionElement.SHIFT, " +
    transition.getDestination() + ");";
75                  }
76              }
77          }
78
79          if (!devolver.isEmpty())
80              devolver += "\n";
81
82          return devolver;
83      }
84
85      private String generateReduceActionTable(Transition transition, int numEstado) {
86          String devolver = "";
87
88          // Aceptaci n
89          if (transition.getDestination() == 1) {
90              devolver += "\n\t\tactionTable[" + numEstado + "][TokenConstants.EOF] = new
    ActionElement(ActionElement.ACCEPT, 0);";
91          } else {
92              ArrayList<Integer> prod = transition.getSource().getProduction();
93
94              for (Integer integer : prod) {
95                  devolver += "\n\t\tactionTable[" + numEstado + "][" + eliminaCuadrilla(
    integer.toString()) + "] = new ActionElement(ActionElement.REDUCE, " + transition.
    getDestination() + ");";
96              }
97          }
98
99          return devolver;
100     }
101
102     private String generateGotoTable(State state, int numEstado) {
103         String devolver = "";
104         ArrayList<Transition> transicionesEstado = state.getTransiciones();
105
106         for (Transition transition : transicionesEstado) {
107             if (!transition.getSource().terminal) {
108                 devolver += "\n\t\tgotoTable[" + numEstado + "][" + eliminaCuadrilla(
    transition.getSource().expression) + "] = " + transition.getDestination() + ";";
109             }
110         }
111
112         if (!devolver.isEmpty())
113             devolver += "\n";
114
```

```
115             return devolver;
116         }
117 
118         // ...
119 }
```

Listing 16: Código clase Generator

9. Código de las clases que describen la tabla de desplazamiento/reducción generada por el algoritmo SLR

Estas clases permiten representar y acceder a la tabla de desplazamiento/reducción y las tablas ActionTable y GotoTable generadas por el algoritmo SLR.

9.1. Clase `ActionElement`

En la clase `ActionElement`, se definen tres constantes **SHIFT**, **REDUCE** y **ACCEPT** para representar los tipos de acciones posibles en la tabla. Además, tiene un constructor que recibe el tipo de acción (`type`) y el valor correspondiente (`value`), y métodos para obtener el tipo y el valor.

```
1
2  /*
3   * Elemento de la tabla de accion
4   */
5  public class ActionElement {
6
7          // Constante que identifica un elemento de tipo aceptar
8    public static final int ACCEPT = 0;
9
10         // Constante que identifica un elemento de tipo shift
11   public static final int SHIFT = 1;
12
13   // Constante que identifica un elemento de tipo reduce
14   public static final int REDUCE = 2;
15
16   /**
17    * Tipo de elemento (ACCEPT, SHIFT o REDUCE)
18    */
19   private int type;
20
21   /**
22    * Valor del elemento
23    */
24   private int value;
25
26   public ActionElement(int type, int value)
27   {
28     this.type = type;
29     this.value = value;
30   }
31
32   public int getType()
33   {
34     return type;
35   }
36
37   public int getValue()
38   {
39     return value;
40   }
41 }
```

Listing 17: Clase ActionElement

9.2. Clase `Generator`

Los objetos `ActionElement` definen el tipo de acción a realizar en las tablas. A continuación se incluye el código que genera `ActionTable` y `GotoTable`. Este código ha sido extraído de la clase `Generator`, se muestran aquellas partes encargadas de generar las tablas GotoTable y ActionTable:

9.2.1. Método `generateActionTable`

El método `generateActionTable` se utiliza para generar la tabla de desplazamiento/reducción (`actionTable`) a partir de un estado y un número de estado dado. Recorre las transiciones del estado y, para cada transición que represente un símbolo terminal, verifica si es una reducción o un desplazamiento. Si es una reducción, se invoca el método `generateReduceActionTable` para generar las acciones de reducción correspondientes. Si es un desplazamiento, se agrega la acción de desplazamiento a la tabla `actionTable`.

El método `generateReduceActionTable` se utiliza para generar las acciones de reducción para una transición dada y un número de estado. Si el destino de la transición es -1, lo que indica la regla inicial, se agrega una acción de aceptar (`ActionElement.ACCEPT`). De lo contrario, se obtienen los símbolos siguientes de la regla y se agrega una acción de reducción (`ActionElement.REDUCE`) para cada símbolo siguiente en la tabla `actionTable`.

```
1
2  // ...
3
4  private String generateActionTable(State state, int numEstado) {
5      String devolver = "";
6      ArrayList<Transition> transicionesEstado = state.getTransiciones();
7      for(Transition transition : transicionesEstado) {
8          if(transition.getSource().terminal) {
9              // Reduccion
10             if(transition.isReduce()) {
11                 devolver += generateReduceActionTable(transition, numEstado);
12             // Desplazamiento
13             }else {
14                 devolver += "\n\t\tactionTable["+ numEstado +"]["+ eliminaCuadrilla(
       transition.getSource().expression) +"] = new ActionElement(ActionElement.SHIFT, "+
       transition.getDestination() +");";
15             }
16         }
17     }
18     if(!devolver.isEmpty())
19         devolver += "\n";
20     return devolver;
21 }
22
23 private String generateReduceActionTable(Transition transition, int numEstado) {
24     String devolver = "";
25     if(transition.getDestination() == -1) {
26         devolver +="\n\t\tactionTable["+ numEstado +"][EOF] = new ActionElement(
       ActionElement.ACCEPT, 0);";
27     }else {
28         ArrayList<String> siguientes = grammar.rules.get(transition.getDestination()).
       follows;
29         for(String siguiente : siguientes) {
30             devolver += "\n\t\tactionTable["+ numEstado +"]["+ eliminaCuadrilla(siguiente) +
       "] = new ActionElement(ActionElement.REDUCE, "+ (transition.getDestination()+1) +");";
31         }
32     }
33     return devolver;
34 }
35
36 // ...
```

Listing 18: Método generar ActionTable

9.2.2. Método `generateGotoTable`

El método **`generateGotoTable`** se utiliza para generar la tabla Goto (**`gotoTable`**) a partir de un estado y un número de estado dado. Recorre las transiciones del estado y, para cada transición que represente un símbolo no terminal, agrega la entrada correspondiente a la tabla **`gotoTable`**.

```
1
2  // ...
3
4  private String generateGotoTable(State state, int numEstado) {
5      String devolver = "";
6      ArrayList<Transition> transicionesEstado = state.getTransiciones();
7
8      for(Transition transition : transicionesEstado) {
9          if(!transition.getSource().terminal) {
10             devolver += "\n\t\tgotoTable[" + numEstado + "][" + transition.getSource().
       expression + "] = " + transition.getDestination() + ";";
11         }
12     }
13
14     if(!devolver.isEmpty())
15         devolver += "\n";
16
17     return devolver;
18 }
19
20 // ...
```

Listing 19: Método generar ActionTable

10. Código de las clases que desarrollen la generación de los ficheros "TokenConstants.java", "SymbolConstants.javaz "Parser.javaz explicación de su funcionamiento

En este apartado, se muestra el código de las clases que se encargan de generar los archivos `TokenConstants.java`, `SymbolConstants.java` y `Parser.java`. El objetivo del proyecto es generar estos archivos que son esenciales para el funcionamiento del analizador léxico y sintáctico generado a partir de la gramática y el autómata.

10.1. Código de la clase `Generator`

La clase `Generator` es la clase principal encargada de generar los archivos mencionados. Tiene un constructor que inicializa las listas `symbols` y `tokens`, así como los objetos `grammar` y `automaton`.

El método `generateOutput` es el punto de entrada para la generación de los archivos. Toma como entrada la ruta del archivo de salida y realiza los siguientes pasos:

1. Lee el archivo de salida para obtener la lista de símbolos y tokens.
2. Genera el archivo `TokenConstants.java` llamando al método `generateTokenConstants`.
3. Genera el archivo `SymbolConstants.java` llamando al método `generateSymbolConstants`.
4. Realiza algunas modificaciones en la gramática y el autómata.
5. Genera el archivo `Parser.java` llamando al método `generateParser`.
6. Genera la gramática llamando al método `generateGrammar`.

```

public class Generator {

    ArrayList<String> symbols;
    ArrayList<String> tokens;
    Grammar grammar;
    Automaton automaton;

    public Generator() {
        symbols = new ArrayList<String>();
        tokens = new ArrayList<String>();
        grammar = new Grammar();
        automaton = new Automaton(grammar);
    }

    public void generateOutput(String ruta) throws IOException {
        readOutputFile(ruta);
        generateTokenConstants();
        generateSymbolConstants();

        grammar.removeLambda();
        automaton.createState0();
        generateParser();
        generateGrammar();
    }
```

Listing 20: Código Generator

A continuación se detallan los métodos dentro de la clase `Generator` que se encargan de generar los ficheros antes mencionados.

10.1.1. Código del método `generateSymbolConstants`

El método `generateSymbolConstants` se encarga de generar el archivo `SymbolConstants.java`. Toma la lista de símbolos y genera una interfaz que contiene las constantes correspondientes a cada símbolo. El archivo generado se coloca en el directorio `generated` dentro del proyecto.

El método itera sobre la lista de símbolos y escribe una línea en el archivo para cada símbolo, asignándole un valor numérico único. El valor se incrementa en cada iteración.

```
1
2  private void generateSymbolConstants() {
3
4        System.out.println("Generando SymbolConstants.java");
5
6          File workingdir = workingDir();
7
8          try (PrintStream stream = new PrintStream(new FileOutputStream(new File(workingdir,
     "src/generated/SymbolConstants.java")))) {
9              stream.println("package generated;\n");
10             stream.println("public interface SymbolConstants {\n");
11
12             int i = 0;
13             for (String symbol : symbols) {
14                 stream.println("\t public int " + symbol + " = " + i + ";");
15                 i++;
16             }
17
18             stream.println("\n}");
19
20             // stream.close();
21
22             System.out.println("SymbolConstants.java Generado\n\n");
23
24         } catch (Exception ex) {
25             printError(workingdir, ex);
26         }
27     }
```

Listing 21: Código generateSymbolConstants

10.1.2. Código del método `generateTokenConstants`

El método `generateTokenConstants` se encarga de generar el archivo `TokenConstants.java`. Toma la lista de tokens y genera una interfaz que contiene las constantes correspondientes a cada token. El archivo generado se coloca en el directorio `generated` dentro del proyecto.

El método inicia el archivo escribiendo la declaración de la interfaz y la constante especial `EOF` con el valor 0. Luego, itera sobre la lista de tokens y escribe una línea en el archivo para cada token, asignándole un valor numérico único. El valor se incrementa en cada iteración.

```
1
2  private void generateTokenConstants() {
3
4      System.out.println("Generando TokenConstants.java");
5
6      File workingdir = workingDir();
7
8      try {
9        FileOutputStream outputfile = new FileOutputStream(new File(workingdir, "src/generated
     /TokenConstants.java"));
10       PrintStream stream = new PrintStream(outputfile);
11
12       // package
13       stream.println("package generated;\n");
14
15       // Declarar clase
16       stream.println("public interface TokenConstants {\n");
```

```
      // Incluir siempre al inicio el Token EOF = 0
      stream.println("\t public int EOF = 0;");
      // Al empezar con EOF = 0, iniciar int de tokens a 1
      int i = 1;
      // A adir cada token
      for (String token : tokens) {
        stream.println("\t public int " + token + " = " + i + ";");
        i++;
      }
      stream.println("\n}");

      stream.close();

      System.out.println("TokenConstants.java Generado\n\n");

    } catch (Error err) {
      printError(workingdir, err);
    } catch (Exception ex) {
      printError(workingdir, ex);
    }

  }
```

Listing 22: Código generateTokenConstants

10.1.3. Código del método `generateParser`

El método `generateParser` se encarga de generar el archivo `Parser.java`. Este archivo contiene la implementación de la clase `Parser`, que extiende la clase `SLRParser` y proporciona la funcionalidad para analizar el lenguaje definido por la gramática.

El método inicia el archivo escribiendo la declaración de la clase y el constructor. Luego, genera las llamadas a los métodos `initRules`, `initActionTable` y `initGotoTable` dentro del constructor.

El método `initRules` inicializa la tabla de reglas, donde cada fila representa una regla de la gramática. Luego, el método genera las llamadas a los métodos `generateActionTable` y `generateGotoTable` para inicializar las tablas `actionTable` y `gotoTable`, respectivamente, esto se explica en el apartado 9.

El método `generateActionTable` genera las entradas correspondientes a la tabla `actionTable` a partir de un estado y un número de estado. El método itera sobre las transiciones del estado y genera las entradas de desplazamiento y reducción en función del tipo de transición.

El método `generateGotoTable` genera las entradas correspondientes a la tabla `gotoTable` a partir de un estado y un número de estado. El método itera sobre las transiciones del estado y genera las entradas de transición en función del tipo de transición.

```

public void generateParser() {

    System.out.println("Generando Parser.java");

    File workingdir = workingDir();

    try {
      FileOutputStream outputfile = new FileOutputStream(new File(workingdir, "src/generated
    /Parser.java"));
      PrintStream stream = new PrintStream(outputfile);

      // package
      stream.println("package generated;\n");

      // imports
      stream.println("import auxiliares.ActionElement;");
```

```
17      stream.println("import auxiliares.SLRParser;\n\n");
18
19      // Declarar clase
20      stream.println("public class Parser extends SLRParser implements TokenConstants,
    SymbolConstants {\n\n "
21            + "\tpublic Parser() {\r\n\t\tinitRules();\r\n\t\tinitActionTable();\r\n\t\
    tinitGotoTable();\r\n\t}\n");
22
23
24      // Declarar InitRules
25      stream.println("\t"
26          + "private void initRules() {\n\t\t"
27          + "int[][] initRule = {\n"
28          + "\t\t\t\t{ 0, 0 },");
29
30      // Reglas
31      for (Rule reg : grammar.rules) {
32        stream.println("\t\t\t\t{ " + reg.identifier + ", " + reg.production.size() + " },")
    ;
33      }
34
35      // Cerrar InitRule
36      stream.println("\t\t};\n\n\t\t"
37          + "this.rule = initRule;\n\t}\n");
38
39      // ActionTable
40      stream.println("\tprivate void initActionTable(){");
41        //  Declarar ActionTable (size)
42      stream.println("\t\tactionTable = new ActionElement["+ automaton.getStates().size() +"
    ]["+ tokens.size() +"];");
43        // Informaci n ActionTable a generar
44      for (int i=0; i< automaton.getStates().size() ; i++) {
45        stream.print(generateActionTable(automaton.getStates().get(i), i));
46      }
47      // Cerrar ActionTable
48      stream.println("\t}\n");
49
50      // TODO esta tabla podr  tener los enteros diferentes ya que depende de la
    implementaci n , pero el resto de valores deben ser iguales (depende de los estados del
    autom)
51      // GotoTable()
52      stream.println("\tprivate void initGotoTable() {");
53        // Declarar GotoTable (size)
54      stream.println("\t\tgotoTable = new int[" + automaton.getStates().size() + "][" +
    symbols.size() + "];");
55        // Informaci n GotoTable a generar
56      for (int i=0; i< automaton.getStates().size() ; i++) {
57        stream.print(generateGotoTable(automaton.getStates().get(i), i ));
58      }
59      // Cerrar GoToTable
60      stream.println("\t}\n");
61
62      // Cerrar clase
63      stream.println("}");
64      stream.close();
65
66      System.out.println("Parser.java Generado\n\n");
67
68    } catch (Error err) {
69      printError(workingdir, err);
70    } catch (Exception ex) {
71      printError(workingdir, ex);
72    }
73
74  }
```

Listing 23: Código generateTokenConstants

11. Pruebas del funcionamiento de la aplicación

En esta sección, se presentan las pruebas realizadas para verificar el correcto funcionamiento de la aplicación de generación de archivos `TokenConstants`, `SymbolConstants` y `Parser`.

11.1. Prueba con el archivo `main.txt`

Se realizó una prueba utilizando el archivo `main.txt` proporcionado en el enunciado del proyecto. Este archivo contiene la definición de una gramática para expresiones aritméticas, junto con reglas para los términos, factores y argumentos. Su contenido es el que se puede encontrar en 1.

El resultado esperado de esta prueba es la generación de los archivos `TokenConstants.java`, `SymbolConstants.java` y `Parser.java` con los contenidos correctos.

A continuación, se muestran los contenidos obtenidos para cada uno de los archivos generados:

11.1.1. Archivo `TokenConstants.java`

El archivo `TokenConstants.java` contiene las constantes correspondientes a los tokens utilizados en la gramática. A continuación se muestra el contenido obtenido:

```
package generated;

public interface TokenConstants {

    public int EOF = 0;
    public int PLUS = 1;
    public int MINUS = 2;
    public int PROD = 3;
    public int DIV = 4;
    public int NUM = 5;
    public int LPAREN = 6;
    public int RPAREN = 7;
    public int IDENTIFIER = 8;
    public int COMMA = 9;

}
```

El contenido coincide con el esperado y se generaron correctamente las constantes correspondientes a cada token definido en la gramática.

11.1.2. Archivo `SymbolConstants.java`

El archivo `SymbolConstants.java` contiene las constantes correspondientes a los símbolos no terminales utilizados en la gramática. A continuación se muestra el contenido obtenido:

```
package generated;

public interface SymbolConstants {

    public int Expr = 0;
    public int Term = 1;
    public int Factor = 2;
    public int Args = 3;
    public int ArgumentList = 4;

}
```

El contenido coincide con el esperado y se generaron correctamente las constantes correspondientes a cada símbolo no terminal definido en la gramática.

11.1.3. Archivo `Parser.java`

El archivo `Parser.java` contiene la implementación de la clase `Parser`, que extiende la clase `SLRParser` y proporciona la funcionalidad para analizar el lenguaje definido por la gramática. A continuación se muestra el contenido obtenido:

```
package generated;

import auxiliares.ActionElement;
import auxiliares.SLRParser;


public class Parser extends SLRParser implements TokenConstants, SymbolConstants {

  public Parser() {
    initRules();
    initActionTable();
    initGotoTable();
  }

  private void initRules() {
    int[][] initRule = {
        { 0, 0 },
        { Expr, 1 },
        { Expr, 3 },
        { Expr, 3 },
        { Term, 1 },
        { Term, 3 },
        { Term, 3 },
        { Factor, 1 },
        { Factor, 3 },
        { Factor, 4 },
        { Args, 1 },
        { Args, 0 },
        { ArgumentList, 1 },
        { ArgumentList, 3 },
    };

    this.rule = initRule;
  }

  private void initActionTable(){
    actionTable = new ActionElement[24][9];

    actionTable[0][NUM] = new ActionElement(ActionElement.SHIFT, 6);
    actionTable[0][LPAREN] = new ActionElement(ActionElement.SHIFT, 7);
    actionTable[0][IDENTIFIER] = new ActionElement(ActionElement.SHIFT, 14);

    actionTable[1][EOF] = new ActionElement(ActionElement.ACCEPT, 0);
    actionTable[1][PLUS] = new ActionElement(ActionElement.SHIFT, 2);
    actionTable[1][MINUS] = new ActionElement(ActionElement.SHIFT, 10);

    actionTable[2][NUM] = new ActionElement(ActionElement.SHIFT, 6);
    actionTable[2][LPAREN] = new ActionElement(ActionElement.SHIFT, 7);
    actionTable[2][IDENTIFIER] = new ActionElement(ActionElement.SHIFT, 14);

    actionTable[3][PROD] = new ActionElement(ActionElement.SHIFT, 4);
    actionTable[3][DIV] = new ActionElement(ActionElement.SHIFT, 12);

    actionTable[4][NUM] = new ActionElement(ActionElement.SHIFT, 6);
    actionTable[4][LPAREN] = new ActionElement(ActionElement.SHIFT, 7);
    actionTable[4][IDENTIFIER] = new ActionElement(ActionElement.SHIFT, 14);

    actionTable[7][NUM] = new ActionElement(ActionElement.SHIFT, 6);
    actionTable[7][LPAREN] = new ActionElement(ActionElement.SHIFT, 7);
    actionTable[7][IDENTIFIER] = new ActionElement(ActionElement.SHIFT, 14);

    actionTable[8][RPAREN] = new ActionElement(ActionElement.SHIFT, 9);
    actionTable[8][PLUS] = new ActionElement(ActionElement.SHIFT, 2);
    actionTable[8][MINUS] = new ActionElement(ActionElement.SHIFT, 10);
```

```
65
66      actionTable[10][NUM] = new ActionElement(ActionElement.SHIFT, 6);
67      actionTable[10][LPAREN] = new ActionElement(ActionElement.SHIFT, 7);
68      actionTable[10][IDENTIFIER] = new ActionElement(ActionElement.SHIFT, 14);
69
70      actionTable[11][PROD] = new ActionElement(ActionElement.SHIFT, 4);
71      actionTable[11][DIV] = new ActionElement(ActionElement.SHIFT, 12);
72
73      actionTable[12][NUM] = new ActionElement(ActionElement.SHIFT, 6);
74      actionTable[12][LPAREN] = new ActionElement(ActionElement.SHIFT, 7);
75      actionTable[12][IDENTIFIER] = new ActionElement(ActionElement.SHIFT, 14);
76
77      actionTable[14][LPAREN] = new ActionElement(ActionElement.SHIFT, 15);
78
79      actionTable[15][NUM] = new ActionElement(ActionElement.SHIFT, 6);
80      actionTable[15][LPAREN] = new ActionElement(ActionElement.SHIFT, 7);
81      actionTable[15][IDENTIFIER] = new ActionElement(ActionElement.SHIFT, 14);
82
83      actionTable[16][RPAREN] = new ActionElement(ActionElement.SHIFT, 17);
84
85      actionTable[18][COMMA] = new ActionElement(ActionElement.SHIFT, 19);
86
87      actionTable[19][NUM] = new ActionElement(ActionElement.SHIFT, 6);
88      actionTable[19][LPAREN] = new ActionElement(ActionElement.SHIFT, 7);
89      actionTable[19][IDENTIFIER] = new ActionElement(ActionElement.SHIFT, 14);
90
91      actionTable[20][PLUS] = new ActionElement(ActionElement.SHIFT, 2);
92      actionTable[20][MINUS] = new ActionElement(ActionElement.SHIFT, 10);
93
94      actionTable[21][PROD] = new ActionElement(ActionElement.SHIFT, 4);
95      actionTable[21][DIV] = new ActionElement(ActionElement.SHIFT, 12);
96
97      actionTable[23][PLUS] = new ActionElement(ActionElement.SHIFT, 2);
98      actionTable[23][MINUS] = new ActionElement(ActionElement.SHIFT, 10);
99    }
100
101   private void initGotoTable() {
102     gotoTable = new int[24][5];
103
104     gotoTable[0][Expr] = 1;
105     gotoTable[0][Term] = 21;
106     gotoTable[0][Factor] = 22;
107
108     gotoTable[2][Term] = 3;
109     gotoTable[2][Factor] = 22;
110
111     gotoTable[4][Factor] = 5;
112
113     gotoTable[7][Expr] = 8;
114     gotoTable[7][Term] = 21;
115     gotoTable[7][Factor] = 22;
116
117     gotoTable[10][Term] = 11;
118     gotoTable[10][Factor] = 22;
119
120     gotoTable[12][Factor] = 13;
121
122     gotoTable[15][Args] = 16;
123     gotoTable[15][ArgumentList] = 18;
124     gotoTable[15][Expr] = 23;
125     gotoTable[15][Term] = 21;
126     gotoTable[15][Factor] = 22;
127
128     gotoTable[19][Expr] = 20;
129     gotoTable[19][Term] = 21;
130     gotoTable[19][Factor] = 22;
131   }
132
133 }
```

El contenido coincide con el esperado, aunque se observa una diferencia en los números de las tablas `actionTable` y `gotoTable` debido a un posible ordenamiento diferente de los símbolos terminales y no terminales durante la implementación. Sin embargo, esta diferencia no afecta el funcionamiento del analizador.

En conclusión, la prueba con el archivo `main.txt` ha sido exitosa, generando correctamente los archivos `TokenConstants`, `SymbolConstants` y `Parser` con los contenidos esperados.

12. Conclusiones

En este proyecto, se ha desarrollado una aplicación que cumple con el objetivo de generar un Analizador Sintáctico Ascendente a partir de la descripción de una gramática en notación BNF. La aplicación toma como entrada la gramática y calcula la tabla de desplazamiento/reducción utilizando el algoritmo SLR. Como resultado, se generan los archivos `TokenConstants.java`, `SymbolConstants.java` y `Parser.java`, que implementan el analizador sintáctico para la gramática especificada.

12.1. Sencillez en la generación de GrammarLexer

La generación de la clase `GrammarLexer`, encargada de reconocer y clasificar los tokens de la gramática, resultó ser una tarea sencilla. Esto se debe a que los estados del autómata correspondiente al lexer se generaron manualmente y se basaron en la estructura de la gramática en notación BNF. Haber hecho el autómata de forma manual y contar con los estados ya estudiados permitió implementar rápidamente el lexer y obtener los tokens necesarios para el análisis sintáctico.

12.2. Dificultad en la generación de las tablas de análisis

Una de las partes más desafiantes del proyecto ha sido la generación de las tablas de desplazamiento/reducción (*actionTable* y *gotoTable*) específicas para la gramática de entrada. Esta tarea requería comprender en profundidad el algoritmo SLR y aplicarlo correctamente. La generación de estas tablas implica considerar los conjuntos de elementos LR(0) y LR(1), calcular los estados y transiciones del autómata LR(1) y finalmente construir las tablas correspondientes. Este proceso resultó complejo y requirió tiempo y esfuerzo para asegurar la corrección y eficiencia de las tablas generadas.

12.3. Extracción sencilla de TokenConstants y SymbolConstants

La generación de los archivos `TokenConstants.java` y `SymbolConstants.java`, que contienen las constantes correspondientes a los símbolos terminales y no terminales de la gramática, resultó ser una tarea sencilla. Una vez implementado el autómata y el `GrammarLexer`, la extracción de los símbolos terminales y no terminales fue directa, ya que el estado final del lexer determinaba el tipo de token correspondiente. Esto facilitó la obtención de los símbolos necesarios para la implementación del analizador sintáctico.

12.4. Acceso a materiales de apoyo

Durante el desarrollo del proyecto, se tuvo acceso a prácticas y materiales de la asignatura que resultaron ser muy útiles. Muchas clases utilizadas en la implementación se basaron en códigos proporcionados en las prácticas, o bien se utilizaron códigos de las prácticas con modificaciones mínimas. Esto permitió aprovechar los conocimientos adquiridos previamente y agilizar el proceso de implementación.

Los materiales de apoyo se pueden encontrar en la página web de la asignatura [2], [3], [4].

En resumen, el desarrollo de este proyecto ha permitido alcanzar el objetivo propuesto, que consistía en desarrollar una aplicación capaz de **generar un Analizador Sintáctico Ascendente a partir de una gramática en notación BNF**. A lo largo del proyecto, se ha enfrentado y superado la dificultad de generar las tablas de análisis específicas para la gramática, al tiempo que se han identificado las partes más sencillas del proceso, como la generación de `GrammarLexer` y la extracción de los símbolos terminales y no terminales. Además, se ha aprovechado el acceso a materiales de apoyo que han facilitado la implementación del proyecto.

Este trabajo ha demostrado la importancia del análisis sintáctico en el proceso de compilación y la complejidad asociada a la generación automática de analizadores sintácticos. Al comprender y aplicar conceptos como la gramática, el algoritmo SLR y el autómata LR(1), se ha logrado implementar un analizador sintáctico que puede ser utilizado como herramienta para el análisis de diferentes gramáticas y lenguajes de programación.

Aunque existen herramientas como ***JavaCC*** que automatizan gran parte de este proceso, la implementación manual tiene sus ventajas. Al realizarlo manualmente, se adquiere una mejor comprensión de

los conceptos necesarios para implementar un Analizador Sintáctico, como los conjuntos LR(0) y LR(1), el autómata de análisis y las tablas de desplazamiento/reducción. Sin embargo, implementar manualmente lo que estas herramientas automatizan requiere considerable tiempo y esfuerzo. Además, las herramientas como ***JavaCC*** ofrecen una sintaxis más concisa y un proceso más rápido para generar analizadores sintácticos, lo que resulta muy útil en proyectos más grandes y complejos.

Referencias

[1] Francisco José Moreno Velo, "Procesadores de Lenguajes (GII-PL) - Trabajo." http://www.uhu.es/francisco.moreno/gii_pl/trabajo/trabajo.htm. Accedido en junio de 2023.

[2] Francisco José Moreno Velo, "Procesadores de Lenguajes (GII-PL)." http://www.uhu.es/francisco.moreno/gii_pl/. Accedido en junio de 2023.

[3] Francisco José Moreno Velo, "Procesadores de Lenguajes (GII-PL) - Práctica 2." http://www.uhu.es/francisco.moreno/gii_pl/practicas/practica02.htm. Accedido en junio de 2023.

[4] Francisco José Moreno Velo, "Procesadores de Lenguajes (GII-PL) - Práctica 7." http://www.uhu.es/francisco.moreno/gii_pl/practicas/practica07.htm. Accedido en junio de 2023.