Teoría de la Computación

TRABAJO INTEGRADOR PARSER

DOCUMENTACIÓN TÉCNICA

Miembros:

Sánchez Octavio Alfredo - 44622119

Mieres Julián Edgardo - 41442242

Freixes Matias - 43822966

Descripción del Lenguaje Definido

En este trabajo nos empeñamos en desarrollar un mini-lenguaje que permite ejecutar consultas booleanas para poder usarse en sistemas de búsqueda, similares a los que se utilizan en motores como Google o similares, en filtrado de datos y en interfaces conversacionales, usando los operadores lógicos AND, OR y NOT tanto coloquial como verbalmente y agrupaciones anidadas y estructuradas entre paréntesis y comillas.

Gramática utilizada

La gramática utilizada para lo construcción de este mini-lenguaje es la siguiente:

<consulta> ::= <expresion>

<expresion> ::= <expresion> AND <expresion>

| <expresion> OR <expresion>

| NOT <expresion>

| "(" <expresion> ")"

| <termino>

<termino> ::= <palabra> | <frase>

<frase> ::= "" <caracteres> ""

<palabra> ::= secuencia de letras y/o números

La gramática permite reconocer combinaciones lógicas arbitrariamente anidadas, con precedencia controlada mediante paréntesis y operadores explícitos. También se aceptan equivalentes simbólicos como &&, || y !.

Elección del tipo de parser

Se implementó un parser descendente recursivo en Python, debido a la claridad que ofrece para gramáticas de complejidad media como esta, y la facilidad de manipulación directa del árbol sintáctico. El parser procesa los tokens generados por un analizador léxico personalizado, y construye un árbol jerárquico que representa la estructura lógica de la consulta. Se incorporó manejo de errores y validaciones sintácticas, con retroalimentación clara en caso de entradas incorrectas.

Breve explicación del código y las estructuras utilizadas.

A continuación, pasaremos a explicar un poco el código como vemos decidimos usar C# para realizar el parse esto por distintos motivos, pero los principales fueron que nos permite crear un ejecutable para obtener una mayor portabilidad del código sin estar dependiendo de poseer un entorno para ejecutar el código y la segunda es por su gran facilidad para realizar interfaces

Explicare un poco las clases que encontramos dentro del analizador sintáctico

1. Clase Nodo (AST / Árbol de Derivación) **Archivo:Nodo.cs**

EL propósito de esta clase es representar cada vértice del árbol sintáctico tanto los no terminales (<CONSULTA>, <EXP>, <TERMINO>) como los terminales (operadores, palabras, frases).

Como atributos de la clase tendremos

- Tipo (string): el nombre del símbolo (por ejemplo, "<EXP>", "AND", "palabra").
- Valor (string, opcional): el lexema real cuando es un token hoja (p. ej. texto de una frase o palabra).
- Hijos (List<Nodo>): subnodos derivados de esa producción.

Tendremos únicamente un método llamado Mostrar(nivel):

Este recorre recursivamente el árbol y indenta según la profundidad (nivel) y escribe primero el Tipo. Si tiene Valor y ningún hijo, lo imprime como hoja si no, despliega sus hijos.

La estructura es un árbol genérico de nodos etiquetados, ideal para visualizar la derivación o el AST.

2. Clase MiParser (Lexer + Parser Descendente) **Archivo:MiParse.cs**

Explciaremos como funciona cada parte de esta clase

- Lexer (Lexer(string entrada))

  En esta clase hacemos uso de una expresión regular para segmentar la cadena en tokens: Frases ("[^\"]*"), paréntesis, operadores (&&,||,!), palabras (\ w+).

  Tambien se normaliza los operadores a mayúsculas para evitar que se rechazen por estar en minúscula ("AND", "OR", "NOT").

  Acá se devuelve List<string> con la secuencia de tokens.

- Parser (clase interna Parser)

  Implementa un parser recursivo descendente que sigue la gramática BNF y tiene lo siguiente metodos:

- 1.Método Consulta()

  Regla <CONSULTA> ::= <EXP>.

  Invoca Exp(), envuelve el resultado en un nodo "<CONSULTA>".

- 2.Método Exp()

  Cubre las producciones:

  NOT <EXP>

  (<EXP>)

  <TERMINO>

  Y luego, en un bucle, gestiona secuencias izquierda-asociativas de AND / OR:

  ```
  while (Actual()=="AND"||Actual()=="OR") {
          Coincidir(op);
          var right = Exp();
          nodo = new Nodo("<EXP>", null, [ nodo, new Nodo(op), right ]);
  }
  ```

  Cada vez que coincide AND u OR, crea un nuevo <EXP> con los dos operandos y el operador intermedio.

- 3.Método Termino()

  Regla <TERMINO> ::= frase | palabra.

  Detecta si el token actual es cadena entre comillas → crea nodo ("frase",valor);

  si es alfanumérico → nodo ("palabra",valor);

  en caso contrario arroja excepción.

- 4.Método Coincidir(string esperado)

  Avanza pos si el token coincide;

  Si no, lanza SyntaxErrorException con mensaje claro (token esperado, token encontrado y posición).

3. Clase GraphvizExporter **Archivo:GraphvizExporter.cs**

Esta clase toma un Nodo raíz y generar un diagrama Graphviz (“dot”) (herramienta que importamos para generar automaticamente los diagramas) que luego se compila a PNG y se puede exportar la secuencia,la estructura jerárquica y el PNG del árbol generado.

Funciona de la siguiente manera:

- Primero GenerarDot(Nodo) — recorre el árbol y compone el texto del .dot.
- Escribe el .dot en disco.
- Llama a dot.exe (ubicado en graphviz_bin) con argumentos para producir el .png.
- Devuelve la ruta del .png.

Se usa ProcessStartInfo para invocar dot.exe sin ventana.

Por ultimo en GenerarNodos(), se asigna IDs únicos (n0, n1, …) y este dibuja aristas entre padre e hijo.

Lecciones aprendidas.

Durante el desarrollo de este proyecto de analizador sintáctico para expresiones booleanas, aprendimos la importancia de definir una gramática clara y bien estructurada. También hicimos uso de expresiones regulares, temas vistos a lo largo de la asignatura.

Al principio subestimamos el nivel de detalle necesario para modelar correctamente todas las combinaciones válidas de operadores y paréntesis.

Otra lección importante fue la relación entre la gramática y el diseño del parser: vimos cómo pequeñas ambigüedades o definiciones incompletas complicaban la construcción del árbol de derivación. Aprendimos a analizar recursivamente siguiendo la estructura de la gramática y a respetar la precedencia y la asociatividad de los operadores.

También fue muy valioso integrar herramientas externas como Graphviz para la generación automática de diagramas, y aprovechar canales de comunicación como Discord o WhatsApp para mantener un diálogo fluido y organizarnos mejor.

Finalmente, reforzamos la importancia de diseñar casos de prueba variados, tanto válidos como con errores, para validar la robustez del analizador y mejorar los mensajes de error. Esto nos ayudó a entregar un sistema más completo y fácil de usar. Sobre todo, queremos resaltar la importancia de realizar estos tipos de proyectos, ya que nos dio una dimensión de lo que se puede llegar a lograr y cómo los analizadores están presentes en muchos sistemas que usamos hoy en día.