UNIVERSIDAD MARIANO GALVEZ DE GUATEMALA
FACULTAD DE INGENIERIA EN SISTEMAS
COMPILADORES
ING. MIGUEL CATALAN

NOMBRE	NUMERO DE CARNE
Angel Enrique Juarez Castellanos	7590-21-21054

Guatemala, 31/05/2024

COMPILADOR MINILAP

Objetivos de la aplicación:

- Entender a fondo la funcionalidad de un analizador léxico y sintáctico que serian sus dos primeras fases.
- Lograr crear un analizador léxico que pueda entender los patrones y lexemas que ingresa el usuario.
- Lograr crear un analizador sintáctico que contenga las gramáticas correspondientes al lenguaje Minilab siendo fáciles de entender e interpretar.
- Integrar funcionalidad a la gramática definida para poder realizar operaciones, funciones, procedimientos etc.

Funcionalidad

El programa tiene como flujo básico ingresar código minilab en una caja de texto y al darle en el botón de compilar este recibirá el texto desde un controlador el cual se encargará de analizar léxica y sintácticamente el texto ingresado dándole la funcionalidad que escribió el usuario. El programa tiene la capacidad de devolver problemas léxicos o sintácticos según corresponda.

Especificación Técnica

Requisitos de Hardware

- **Procesador**: Cualquier procesador moderno de al menos 1 GHz. Se recomienda un procesador multinúcleo para un mejor rendimiento.
- **RAM**: Mínimo 4 GB. Para un rendimiento óptimo, especialmente si se ejecutan otros programas simultáneamente, se recomienda 8 GB o más.
- **Disco Duro**: Mínimo 10 GB de espacio disponible. Esto incluye espacio para el sistema operativo, el entorno de desarrollo (IDE), y archivos de proyecto. Se recomienda un SSD para tiempos de acceso más rápidos.

Requisitos de Software:

- **Sistema Operativo**: Windows 7/8/10/11, macOS 10.12 o superior, o una distribución de Linux reciente (Ubuntu 18.04 o superior, Fedora, etc.).
- **Java Development Kit (JDK)**: JDK 8 o superior. Asegúrate de tener instalada una versión compatible con JFlex y CUP.
- **Entorno de Desarrollo Integrado (IDE)**: IntelliJ IDEA, Eclipse, NetBeans o cualquier otro IDE compatible con Java. El IDE debe estar configurado para trabajar con JFlex y CUP.
- **JFlex**: 1.9.1
- **CUP**: 1.3

Herramientas Adicionales:

- **Maven** (opcional, pero recomendado) para la gestión de dependencias y la construcción del proyecto.
- **Git** para el control de versiones, si se trabaja en un entorno de desarrollo colaborativo.

ARCHIVO JFLEX

Lexer.jflex: Un archivo de especificaciones JFlex define cómo se debe analizar el texto de entrada para identificar tokens (símbolos léxicos) y generar un escáner léxico. Este archivo sigue una estructura específica que incluye varias secciones.

Encabezado y Definiciones:

Estas líneas indican el paquete al que pertenece la clase generada y los imports necesarios.package

- **mi.primer.scanner;**
- **import util.*;**
- **import java.util.LinkedList;**
- **import java_cup.runtime.*;**

Opciones y Definiciones Regulares:

- **`%public`:** Hace que la clase generada sea pública.
- **`%class Scanner`:** Define el nombre de la clase generada (`Scanner`).
- **`%char`, `%line`, `%column`:** Habilitan el seguimiento del número de caracteres, líneas y columnas en el texto de entrada.
- **`%cup`:** Indica que se generarán objetos `Symbol` para usar con el parser `CUP`.

Código Java Incluido

```
%{
    public  static LinkedList < String > TablaEL =  new LinkedList < String >();
    public LinkedList < String >  getTablaEL (){
        return TablaEL ;
    }

    public  void  limpiarTablaEL (){
        TablaEL.clear();
    }
     Símbolo privado símbolo ( int tipo , valor del objeto )  {
        return  nuevo  símbolo ( tipo , yyline , yycolumn , valor );
    }
%}
```

Esta sección del código tiene métodos que utilizaremos al generar el analizador léxico así como atributos.

Definiciones Regulares

```
palabra =  [ a - zA - Z ]+
identificador =  [ a - zA - Z ][ a - zA - Z0 - 9 ]*
digito =  -?[ 0 - 9 ]+
espacios_blanco =  [ \ r | \ norte | \ r \ n |  | \ t ]
d_float   =  -?[ 0 - 9 ]+( \ .[ 0 - 9 ]+)?
texto =  \ "[^\"\\]+\"
```

Estas definiciones regulares son utilizadas por el analizador léxico para dividir el texto de entrada en tokens, que son unidades léxicas reconocidas por el compilador. Cada token representa una parte específica del código fuente, como palabras clave, identificadores, números, etc.

REGLAS LEXICAS

```
"si"       { Sistema . afuera . println ( "Lexema: "
              + yytext ()
              +  " columna: "
              + yychar
               +  " fila: "
              + yyline ); símbolo de retorno ( sym . SI , yytext ());  }
```

Las palabras reservervadas como esta van entre dos comillas ya que literalmente recibiremos ese texto e indicando en que columna y fila se encuentra la misma, así como el Símbolo que retornan.

```
{identificador}  { Sistema . afuera . println ( "Id. de lexema: "
              + yytext ()
              +  " columna: "
              + yychar
               +  " fila: "
              + yyline );  símbolo de retorno ( sym . ID , yytext ());}
```

Para indicarle a nuestro analizador léxico que un lexema estará enlazado con un expresión regular es necesario indicarlo entre llave {} y el nombre de la expresión regular de igual manera se indica que símbolo retornara.

Manejo de Errores

```
.               { System.out.println("Error Lexico"+yytext()+" Linea "+yyline+" Columna
"+yycolumn);
                          String datos = new TError(yytext(),yyline,yycolumn,"Error
Lexico","Simbolo no existe en el lenguaje").toString();
                          TablaEL.add(datos);}
```

El punto (.) representa cualquier carácter no reconocido por las reglas anteriores. Se registra como un error léxico.

ARCHIVO CUP

Analizador Sintáctico:

El archivo .cup contiene la especificación de la gramática del lenguaje, definiendo las reglas de producción que describen cómo se construyen las diferentes estructuras sintácticas, como expresiones y declaraciones. También asocia acciones semánticas con estas reglas para realizar tareas como la construcción de un árbol de sintaxis abstracta o la generación de código intermedio. A partir de este archivo, se genera un parser que, junto con el analizador léxico, analiza el código fuente y realiza el proceso de compilación, interpretando su estructura sintáctica y aplicando las acciones semánticas correspondientes.

- **`import java_cup.runtime.*;`**: Importa todas las clases del paquete `java_cup.runtime`, que contiene las clases necesarias para la ejecución del parser generado por CUP. Esto incluye la clase `Symbol`, que representa los tokens del análisis léxico, y otras utilidades relacionadas con el análisis sintáctico.

- **`import java.util.ArrayList;`** y **`import java.util.List;`**: Importan las clases `ArrayList` y `List` de Java, que se utilizan para manejar listas de elementos en el compilador. Estas clases son esenciales para gestionar estructuras de datos dinámicas durante el análisis y la generación de código.

- **`import util.*;`**: Importa todas las clases del paquete `util`, que presumiblemente contiene utilidades específicas del proyecto, como clases para manejar errores, tablas de símbolos, o cualquier otra funcionalidad auxiliar que se haya definido en el paquete `util`.

- **`import javax.script.ScriptEngine;`**, **`import javax.script.ScriptEngineManager;`**, y **`import javax.script.ScriptException;`**: Importan clases del paquete `javax.script`, que proporciona la capacidad de ejecutar scripts escritos en lenguajes de scripting desde Java. En el contexto de un compilador, esto podría utilizarse para ejecutar expresiones o fragmentos de código durante el análisis sintáctico o semántico.

```
parser code {:
    public ArrayList<Object> resultados = new ArrayList<>();

    public ArrayList<Variables> variables = new ArrayList<>();
    public String error = "";
    public Condicion condicion = new Condicion();
    public Variables variable = new Variables();
    public InstruccionesIf instruccionesIf = new InstruccionesIf();
    public Imprimir imprimir = new Imprimir();
```

```
     public ArrayList<String> consola = imprimir.getConsola();

     public void limpiarConsola(){
         imprimir.listaImpri.clear();
     }
    public void syntax_error(Symbol s){
            error = "Error Sintáctico en la Línea " + (s.left) +" Columna
"+s.right+ ". No se esperaba este componente: " +s.value+".";
            System.err.println(error);
            consola.add(error);
    }

    public void unrecovered_syntax_error(Symbol s) throws
java.lang.Exception{
            System.err.println("Error síntactico irrecuperable en la Línea
" + (s.left)+ " Columna "+s.right+". Componente " + s.value + " no
reconocido.");
    }

:};
```

- **public ArrayList<Object> resultados = new ArrayList<>();**:
  - Una lista para almacenar los resultados del análisis sintáctico.
- **public ArrayList<Variables> variables = new ArrayList<>();**:
  - Una lista para almacenar las variables que se encuentran durante el análisis.
- **public String error = "";**:
  - Una cadena para almacenar mensajes de error.
- **public Condicion condicion = new Condicion();**:
  - Un objeto para manejar condiciones.
- **public Variables variable = new Variables();**:
  - Un objeto para manejar una variable individual
- **public InstruccionesIf instruccionesIf = new InstruccionesIf();**:
  - Un objeto para manejar instrucciones "if" (presumiblemente una clase personalizada).
- **public Imprimir imprimir = new Imprimir();**:
  - Un objeto para manejar la impresión (presumiblemente una clase personalizada).
- **public ArrayList<String> consola = imprimir.getConsola();**:
  - Una lista para almacenar los mensajes que se imprimen en la consola, obtenidos a través del objeto imprimir.

Manejo de errores CUP

syntax_error(Symbol s)

Este método se invoca cuando se detecta un error sintáctico recuperable durante el análisis del código. Un error sintáctico recuperable es un error del cual el parser puede continuar el análisis después de haberlo encontrado.

`unrecovered_syntax_error(Symbol s) throws java.lang.Exception`

Este método se invoca cuando se detecta un error sintáctico irrecuperable, es decir, un error tan severo que el parser no puede continuar con el análisis del código.

Declaración de Terminales y No Terminales en CUP

En un archivo CUP (similar a un archivo de gramática para un parser), se definen terminales y no terminales para especificar la estructura del lenguaje que se está analizando. Aquí se hace uso de terminales y no terminales para describir las partes básicas del lenguaje y su estructura.

Terminales

Los terminales son los tokens básicos que el lexer (analizador léxico) identifica en el código fuente. Son las "hojas" del árbol sintáctico y representan las unidades más pequeñas del lenguaje (como palabras clave, operadores, identificadores, etc.).

Tipos básicos y literales:

- NUM (números enteros)
- ID (identificadores)
- TIPO (tipos de datos)
- TEXTO (cadenas de texto)
- FLOTANTE (números de punto flotante)

Operadores y símbolos especiales:

- MULTI, SUMA, RESTA, DIVIDIR, ASIGNAR (operadores aritméticos y de asignación)

- MAYOR, MENOR, IGUAL, DIFERENTE, MAYOR_IGUAL, MENOR_IGUAL (operadores de comparación)
- PUNTO_COMA, COMA, PARENTESIS_IZQ, PARENTESIS_DER (símbolos de puntuación)

Palabras clave del lenguaje:

- FLOT, ENT, CADENA (declaración de tipos de variables)
- IMPRMIR (palabra clave para imprimir)
- FUNCTION, SI, ENTONCES, FIN_SI, SINO (control de flujo y definición de funciones)
- RETURN, RETORNO, FIN_FUNCION (control de flujo en funciones)
- HACER, FINPROCEDIMIENTO, PROCEDIMIENTO, FINMIENTRAS, MIENTRAS (control de flujo en procedimientos y bucles)

No Terminales

Estructura básica del programa:

- inicio: Punto de entrada del programa.
- procedimiento: Definición de procedimientos.
- si: Sentencia condicional if.

Expresiones y tipos:

- valor: Representa un valor en el programa.
- tipo: Tipo de datos.
- declaracion: Declaración de variables.
- printP, print: Instrucciones de impresión.

Operaciones y condiciones:

- operaciones, operador, comparar, operacionesP: Operaciones aritméticas y comparaciones.
- condicionP: Parte de una condición.

Parámetros e instrucciones:

- parametros: Parámetros en funciones o procedimientos.
- intrucciones: Conjunto de instrucciones.
- concatenar, concatenarp: Concatenación de cadenas.
- declaracionP: Parte de una declaración.

Funciones y métodos:

- funcion: Definición de funciones.

- condicion: Condiciones booleanas.

Control de flujo:

- ifP, if: Sentencias if.
- ciclo: Bucles (while, for).
- instruccionesif, instruccionesifElse: Instrucciones dentro de un if o if-else.

Reglas de Producción

Las reglas de producción especifican cómo se pueden combinar los terminales y no terminales para formar estructuras válidas del lenguaje. Cada regla de producción puede tener acciones asociadas que se ejecutan cuando la regla se aplica. Estas acciones suelen modificar el estado del programa o construir partes del árbol sintáctico.

1. **Inicio del Programa**
    - El punto de entrada es la producción inicio, que espera encontrar una serie de instrucciones. Al final, se imprimen todas las variables y se ejecutan las instrucciones de impresión acumuladas.
2. **Instrucciones**
    - instrucciones puede ser una serie de declaraciones, impresiones, sentencias if, ciclos while o estar vacía.
    - Cada tipo de instrucción se procesa y se acumula en listas de variables o de impresiones según corresponda.
3. **Declaraciones**
    - Las declaraciones pueden ser simplemente la declaración de una variable con su tipo y nombre, o una declaración con asignación de valor.
    - En el caso de asignaciones, si la variable ya existe, se actualiza su valor; si es nueva, se agrega a la lista de variables.
4. **Impresiones**
    - La producción print maneja las instrucciones de impresión, permitiendo imprimir texto literal, el valor de una variable, o el resultado de una concatenación de textos y variables.

5. **Condiciones if**
    - La sentencia if evalúa una condición y ejecuta una serie de instrucciones si la condición es verdadera, o una serie de instrucciones alternativas si es falsa.
    - Se gestionan las listas de variables e impresiones tanto para la parte verdadera (if) como para la alternativa (else).

6. **Bucles while**
    - La producción ciclo define la estructura de un bucle while, evaluando una condición antes de ejecutar las instrucciones del cuerpo del bucle.
    - Se menciona la estructura del bucle, pero la implementación detallada del cuerpo del bucle está comentada y puede ser adaptada según las necesidades del lenguaje.
7. **Condiciones y Operaciones**
    - **Condiciones**: Estas producen booleanos al comparar valores o variables usando operadores de comparación (>, <, ==, etc.).
    - **Operaciones**: Permiten realizar operaciones aritméticas entre valores o variables, acumulando los resultados en una cadena que representa la expresión aritmética.
8. **Concatenaciones**
    - Las producciones concatenar y concatenarp permiten construir cadenas de texto combinando literales y valores de variables.
9. **Funciones**
    - La producción funcion define la estructura de una función, incluyendo su nombre, parámetros, tipo de retorno, cuerpo de instrucciones y valor de retorno.