Manual tecnico

Practica 1

Compiladores 1

Lexer

ANÁLISIS LÉXICO

Estructura General del Lexer

El analizador léxico (Lexer) constituye la primera fase del compilador y se encarga de transformar el flujo de caracteres de entrada en una secuencia de tokens significativos. Implementado utilizando JFlex, el lexer se organiza en tres secciones fundamentales: declaraciones globales, macros, y reglas léxicas.

Declaraciones Iniciales

```
package com.example.practica1_compi.analizadores;
import java_cup.runtime.Symbol;

%% //separador de area
```

La sección inicial contiene las declaraciones de paquete, importaciones necesarias y las directivas de configuración del analizador:

%public: Define la clase generada como pública

%unicode: Habilita el soporte para caracteres Unicode

%class Lexer: Especifica el nombre de la clase generada

%cup: Indica compatibilidad con el generador de parser CUP

%line y %column: Habilitan el seguimiento de número de línea y columna

Código de Inicialización

```
%init{
    errorList = new ArrayList<>();
%init}
```

Este bloque se ejecuta al instanciar el lexer, inicializando las estructuras necesarias para el manejo de errores.

Macros Léxicas

Las macros son definiciones de patrones regulares que facilitan la reutilización y mantenibilidad del código. Se definen en la sección identificada por el comentario "/********** macros **********/".

Definición de Macros

Macro	Expresión Regular	Descripción
DIGITO	[0-9]	Define un carácter numérico del 0 al 9
LETRA	[a-zA-Z]	Define un carácter alfabético mayúscula o minúscula
ID	{LETRA}({LETRA}|{DIGITO}|_)*	Define identificadores válidos que deben comenzar con letra, seguido de letras, dígitos o guiones bajos

ENTERO {DIGITO}+ Define números enteros compuestos por uno o más dígitos
DECIMAL {DIGITO}+"."{DIGITO}+ Define números decimales con parte entera y decimal
DECIMAL_INCOMPLETO {DIGITO}+ "." Define números decimales incompletos (sin parte decimal)
CADENA "[^"]*" Define cadenas de texto entre comillas dobles
HEXCOLOR "H"[0-9a-fA-F]{6} Define códigos de color hexadecimal con prefijo H seguido de 6 caracteres hexadecimales
ESPACIO [\t\r\n\f]+ Define espacios en blanco, tabulaciones y saltos de línea
ID_INVALIDO_NUM ({DIGITO}+("."{DIGITO}+)?)({LETRA}|)+({LETRA}|{DIGITO}|)* Detecta identificadores inválidos que comienzan con números
ID_INVALIDO_GUION ({LETRA}|{DIGITO}|)+ Detecta identificadores inválidos que comienzan con guión bajo

Código de Usuario

```
%{
   //metodo para el manejo de errores
   private List<String> errorList;
   ...
%}
```

Esta sección contiene métodos y atributos que se incorporarán directamente en la clase generada, proporcionando funcionalidades auxiliares para el análisis léxico.

Gestión de Errores Léxicos

El sistema implementa un mecanismo completo para la detección y reporte de errores léxicos:

errorList: Almacena mensajes descriptivos de errores

errorReportes: Contiene objetos estructurados con información detallada de cada error (lexema, línea, columna, tipo, mensaje)

Método error(String message): Registra un error con la información contextual necesaria

Generación de Tokens

```
private Symbol symbol(int type) {
   return new Symbol(type, yyline + 1, yycolumn + 1, yytext());
}
```

Los métodos symbol sobrecargados permiten crear tokens con información de posición y valor asociado.

Reporte de Operadores

Se mantiene una lista de operadores encontrados durante el análisis para generar reportes específicos que documenten los operadores aritméticos utilizados en el código fuente.

Reglas Léxicas
La sección de reglas, delimitada por "%%", define la correspondencia entre patrones y acciones a ejecutar.

Comentarios

"#".* { /* provisionalmente se ignora */}
Los comentarios iniciados con # son ignorados por el analizador.

Elementos de Configuración
Token Patrón Descripción
SEPARADOR_SECCION "%%%%" Define el separador entre secciones del diagrama
CONFIG Múltiples patrones como "%DEFAULT", "%COLOR_TEXTO_SI", etc. Instrucciones de configuración de diagramas
FIGURA "ELIPSE", "CIRCULO", "PARALELOGRAMO", etc. Tipos de figuras geométricas para diagramas
LETRA"ARIAL", "TIMES_NEW_ROMAN", etc. Tipos de fuentes tipográficas
COLOR_HEX {HEXCOLOR} Códigos de color en formato hexadecimal

Manejo de Identificadores Inválidos
Se implementa una detección temprana de identificadores que no cumplen con las reglas sintácticas:

ID_INVALIDO_NUM: Identificadores que comienzan con dígitos

ID_INVALIDO_GUION: Identificadores que comienzan con guión bajo

Estos patrones generan errores léxicos específicos antes de ser procesados por otras reglas.

Palabras Reservadas del Pseudocódigo
Token Patrón Uso
INICIO "INICIO" Marca el inicio del programa
FIN "FIN" Marca el final del programa
VAR "VAR" Declaración de variables
SI "SI" Estructura condicional
ENTONCES "ENTONCES" Marca el inicio del bloque en una condicional
FINSI "FINSI"Marca el final de una condicional
MIENTRAS "MIENTRAS" Estructura de bucle
HACER "HACER" Marca el inicio del bloque en un bucle
FINMIENTRAS "FINMIENTRAS" Marca el final de un bucle
MOSTRAR "MOSTRAR" Instrucción de salida
LEER "LEER" Instrucción de entrada

Operadores
Operadores Relacionales:

"==" (IGUALDAD)

"!=" (DISTINTO)

"<=" (MENOR_IGUAL)

">=" (MAYOR_IGUAL)

"<" (MENOR)

">" (MAYOR)

Operadores Lógicos:

"&&" (AND)

"||" (OR)

"!" (NOT_LOGICO)

Operadores Aritméticos:

"+" (SUMA)

"-" (RESTA)

"*" (MULT)

"/" (DIV)

Símbolos Especiales

Símbolo	Token	Descripción
"="	OPERADOR_ASIGNACION	Operador de asignación
","	COMA	Separador de elementos
" "	PIPE	Símbolo de tubería
"("	LPAREN	Paréntesis izquierdo
")"	RPAREN	Paréntesis derecho

Literales
Números: Se procesan enteros y decimales, convirtiéndolos a los tipos Java apropiados (Integer o Double)

Cadenas: Se extrae el contenido eliminando las comillas delimitadoras

Identificadores: Se validan según la macro ID definida

Manejo de Espacios y Caracteres Especiales
ESPACIO: Se ignoran espacios en blanco, tabulaciones y saltos de línea

Caracteres no reconocidos: Se registran como errores léxicos

Fin de archivo: Se genera un token especial EOF

Consideraciones de Implementación
El analizador léxico ha sido diseñado considerando:

Orden de las reglas: Las reglas más específicas preceden a las más generales para evitar capturas incorrectas

Detección temprana de errores: Los patrones de identificadores inválidos se evalúan antes que los válidos

Registro contextual: Cada token generado incluye información de línea y columna para facilitar el reporte de errores

Flexibilidad: Las macros permiten modificar fácilmente las definiciones de patrones sin afectar las reglas principales

Parser

ANÁLISIS SINTÁCTICO
Estructura General del Parser
El analizador sintáctico constituye la segunda fase del compilador y se encarga de verificar la estructura gramatical del código fuente, construyendo una representación intermedia en forma de árbol de sintaxis abstracta (AST). Implementado mediante CUP, el parser recibe la secuencia de tokens del lexer y aplica las reglas de producción de la gramática definida.

Declaraciones Iniciales

```
package com.example.practica1_compi.analizadores;
import java_cup.runtime.*;
import java.util.*;
import com.example.practica1_compi.models.*;
```

La sección inicial contiene las declaraciones de paquete y las importaciones necesarias para el funcionamiento del parser, incluyendo las clases del modelo de datos que representan los nodos del AST.

Código Embebido del Parser

```
parser code {:
    private Lexer lex;
    private List<ErrorReporte> errorReportes = new ArrayList<>();
    private List<ReporteEstructura> estructuras = new ArrayList<>();
:}
```

El bloque de código embebido contiene atributos y métodos que se incorporan directamente en la clase Parser generada, proporcionando funcionalidades para el manejo de errores y el acceso a componentes relacionados.

Constructor y Accesores

```
public Parser(Lexer lex) {
    super(lex);
    this.lex = lex;
}

public Lexer getLexer() {
    return this.lex;
}
```

El constructor recibe una instancia del analizador léxico, estableciendo la comunicación entre ambas fases del compilador. Los métodos accesores permiten obtener referencias al lexer asociado.

Gestión de Errores Sintácticos
El parser implementa un sistema completo para la detección y reporte de errores sintácticos:

El método syntax_error se invoca cuando se encuentra un token inesperado durante el análisis. Registra el token problemático con su información contextual y un mensaje descriptivo.

El método report_error proporciona un mecanismo general para reportar errores sintácticos, aceptando un mensaje personalizado y la información del símbolo asociado.

El método unrecovered_syntax_error maneja errores sintácticos fatales de los cuales el parser no puede recuperarse, registrando el error y continuando con el análisis cuando sea posible.

Definición de Terminales y No Terminales
Terminales
Los terminales representan los tokens reconocidos por el analizador léxico que forman los elementos básicos de la gramática:

SEPARADOR_SECCION marca la separación entre secciones del programa.
CONFIG, FIGURA, LETRA y COLOR_HEX son elementos de configuración de diagramas.
INICIO, FIN y VAR son palabras reservadas para estructura del programa.
SI, ENTONCES y FINSI corresponden a estructuras condicionales.
MIENTRAS, HACER y FINMIENTRAS representan estructuras de iteración.
MOSTRAR y LEER son instrucciones de entrada y salida.
OPERADOR_ASIGNACION es el símbolo de asignación igual.
SUMA, RESTA, MULT y DIV son operadores aritméticos.
MENOR, MAYOR, MENOR_IGUAL, MAYOR_IGUAL, IGUALDAD y DISTINTO son operadores relacionales.
AND, OR y NOT_LOGICO son operadores lógicos.
COMA, PIPE, LPAREN y RPAREN son símbolos especiales.
CADENA e IDENTIFICADOR son literales de cadena e identificadores.
NUMERO representa literales numéricos.
ERROR, ERROR_IDENTIFICADOR_INVALIDO y ERROR_DECIMAL_INVALIDO son tokens de error.

No Terminales
Los no terminales representan construcciones gramaticales que se derivan en secuencias de terminales y otros no terminales:

programa representa la unidad completa de compilación. seccion_pseudocodigo contiene las instrucciones del programa principal. lista_instrucciones es una secuencia de instrucciones del pseudocódigo. instruccion es una abstracción de cualquier instrucción válida. declaracion es una instrucción de declaración de variables. asignacion es una instrucción de asignación de valores. mostrar es una instrucción de salida. leer es una instrucción de entrada. si_condicional es una estructura condicional completa. mientras_ciclo es una estructura de iteración. bloque_simple es un conjunto de

instrucciones simples. expresion representa expresiones aritméticas. condicion representa expresiones lógicas y relacionales. seccion_configuracion contiene las configuraciones del diagrama. configuracion es un elemento individual de configuración. COLOR_RGB es la representación de color en formato RGB.

Declaraciones de Precedencia

precedence left OR;
precedence left AND;
precedence right NOT_LOGICO;
precedence left MENOR, MAYOR, MENOR_IGUAL, MAYOR_IGUAL, IGUALDAD, DISTINTO;
precedence left SUMA, RESTA;
precedence left MULT, DIV;
Las declaraciones de precedencia establecen el orden de evaluación de los operadores, resolviendo ambigüedades gramaticales. Los operadores lógicos OR y AND tienen asociatividad izquierda, con OR de menor precedencia que AND. El operador NOT tiene asociatividad derecha y mayor precedencia que AND y OR. Los operadores relacionales tienen precedencia intermedia. Los operadores aritméticos siguen la precedencia convencional, donde multiplicación y división tienen mayor precedencia que suma y resta.

Reglas de Producción
Estructura del Programa
La producción principal define la estructura general del programa, compuesta por una sección de pseudocódigo seguida de un separador y una sección de configuraciones. El resultado es un nodo NodoPrograma que encapsula ambas secciones.

Sección de Pseudocódigo
La sección de pseudocódigo está delimitada por las palabras reservadas INICIO y FIN, conteniendo una lista de instrucciones en su interior. Esta estructura representa el cuerpo principal del programa.

Lista de Instrucciones
Las listas de instrucciones se construyen de manera recursiva, permitiendo secuencias de cualquier longitud. Una lista puede ser una instrucción individual o una lista existente seguida de una nueva instrucción. Este enfoque recursivo facilita el procesamiento secuencial de las instrucciones.

Instrucciones
El no terminal instrucción agrupa todos los tipos posibles de instrucciones que pueden aparecer en el programa. Incluye declaraciones, asignaciones, instrucciones de entrada y salida, así como estructuras de control condicionales y de iteración. Esta abstracción permite tratar uniformemente cualquier instrucción válida.

Declaración de Variables
Las declaraciones pueden presentarse de dos formas: con inicialización o sin ella. En el primer caso, se utiliza la palabra reservada VAR seguida de un identificador, el operador de

asignación y una expresión. En el segundo caso, solo se declara el identificador sin valor inicial, creando una variable sin inicializar.

Asignación
La asignación permite modificar el valor de una variable previamente declarada. Consiste en un identificador seguido del operador de asignación y una expresión que proporciona el nuevo valor. Esta construcción es fundamental para la manipulación de datos durante la ejecución.

Instrucción Mostrar
La instrucción MOSTRAR tiene dos variantes. Puede mostrar una cadena literal, donde el mensaje se especifica directamente entre comillas. También puede mostrar el resultado de evaluar una expresión, permitiendo visualizar valores calculados o contenidos de variables.

Instrucción Leer
La instrucción LEER permite la entrada de datos desde el usuario. Asocia el valor ingresado con un identificador de variable, almacenando la información para su uso posterior en el programa.

Estructura Condicional SI
La estructura condicional evalúa una condición y ejecuta un bloque de instrucciones si esta se cumple. La gramática acepta dos formatos de cierre, permitiendo tanto FINSI como FIN SI como delimitadores del bloque condicional. Esta flexibilidad facilita la escritura del código fuente.

Estructura de Iteración MIENTRAS
El ciclo MIENTRAS ejecuta repetidamente un bloque de instrucciones mientras se cumpla una condición especificada. Similar a la estructura condicional, acepta dos formatos de cierre: FINMIENTRAS y FIN MIENTRAS, proporcionando alternativas para el programador.

Sección de Configuración
La sección de configuraciones contiene una lista de elementos de configuración que definen aspectos visuales del diagrama. Cada configuración asocia una directiva con un valor, pudiendo incluir opcionalmente un modificador numérico separado por el símbolo PIPE.

Valores de Configuración
Los valores de configuración pueden ser de diversos tipos: colores en formato hexadecimal, colores RGB, tipos de figura, tipos de letra o valores numéricos. Esta variedad permite personalizar diferentes aspectos del diagrama según las necesidades.

Expresiones Aritméticas
Las expresiones aritméticas se construyen jerárquicamente respetando la precedencia de operadores. Una expresión puede ser una suma o resta de términos, o un término simple. Los términos, a su vez, pueden ser multiplicaciones o divisiones de factores, o factores simples. Los factores incluyen números literales, identificadores de variables y expresiones entre paréntesis.

Condiciones

Las condiciones combinan expresiones mediante operadores relacionales y lógicos. Pueden comparar dos expresiones con operadores como menor, mayor, igualdad o distinto. También pueden combinar condiciones mediante operadores lógicos AND y OR, o negar una condición con NOT. Los paréntesis permiten agrupar condiciones para modificar la precedencia de evaluación.

Color RGB

La producción para COLOR_RGB permite definir colores mediante componentes rojo, verde y azul. Cada componente es una expresión, proporcionando flexibilidad para especificar valores mediante cálculos o constantes. El resultado es una cadena que concatena los valores de los tres componentes separados por comas.

Consideraciones de Implementación

El parser ha sido diseñado considerando varios aspectos importantes para garantizar su correcto funcionamiento. La gramática es no ambigua gracias a las declaraciones de precedencia que resuelven conflictos potenciales. El manejo de errores está integrado en todas las etapas del análisis, proporcionando información detallada para la depuración. La construcción del AST se realiza de manera incremental, creando nodos especializados para cada tipo de construcción gramatical. La separación entre la sección de pseudocódigo y la sección de configuraciones permite un procesamiento independiente de ambas partes del programa.

Analizador

CLASE ANALIZADOR
Descripción General
La clase Analizador constituye el componente central de integración del compilador, actuando como fachada que coordina las fases de análisis léxico y sintáctico. Proporciona un punto de entrada unificado para el procesamiento de código fuente, encapsulando la complejidad de la interacción entre los diferentes componentes del sistema y gestionando la recopilación de resultados y errores.

Ubicación y Dependencias
package com.example.practica1_compi.logic

import android.util.Log
import com.example.practica1_compi.analizadores.Lexer
import com.example.practica1_compi.analizadores.Parser
import com.example.practica1_compi.models.ErrorReporte
import com.example.practica1_compi.models.NodoPrograma
import com.example.practica1_compi.models.ResultadoAnalisis
import java.io.StringReader
La clase reside en el paquete logic, separando la lógica de coordinación de los componentes de análisis. Las importaciones incluyen los analizadores léxico y sintáctico, los modelos de datos para errores y resultados, así como las utilidades de Java para manejo de flujos de entrada.

Estructura de la Clase
Definición

```
class Analizador {
    // métodos y lógica interna
}
```
La clase se define sin constructor explícito, utilizando el constructor por defecto proporcionado por Kotlin. No mantiene estado interno entre invocaciones, lo que la hace segura para su uso concurrente y facilita su reutilización.

Método Principal
Firma del Método

fun analyze(texto: String): ResultadoAnalisis
El método analyze constituye el punto de entrada principal para el procesamiento de código fuente. Recibe una cadena de texto que contiene el código a analizar y retorna un objeto ResultadoAnalisis que encapsula toda la información generada durante el proceso.

Estructura General del Procesamiento
El método implementa un flujo de trabajo estructurado en etapas secuenciales, con manejo de excepciones que garantiza que siempre se retorne un resultado válido, incluso en condiciones de error.

Inicialización de Componentes

```
val lexer = Lexer(StringReader(texto))
val parser = Parser(lexer)
```

El proceso comienza con la creación del analizador léxico, alimentado con un StringReader que convierte el texto de entrada en un flujo de caracteres procesable. El analizador sintáctico se instancia recibiendo el lexer como parámetro, estableciendo así la conexión entre ambas fases del compilador.

Verificación de Errores Léxicos

```
if (lexer.errorReportes.isNotEmpty()) {
    lexer.errorReportes.forEach { error ->
        // procesamiento silencioso de errores
    }
}
```

Antes de proceder con el análisis sintáctico, se verifica si el lexer ha detectado errores durante su fase de inicialización. Aunque el bloque de código no implementa acciones explícitas sobre cada error, la verificación permite tomar decisiones posteriores basadas en la presencia de errores léxicos.

Ejecución del Análisis Sintáctico

```
val resultadoParse = parser.parse()
```

Se invoca el método parse del parser, que desencadena el análisis sintáctico completo. Este método retorna un objeto Symbol que encapsula el resultado del análisis, incluyendo el árbol de sintaxis abstracta generado.

Verificación de Errores Sintácticos

```
if (parser.errorReportes.isNotEmpty()) {
    parser.errorReportes.forEach { error ->
        // procesamiento silencioso de errores
    }
}
```

Similar a la fase léxica, se verifica la presencia de errores sintácticos registrados durante el análisis. Esta información será crucial para determinar la validez del programa procesado.

Validación del Resultado del Parser

```
if (resultadoParse == null) {
    val errores = mutableListOf<ErrorReporte>()
```

```
    errores.addAll(lexer.errorReportes)
    errores.addAll(parser.errorReportes)

    if (errores.isEmpty()) {
        errores.add(
            ErrorReporte(
                lexema = "SISTEMA",
                linea = 0,
                columna = 0,
                tipo = "sistema",
                descripcion = "El parser retornó null sin errores registrados"
            )
        )
    }

    return ResultadoAnalisis(errores = errores)
}
```

Esta sección maneja el caso crítico donde el parser retorna null, situación que debería ir acompañada de errores registrados. Si no existen errores, se genera un error de sistema para mantener la consistencia del resultado.

Extracción del Programa

```
val programa = when (val value = resultadoParse.value) {
    is NodoPrograma -> {
        value
    }
    else -> {
        null
    }
}
```

El valor contenido en el resultado del parser debe ser un NodoPrograma, la raíz del árbol de sintaxis abstracta. Mediante una expresión when con type checking, se verifica el tipo del objeto y se extrae el programa solo si corresponde al tipo esperado.

Consolidación de Errores

```
val erroresTotales = mutableListOf<ErrorReporte>()
erroresTotales.addAll(lexer.errorReportes)
erroresTotales.addAll(parser.errorReportes)
```

Se consolidan todos los errores detectados durante ambas fases del análisis en una lista única, facilitando su procesamiento posterior y su inclusión en el resultado final.

Validación Final

```
if (erroresTotales.isNotEmpty() || programa == null) {
    return ResultadoAnalisis(errores = erroresTotales)
}
```

Si existen errores o no se pudo obtener un programa válido, se retorna un resultado parcial que incluye únicamente los errores recopilados, indicando que el análisis no pudo completarse exitosamente.

Procesamiento de Resultados Exitosos

```
val operadoresTotales = lexer.operadores
val extractor = ExtractorReporteEstructuras()
val estructurasTotales = extractor.extraer(programa)
```

En caso de análisis exitoso, se procede a extraer información complementaria. Se obtienen los operadores identificados por el lexer y se invoca al extractor de estructuras para analizar el programa y generar reportes detallados de las construcciones sintácticas encontradas.

Enriquecimiento de Operadores

```
val lineas = texto.lines()
val operadoresConOcurrencia = operadoresTotales.map { operadorActual ->
    val ocurrenciaDetallada = ConstructorOcurrencia.construirOcurrencia(
        lineas,
        operadorActual.linea,
        operadorActual.columna
    )
    operadorActual.copy(ocurrencia = ocurrenciaDetallada)
}
```

Para cada operador detectado, se construye una representación enriquecida que incluye el contexto de su ocurrencia en el código fuente. Se divide el texto original en líneas y se utiliza el ConstructorOcurrencia para extraer el fragmento de código donde aparece cada operador.

Construcción del Resultado Final

```
return ResultadoAnalisis(
    errores = erroresTotales,
    operadores = operadoresConOcurrencia,
    estructuras = estructurasTotales,
    programa = programa
)
```

Se construye y retorna el objeto ResultadoAnalisis que encapsula toda la información generada: errores consolidados, operadores enriquecidos con contexto, estructuras extraídas del programa y el árbol de sintaxis abstracta completo.

Manejo de Excepciones

```
catch (e: Exception) {
    return ResultadoAnalisis(
        errores = listOf(
            ErrorReporte(
                lexema = "",
```

```
                linea = 0,
                columna = 0,
                tipo = "sistema",
                descripcion = "Excepcion: ${e.message}\n${e.stackTraceToString()}"
            )
        )
    )
}
```

El bloque catch captura cualquier excepción no manejada durante el proceso de análisis. Convierte la excepción en un error de sistema con formato estándar, incluyendo el mensaje y la traza de pila para facilitar la depuración. Esto garantiza que el método siempre retorne un objeto ResultadoAnalisis válido, manteniendo la robustez del sistema.

Flujo de Trabajo Completo

El método analyze implementa un flujo de trabajo secuencial y robusto que puede resumirse en las siguientes etapas:

Inicialización de componentes léxicos y sintácticos. Ejecución del análisis sintáctico completo. Verificación de la validez del resultado obtenido. Consolidación de errores de todas las fases. Extracción de información complementaria en caso de éxito. Enriquecimiento de datos con contexto del código fuente. Retorno de un resultado estructurado que encapsula toda la información generada.

Consideraciones de Diseño

La clase Analizador ha sido diseñada siguiendo principios de responsabilidad única y separación . Actúa exclusivamente como coordinador, delegando las tareas especializadas a los componentes correspondientes. El manejo de errores es exhaustivo y garantiza que nunca se retorne un estado inconsistente. La inmutabilidad de los datos de entrada y la ausencia de estado interno facilitan su uso en contextos concurrentes. La estructura del resultado final proporciona toda la información necesaria para las etapas posteriores del procesamiento o para la presentación al usuario.