Grupo 10

Informe de Implementación: Analizador Léxico y Sintáctico

Integrantes:

de Cáseres, Gonzalo gonzadecaseres@gmail.com
Halty, Hector hectormanuelhalty@gmail.com
Lasarte, Fermin. fermin.lasarte@icloud.com

Temas particulares asignados:

- **2:** Enteros sin signo (16 bits) con sufijo `UI`.
- **5:** Punto Flotante de 32 bits con exponente `F`.
- **8:** Cadenas multilíneas delimitadas por `&`.
- **9, 14, 28:** Palabras reservadas `var`, `while`, `do`, `lambda`, `cr`, `se`, `le`, `toui`.
- **19, 21, 23:** Asignación múltiple restringida, funciones con retorno múltiple y prefijado opcional.
- **25:** Semántica de pasaje de parámetros por copia-resultado.
- **31:** Conversión explícita a `uint`.
- **32:** Comentarios de una línea iniciados con `@`.

Tutor asignado: José Fernández León

Introducción

La implementación de un compilador es una tarea compleja que involucra varias etapas. En este informe, nos enfocamos en las dos primeras fases del proceso: el análisis léxico y el análisis sintáctico.

El analizador léxico se encarga de leer el flujo de entrada del programa fuente y dividirlo en unidades significativas llamadas tokens. Esta etapa es esencial para simplificar la estructura del programa, eliminando detalles irrelevantes y facilitando la posterior identificación de la gramática del lenguaje.

Una vez que el analizador léxico ha procesado la entrada, el analizador sintáctico (o parser) toma los tokens generados y verifica que su disposición siga las reglas gramaticales del lenguaje. Este paso garantiza que el código fuente esté estructurado correctamente, de acuerdo con la gramática formal predefinida.

En este contexto, el informe detalla las decisiones de diseño e implementación tomadas para desarrollar estas dos fases, las cuales son fundamentales para que el compilador intérprete y procese correctamente el código fuente.

Trabajo Práctico 1: Análisis Léxico

El analizador léxico se implementó en la clase AnalizadorLexico.java, utilizando una máquina de estados finitos para procesar el código fuente. Este componente descompone el archivo en tokens, los clasifica y gestiona la tabla de símbolos. Las acciones específicas que se ejecutan durante las transiciones de estado están encapsuladas en la clase AccionSemantica.java.

Estructura y Componentes Clave

1. **Máquina de Estados Finitos:** El núcleo del analizador es una matriz de transición de estados (matrizTransicionEstados) que dicta el comportamiento del autómata. Las filas representan los estados y las columnas, los tipos de caracteres de entrada.

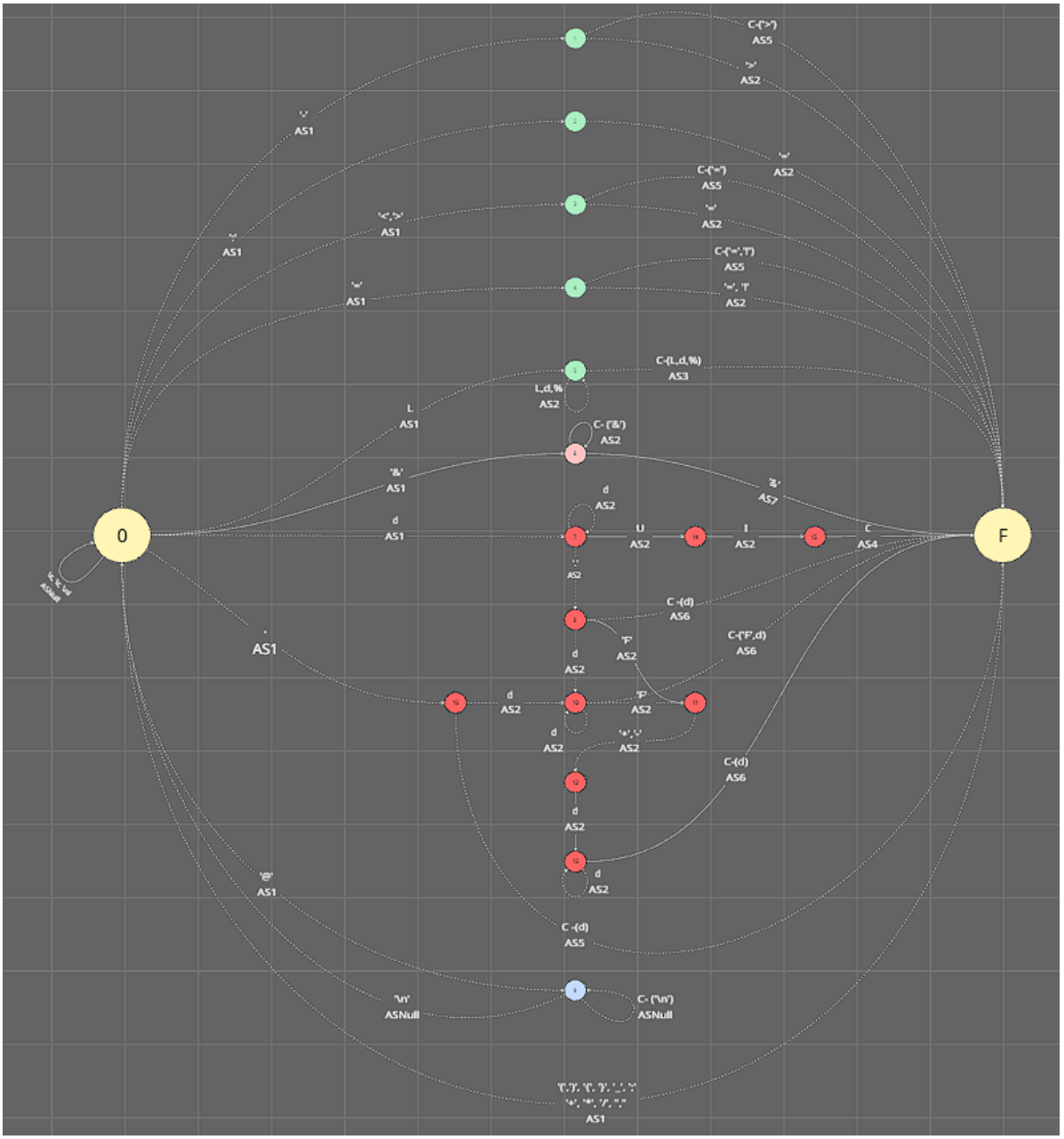
2. **Mapeo de Caracteres:** Para simplificar la matriz, utilizamos una estructura HashMap<Character, Integer> (columnaMatrices) agrupa los caracteres de entrada en categorías (letras, dígitos, operadores, etc.).

3. **Acciones Semánticas**: Una matriz (matrizAccionesSemanticas) que se alinea con la de transiciones y especifica qué acción ejecutar en cada paso.

4. **Tabla de Símbolos:** Implementada con una estructura dinámica HashMap, almacena cada lexema junto con sus atributos ("Uso", "Tipo", "Reservada"). Se precarga con las palabras reservadas del lenguaje.

Diagrama de Transición de Estados

El comportamiento del analizador léxico se modela mediante un autómata finito determinista. El diagrama representa cómo el analizador transita entre estados en función de los caracteres leídos.



LINK DEL AUTOMATA

Matrices de Transición y Acciones Semánticas

El mecanismo implementado utiliza dos matrices: una para la transición de estados (matrizTransicionEstados) y otra para las acciones semánticas (matrizAccionesSemanticas). Ambas son el núcleo del AnalizadorLexico y funcionan coordinadamente para procesar el código fuente. A continuación, vamos a describir la estructura, función, índices y donde sea necesario, mencionar los valores especiales que cargan las matrices, para una correcta implementación en código.

Implementación de las Matrices

1. **Matriz de Transición de Estados (matrizTransicionEstados):**
    - **Estructura:** Es una matriz de tipo int.
    - **Función:** Cada celda almacena el número del próximo estado al que el autómata debe transitar.
    - **Índices:** Las filas representan el estado actual del autómata, mientras que las columnas corresponden a un tipo de carácter de entrada.
    - **Valores especiales:**
        - -1 indica un estado final, lo que significa que se ha reconocido un token completo.
        - -2 representa un estado de error, que se activa cuando llega un carácter inesperado para el estado actual.

2. **Matriz de Acciones Semánticas (matrizAccionesSemanticas):**
    - **Estructura:** Es una matriz de objetos de tipo AccionSemantica.
    - **Función:** Cada celda contiene una instancia de un objeto que representa la acción semántica a ejecutar durante una transición específica. Todas estas clases de acciones se heredan de una clase abstracta AccionSemantica.
    - **Índices:** Está alineada directamente con la matriz de transición de estados. La misma combinación de [estado_actual][caracter_entrada] que determina el próximo estado, también selecciona la acción a ejecutar.

3. **Mapeo de Caracteres (columnaMatrices):**
    - Para simplificar las matrices y evitar tener una columna para cada carácter, se utiliza un HashMap<Character, Integer>.
    - Esta estructura agrupa los caracteres en categorías (letras, dígitos, operadores, etc.) y les asigna un índice de columna numérica. Por ejemplo, a todas las letras se les puede asignar la misma columna.

Mecanismo de Funcionamiento en yylex()

Todo el proceso mencionado anteriormente se lleva a cabo dentro del método yylex(), que se ejecuta en un ciclo para consumir el archivo fuente carácter por carácter:

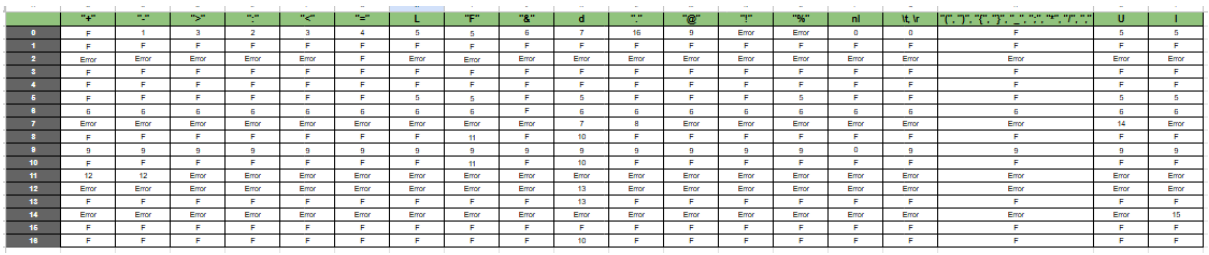
1. **Lectura del Carácter:** En cada iteración, se lee el siguiente carácter del archivo fuente.

2. **Determinación de la Columna:** Se consulta el HashMap columnaMatrices para obtener el índice de columna correspondiente al carácter leído.

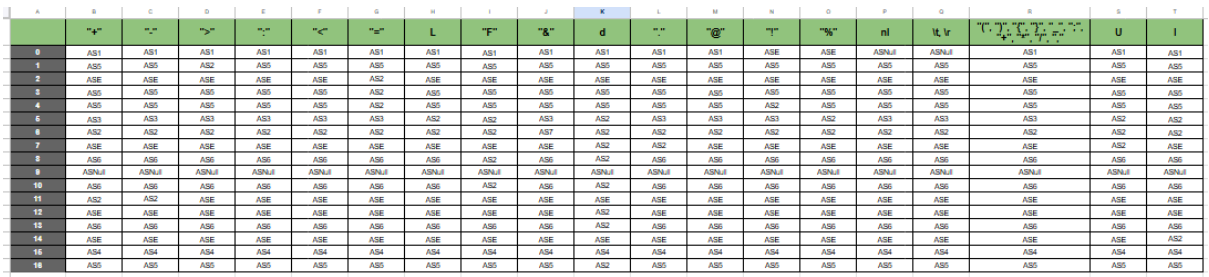
3. **Ejecución de la Acción Semántica:** Se utiliza el estado actual (fila) y el índice del carácter (columna) para acceder a la matrizAccionesSemanticas. Se invoca el método aplicarAS() del objeto de acción semántica encontrado en esa celda.

4. **Transición de Estado:** Usando los mismos índices, se consulta la matrizTransicionEstados para obtener el número del próximo estado. Este número se convierte en el estado actual para la siguiente iteración del ciclo.

5. **Fin del Token:** El ciclo continúa hasta que se alcanza un estado final (-1) o un estado de error (-2). En ese momento, la acción semántica ejecutada en la última transición válida habrá determinado si se reconoció un token, si se debe reportar un error o si la secuencia debe ser ignorada (como en el caso de comentarios o espacios).



	"+"	"-"	">"	":"	"<"	"="	L	"F"	"&"	d	"."	"@"	"!"	"%"	nl	\t, \r	"(", ")", "{", "}", ",", ";", "_", ":", "+", "-", "*", "/", "."	U	I
0	F	1	3	2	3	4	5	5	6	7	16	9	Error	Error	0	0	F	5	5
1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
2	Error	Error	Error	Error	Error	F	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error
3	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
4	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
5	F	F	F	F	F	F	5	5	F	5	F	F	F	5	F	F	F	5	5
6	6	6	6	6	6	6	6	6	F	6	6	6	6	6	6	6	6	6	6
7	Error	Error	Error	Error	Error	Error	Error	Error	Error	7	8	Error	Error	Error	Error	Error	Error	14	Error
8	F	F	F	F	F	F	F	11	F	10	F	F	F	F	F	F	F	F	F
9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	0	9	9	9	9
10	F	F	F	F	F	F	F	11	F	10	F	F	F	F	F	F	F	F	F
11	12	12	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error
12	Error	Error	Error	Error	Error	Error	Error	Error	Error	13	Error	Error	Error	Error	Error	Error	Error	Error	Error
13	F	F	F	F	F	F	F	F	F	13	F	F	F	F	F	F	F	F	F
14	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	15
15	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
16	F	F	F	F	F	F	F	F	F	10	F	F	F	F	F	F	F	F	F



	"+"	"-"	">"	":"	"<"	"="	L	"F"	"&"	d	"."	"@"	"!"	"%"	nl	\t, \r	"(", ")", "{", "}", ",", ";", "_", ":", "+", "-", "*", "/", "."	U	I
0	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	ASE	ASE	ASNull	ASNull	AS1	AS1	AS1
1	AS5	AS5	AS2	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5
2	ASE	ASE	ASE	ASE	ASE	AS2	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE
3	AS5	AS5	AS5	AS5	AS5	AS2	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5
4	AS5	AS5	AS5	AS5	AS5	AS2	AS5	AS5	AS5	AS5	AS5	AS5	AS2	AS5	AS5	AS5	AS5	AS5	AS5
5	AS3	AS3	AS3	AS3	AS3	AS3	AS2	AS2	AS3	AS2	AS3	AS3	AS3	AS2	AS3	AS3	AS3	AS2	AS2
6	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS7	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2
7	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	AS2	AS2	ASE	ASE	ASE	ASE	ASE	ASE	AS2	ASE
8	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS2	AS6	AS2	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6
9	ASNull	ASNull	ASNull	ASNull	ASNull	ASNull	ASNull	ASNull	ASNull	ASNull	ASNull	ASNull	ASNull	ASNull	ASNull	ASNull	ASNull	ASNull	ASNull
10	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS2	AS6	AS2	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6
11	AS2	AS2	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE
12	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	AS2	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE
13	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS2	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6
14	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	ASE	AS2
15	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4
16	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS2	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5

LINK A LAS MATRICES

Acciones Semánticas Implementadas

Las acciones semánticas definen la lógica del analizador. A continuación se describen las más importantes:

- **AS1:** Inicia un nuevo lexema.

```java
public static class AccionSemantica1 extends AccionSemantica{
    // inciar lexema y agregar caracter
    public String aplicarAS(AnalizadorLexico al, char c) {
        al.inicializarLexema();
        al.agregarCaracterLexema(c);
        return null;
    }
}
```

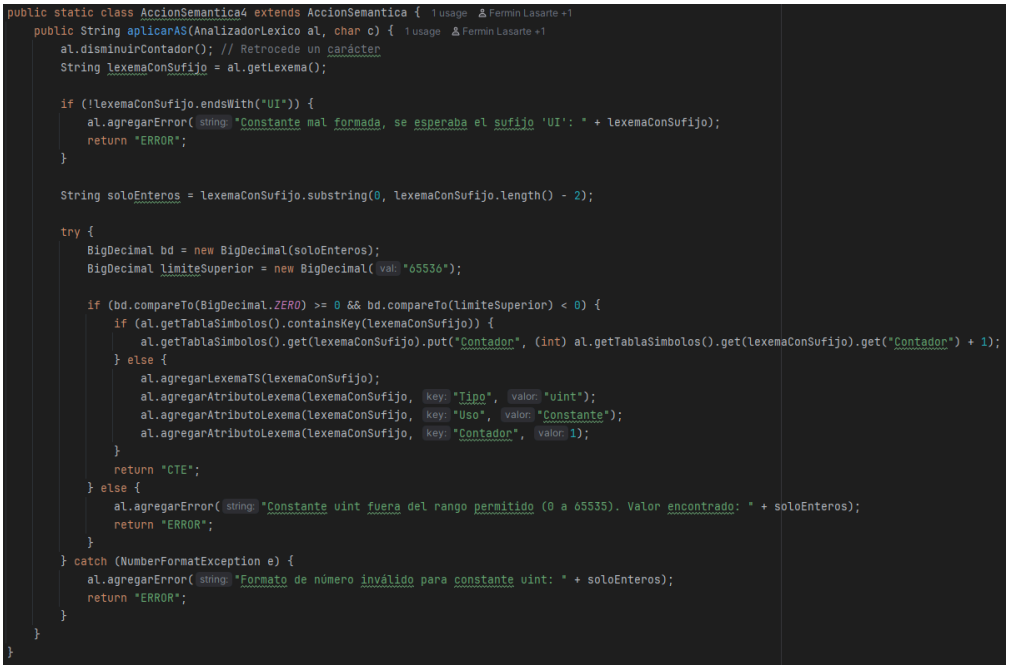
- **AS2:** Agrega el carácter actual al lexema en construcción.

```java
public static class AccionSemantica2 extends AccionSemantica{
    // agregar caracter al lexema
    public String aplicarAS(AnalizadorLexico al, char c) {
        al.agregarCaracterLexema(c);
        return null;
    }
}
```

- **AS3:** Procesa identificadores y palabras reservadas. Chequea que los identificadores comiencen con mayúscula y contengan solo letras mayúsculas, dígitos o %. Trunca los identificadores de más de 20 caracteres y emite un warning.

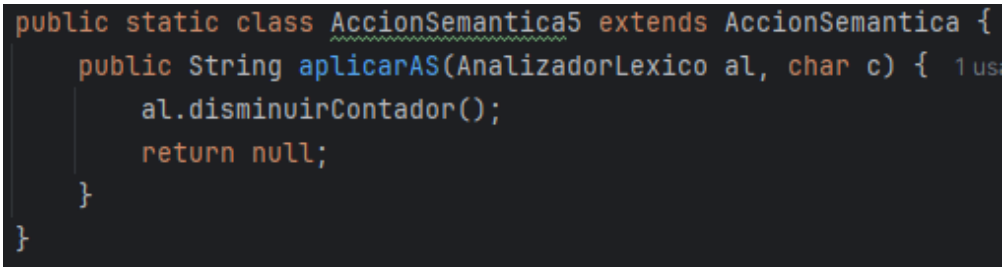
```java
public static class AccionSemantica3 extends AccionSemantica {
    @Override
    public String aplicarAS(AnalizadorLexico al, char c) {
        al.disminuirContador(); // Retrocede para no procesar el caracter que causó el final del token
        String lexemaActual = al.getLexema();

        // las palabras reservadas deben estar en minusculas
        if (al.esPalabraReservada(lexemaActual)) {
            // si es reservada, retornamos el propio lexema en mayúsculas.
            return lexemaActual.toUpperCase();
        }

        char primerChar = lexemaActual.charAt(0);
        if (!Character.isLetter(primerChar) || !Character.isUpperCase(primerChar)) {
            al.agregarError( string: "Identificador '" + lexemaActual + "' debe comenzar con una letra mayuscula.");
            return "ERROR";
        }

        for (int i = 1; i < lexemaActual.length(); i++) {
            char ch = lexemaActual.charAt(i);
            if (!(Character.isUpperCase(ch) && Character.isLetter(ch)) && !Character.isDigit(ch) && ch != '%') {
                al.agregarError( string: "Identificador '" + lexemaActual + "' contiene un caracter invalido ('" + ch + "'). Solo se permiten letras mayusculas, digitos y '%'.");
                return "ERROR";
            }
        }

        if (lexemaActual.length() > 20) {
            String original = lexemaActual;
            lexemaActual = lexemaActual.substring(0, 20);
            al.setLexema(lexemaActual);
            al.agregarWarning( string: "El identificador '" + original + "' fue truncado a 20 caracteres: '" + lexemaActual + "'.");
        }

        if (!al.getTablaSimbolos().containsKey(lexemaActual)) {
            al.agregarLexemaTS(lexemaActual);
            al.agregarAtributoLexema(lexemaActual, key: "Uso", valor: "Identificador");
        }

        return "ID";
    }
}
```

- **AS4:** Chequea y procesa constantes uint, verificando el sufijo UI y el rango [0, 65535].



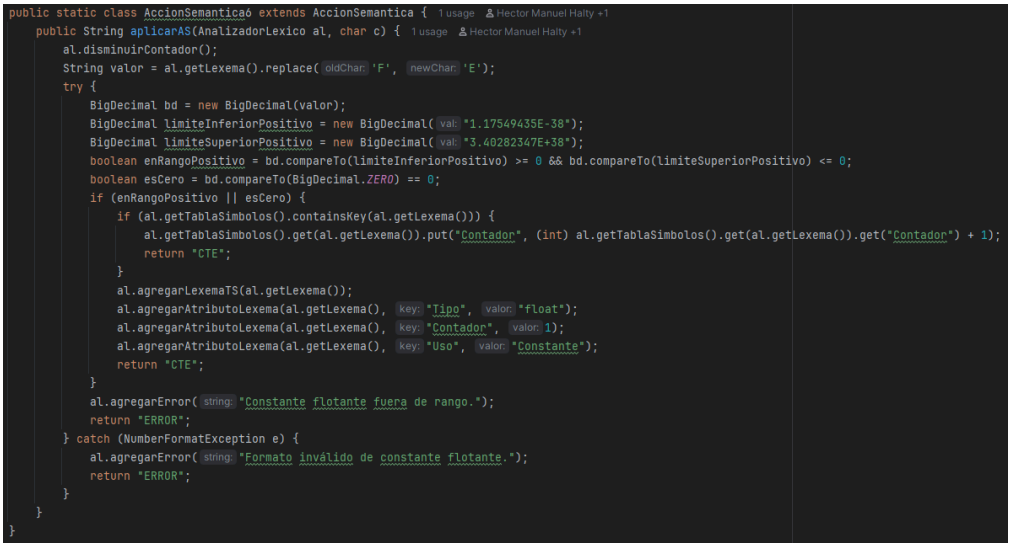
```
public static class AccionSemantica4 extends AccionSemantica {  1 usage   Fermin Lasarte +1
    public String aplicarAS(AnalizadorLexico al, char c) {  1 usage   Fermin Lasarte +1
        al.disminuirContador(); // Retrocede un carácter
        String lexemaConSufijo = al.getLexema();

        if (!lexemaConSufijo.endsWith("UI")) {
            al.agregarError( string: "Constante mal formada, se esperaba el sufijo 'UI': " + lexemaConSufijo);
            return "ERROR";
        }

        String soloEnteros = lexemaConSufijo.substring(0, lexemaConSufijo.length() - 2);

        try {
            BigDecimal bd = new BigDecimal(soloEnteros);
            BigDecimal limiteSuperior = new BigDecimal( val: "65536");

            if (bd.compareTo(BigDecimal.ZERO) >= 0 && bd.compareTo(limiteSuperior) < 0) {
                if (al.getTablaSimbolos().containsKey(lexemaConSufijo)) {
                    al.getTablaSimbolos().get(lexemaConSufijo).put("Contador", (int) al.getTablaSimbolos().get(lexemaConSufijo).get("Contador") + 1);
                } else {
                    al.agregarLexemaTS(lexemaConSufijo);
                    al.agregarAtributoLexema(lexemaConSufijo,  key: "Tipo",  valor: "uint");
                    al.agregarAtributoLexema(lexemaConSufijo,  key: "Uso",  valor: "Constante");
                    al.agregarAtributoLexema(lexemaConSufijo,  key: "Contador",  valor: 1);
                }
                return "CTE";
            } else {
                al.agregarError( string: "Constante uint fuera del rango permitido (0 a 65535). Valor encontrado: " + soloEnteros);
                return "ERROR";
            }
        } catch (NumberFormatException e) {
            al.agregarError( string: "Formato de número inválido para constante uint: " + soloEnteros);
            return "ERROR";
        }
    }
}
```

- **AS5:** Se utiliza en el Analizador Léxico para retroceder un carácter en el flujo de entrada. Esto permite que el último carácter leído (c), que no forma parte del token actual, sea reprocesado en la siguiente transición de la Máquina de Estados.



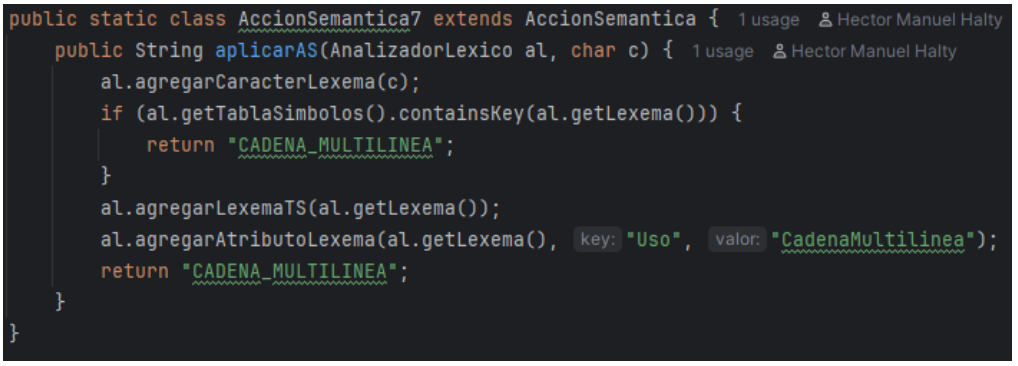
```
public static class AccionSemantica5 extends AccionSemantica {
    public String aplicarAS(AnalizadorLexico al, char c) {  1 us
        al.disminuirContador();
        return null;
    }
}
```

- **AS6:** Valida y procesa constantes float, manejando la notación científica con F y el rango de 32 bits.



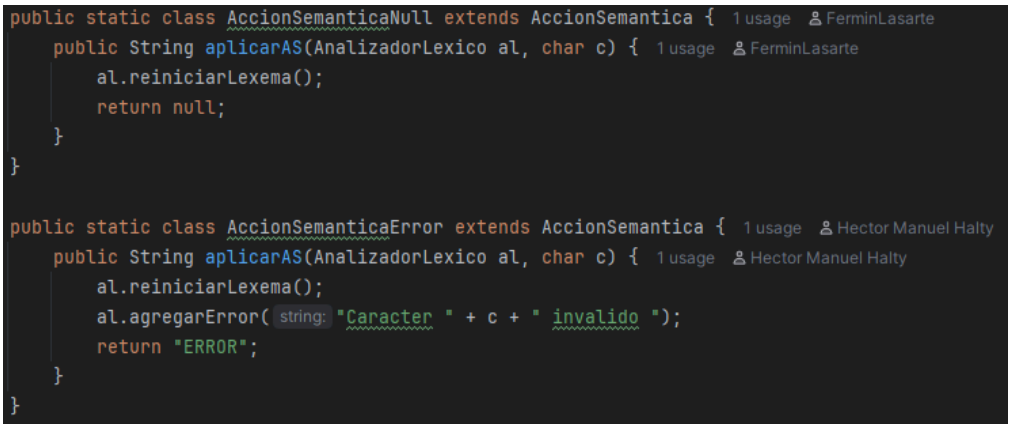
```
public static class AccionSemanticaó extends AccionSemantica {  1 usage  Hector Manuel Halty +1
    public String aplicarAS(AnalizadorLexico al, char c) {  1 usage  Hector Manuel Halty +1
        al.disminuirContador();
        String valor = al.getLexema().replace( oldChar: 'F',  newChar: 'E');
        try {
            BigDecimal bd = new BigDecimal(valor);
            BigDecimal limiteInferiorPositivo = new BigDecimal( val: "1.17549435E-38");
            BigDecimal limiteSuperiorPositivo = new BigDecimal( val: "3.40282347E+38");
            boolean enRangoPositivo = bd.compareTo(limiteInferiorPositivo) >= 0 && bd.compareTo(limiteSuperiorPositivo) <= 0;
            boolean esCero = bd.compareTo(BigDecimal.ZERO) == 0;
            if (enRangoPositivo || esCero) {
                if (al.getTablaSimbolos().containsKey(al.getLexema())) {
                    al.getTablaSimbolos().get(al.getLexema()).put("Contador", (int) al.getTablaSimbolos().get(al.getLexema()).get("Contador") + 1);
                    return "CTE";
                }
                al.agregarLexemaTS(al.getLexema());
                al.agregarAtributoLexema(al.getLexema(),  key: "Tipo",  valor: "float");
                al.agregarAtributoLexema(al.getLexema(),  key: "Contador",  valor: 1);
                al.agregarAtributoLexema(al.getLexema(),  key: "Uso",  valor: "Constante");
                return "CTE";
            }
            al.agregarError( string: "Constante flotante fuera de rango.");
            return "ERROR";
        } catch (NumberFormatException e) {
            al.agregarError( string: "Formato inválido de constante flotante.");
            return "ERROR";
        }
    }
}
```

- **AS7:** Procesa cadenas multilínea delimitadas por &.



```
public static class AccionSemantica7 extends AccionSemantica {  1 usage  Hector Manuel Halty
    public String aplicarAS(AnalizadorLexico al, char c) {  1 usage  Hector Manuel Halty
        al.agregarCaracterLexema(c);
        if (al.getTablaSimbolos().containsKey(al.getLexema())) {
            return "CADENA_MULTILINEA";
        }
        al.agregarLexemaTS(al.getLexema());
        al.agregarAtributoLexema(al.getLexema(),  key: "Uso",  valor: "CadenaMultilinea");
        return "CADENA_MULTILINEA";
    }
}
```

- **ASNull y ASError:** ASNull ignora secuencias como espacios y comentarios, mientras que ASError reporta caracteres inválidos.



```
public static class AccionSemanticaNull extends AccionSemantica {  1 usage  FerminLasarte
    public String aplicarAS(AnalizadorLexico al, char c) {  1 usage  FerminLasarte
        al.reiniciarLexema();
        return null;
    }
}

public static class AccionSemanticaError extends AccionSemantica {  1 usage  Hector Manuel Halty
    public String aplicarAS(AnalizadorLexico al, char c) {  1 usage  Hector Manuel Halty
        al.reiniciarLexema();
        al.agregarError( string: "Caracter " + c + " invalido ");
        return "ERROR";
    }
}
```

Errores Léxicos Considerados y su Tratamiento

El analizador léxico permite que la compilación continúe después de encontrar un error. El sistema registra cada error en una lista interna con su número de línea y columna, para informarlos todos juntos al final del proceso.

Errores y Advertencias en Identificadores

Estos problemas se gestionan en la AccionSemantica3, que se activa al terminar de leer un posible identificador.

- **Identificador Truncado (Warning)**
  - **Causa:** Ocurre cuando un identificador supera la longitud máxima de 20 caracteres.
  - **Tratamiento:** El analizador no lo considera un error fatal. Simplemente, trunca el lexema a 20 caracteres, agrega una advertencia (warning) para notificar al usuario, y continúa el análisis sintáctico con el identificador acortado.
- **Identificador Mal Formado (Error)**
  - **Causa:** Se produce por dos razones principales:
    1. El identificador no comienza con una letra mayúscula.
    2. Después del primer carácter, contiene un símbolo no permitido (cualquier cosa que no sea una letra mayúscula, un dígito o el carácter %).
  - **Tratamiento:** En ambos casos, el identificador es inválido. El analizador agrega un mensaje descriptivo a la lista de errores y retorna un token de ERROR, indicando al parser que la secuencia es incorrecta.

Errores en Constantes Numéricas

Estos errores se manejan en las acciones semánticas específicas para cada tipo de número.

- **Constante uint Fuera de Rango**
  - **Causa:** Una constante con sufijo UI tiene un valor que no está en el rango [0, 65535].
  - **Tratamiento:** La AccionSemantica4 usa BigDecimal para una comparación precisa. Si el valor está fuera de los límites, se registra un error y se retorna un token de ERROR.
- **Constante float Fuera de Rango**
  - **Causa:** Una constante de punto flotante excede los límites de un float de 32 bits.
  - **Tratamiento:** La AccionSemantica6 evalúa el valor. Si está fuera del rango permitido, registra un error y devuelve un token de ERROR. El manejo del signo negativo se comprueba en la parte sintáctica.

- **Formato Numérico Inválido**
    - **Causa:** La parte numérica de una constante no puede ser interpretada como un número. Por ejemplo, 12A3UI.
    - **Tratamiento:** El código intenta convertir el texto a número dentro de un bloque try-catch. Si falla (NumberFormatException), agrega un error de formato inválido y retorna un token de ERROR.

Error General

- **Carácter Inválido o No Reconocido**
    - **Causa:** Es el error más común. Se activa cuando el autómata encuentra un carácter para el cual no tiene una transición válida desde su estado actual.
    - **Tratamiento:** Se invoca la AccionSemanticaError. Esta acción registra un error que informa cuál fue el carácter inválido, reinicia el lexema para evitar problemas en cascada, y retorna el token de ERROR.

Trabajo Práctico 2: Análisis Sintáctico

Para la etapa de análisis sintáctico, se utilizó la herramienta BYACC/J para generar un parser a partir de la gramática formal del lenguaje, definida en el archivo gramatica.y.

Estructura de la Gramática (gramatica.y)

La gramática define la sintaxis del lenguaje. Los no terminales más relevantes son:

Estructura General del Programa

- **programa:** Es el símbolo inicial y representa la estructura completa del código, que consiste en un nombre opcional y un bloque de sentencias.
- **sentencias:** Define una secuencia de una o más sentencias, ya sean declarativas o ejecutables.
- **sentencia:** Representa una única instrucción del lenguaje, que puede ser una declaración, una acción ejecutable o una regla de error.

Declaraciones

- **sentencia_declarativa:** Agrupa todas las reglas que definen elementos, como variables o funciones.
- **declaracion_tipica:** Define la sintaxis para declarar variables con un tipo de dato explícito (ej. UINT a, b;).
- **declaracion_var:** Define la sintaxis para declarar una variable por inferencia, usando la palabra VAR.
- **tipo:** Enumera los tipos de datos válidos en declaraciones, como UINT, FLOAT o LAMBDA.
- **lista_variables:** Representa una secuencia de uno o más identificadores de variables, separados por comas.
- **funcion:** Define la estructura completa de una función, admitiendo retornos simples y múltiples.
- **lista_tipos_retorno_multiple:** Especifica la sintaxis para una lista de dos o más tipos en el retorno de una función.
- **lista_parametros_formales:** Representa la lista de parámetros en la definición de una función.
- **parametro_formal:** Define un único parámetro en la declaración de una función, que puede incluir la semántica de pasaje.
- **sem_pasaje:** Especifica la semántica de pasaje de parámetros, como CR SE (Copia-Resultado, Solo-Escritura).

Sentencias Ejecutables

- **sentencia_ejecutable:** Engloba todas las instrucciones que realizan acciones, como asignaciones, condicionales o bucles.
- **asignacion:** Maneja la asignación simple a una variable usando el operador :=.
- **asignacion_multiple:** Define la asignación a múltiples variables desde una lista de elementos, usando el operador =.
- **lado_derecho_multiple:** Representa el conjunto de elementos a la derecha de una asignación múltiple.
- **lista_elementos_restringidos:** Define una lista de factores para el lado derecho de una asignación múltiple, restringiendo el uso de expresiones complejas.
- **salida_pantalla:** Maneja la sentencia PRINT, que puede imprimir una cadena o una expresión.
- **retorno_funcion:** Define la sentencia RETURN, que puede devolver uno o más valores.
- **lista_expresiones:** Representa una secuencia de una o más expresiones separadas por comas, utilizada en la sentencia RETURN.

Expresiones y Variables

- **variable:** Representa un identificador, que puede ser simple (ID) o tener un prefijo opcional (ID.ID).
- **expresion:** Define la estructura de las operaciones aritméticas de suma y resta.
- **termino:** Define las operaciones de multiplicación y división, con mayor precedencia que expresión.
- **factor:** Es el elemento más básico de una expresión. Puede ser una variable, una constante, una llamada a función o una conversión de tipo.
- **constante:** Representa un valor literal, que puede ser positivo (CTE) o negativo (-CTE).
- **conversion_explicita:** Define la sintaxis para la conversión de tipos, como TOUI(...).

Funciones y Parámetros

- **invocacion_funcion:** Define la sintaxis para llamar a una función.
- **lista_parametros_reales:** Representa la lista de argumentos pasados en la llamada a una función.
- **parametro_real:** Define un único argumento en una llamada a función, especificando a qué parámetro formal corresponde (ej. argumento -> parámetro).
- **parametro_simple:** Representa el valor de un argumento, que puede ser una expresión o una expresión lambda.
- **lambda_expresion:** Define la sintaxis para una función anónima.
- **cuerpo_lambda:** Representa el bloque de código dentro de una expresión lambda.

Estructuras de Control

- **condicional_if:** Define la estructura de la sentencia de selección IF-ELSE-ENDIF.
- **condicional_do_while:** Define la estructura del bucle DO-WHILE.
- **condicion:** Representa una comparación entre dos expresiones, usada en los condicionales y bucles.
- **simbolo_comparacion:** Enumera todos los operadores de comparación válidos (>=, <, ==, etc.).
- **bloque_ejecutable:** Define un bloque de código que puede ser una única sentencia o un conjunto de ellas encerradas entre {}.
- **sentencias_ejecutables_lista:** Es una secuencia de una o más sentencias ejecutables, usada dentro de los bloques de código.

Construcción de la Gramática y Manejo de Errores

La gramática se construyó de manera incremental, comenzando por la regla principal programa y descendiendo jerárquicamente hasta las expresiones más simples como factor.

El manejo de errores sintácticos se implementó directamente en la gramática. Se utilizó la producción especial error ';' para capturar sentencias mal formadas y evitar que el parser se detuviera por completo. Adicionalmente, se redefinió el método yyerror para proporcionar mensajes de error más claros que incluyen el número de línea, facilitando la depuración del código fuente. Se crearon también reglas específicas para detectar errores comunes, como la falta de punto y coma en una sentencia DO-WHILE, lo que permite emitir un mensaje de error preciso en lugar de uno genérico.

Solución de Conflictos

Uno de los mayores desafíos fue la eliminación de conflictos para que la gramática no fuera ambigua. Se abordaron dos tipos principales de conflictos.

Conflicto Shift/Reduce

- **Problema:** El conflicto más significativo apareció al diferenciar una declaración de variable de una declaración de función con un único tipo de retorno. Cuando el parser leía una secuencia como UINT ID, no podía decidir si debía desplazar (SHIFT) el ID como parte de una lista de variables (ej. UINT miVar;) o si debía reducir (REDUCE) UINT a tipo para iniciar el reconocimiento de una función (ej. UINT miFunc(...)).
- **Solución Adoptada:** La ambigüedad se resolvió refactorizando las reglas para la declaración de funciones. Se crearon dos producciones distintas:
    1. Una para funciones de retorno simple, que comienza directamente con el no terminal tipo.
    2. Otra para funciones de retorno múltiple, que utiliza un nuevo no terminal lista_tipos_retorno_multiple. Este nuevo no terminal exige explícitamente la presencia de al menos dos tipos separados por comas (tipo ',' tipo).
- Esta separación eliminó la posibilidad de que un único tipo fuera interpretado como el inicio de una lista de retornos, resolviendo así el conflicto.

Conflicto Reduce/Reduce

- **Problema:** Se presentó un conflicto de doble reducción en la regla lado_derecho_multiple, que define los elementos permitidos a la derecha de una asignación múltiple. Una invocacion_funcion podría ser reconocida a través de dos caminos gramaticales distintos, lo que confundía al parser sobre qué regla aplicar.
- **Solución Adoptada:** Se eliminó la redundancia en la gramática. Dado que una invocacion_funcion ya estaba contemplada como un factor , y factor era la base para lista_elementos_restringidos, se suprimió la regla que permitía reducir invocacion_funcion directamente. Al dejar un único camino de reducción posible, el conflicto fue eliminado.

Decisiones de Implementación Clave

- **Manejo del "Dangling Else":** Se utilizó la directiva de precedencia %prec IFX para asociar correctamente cada ELSE con el IF más cercano y no resuelto, evitando ambigüedades en condicionales anidados.
- **Chequeo Semántico Temprano:** Se incluyó una acción semántica en la regla constante: '-' CTE para detectar el error de negar una constante uint, ya que no pueden tener signo negativo. Esto adelanta una validación semántica a la fase sintáctica.
- **Elección de Herramientas:** Para la implementación se optó por utilizar el lenguaje de programación Java en conjunto con el generador de parsers BYACC/J.

Conclusión

Hemos implementado con éxito las fases de análisis léxico y sintáctico. El analizador léxico es capaz de reconocer los tokens del lenguaje, validar rangos y gestionar la tabla de símbolos. El analizador sintáctico valida la estructura del código, manejando construcciones complejas y resolviendo ambigüedades gramaticales. El sistema de detección de errores informa al usuario de manera clara, permitiendo la continuación de la compilación. Las bases establecidas en estas etapas son fundamentales para avanzar hacia el análisis semántico y la generación de código.