Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Compiladores e intérpretes

Proyecto 1 Avance 2

Análisis Sintáctico

Profesor

Emmanuel Ramírez Segura

Subgrupo 5

Estudiantes

Antonio Fernández García - 2022075006

Marco Rodriguez Vargas - 2022149445

Josué Mena González - 2022138381

II Semestre 2024

Índice

Gramática

Lenguaje L1 – Especificación Formal

programa -> expresion ; { programa } ;
expresion -> identificador = expresion | termino { (+|-) termino } ;
termino -> factor { (*| /) factor } ;
factor -> identificador | numero | (expresion) ;
numero -> digito { digito } ;
digito -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Lenguaje L1 – Especificación Informal

- **Identificador**

Los identificadores son secuencias de una o más letras minúsculas, hasta un máximo de 12 letras.

- **{ }**

Los caracteres { y } se utilizan en las gramáticas formales para indicar repeticiones de elementos. En el contexto de las gramáticas de Backus-Naur (BNF) y otras notaciones similares, se utilizan para denotar que un elemento puede repetirse cero o más veces. Es equivalente a lo visto en la teoría de clase como (r)*. Por lo tanto no debe confundirse con símbolos terminales.

- **ws (White Spaces)**

Los espacios en blanco (tabs, espacios, enters) serán tomados como elementos separadores de los terminales. Será por lo tanto valido cualquiera de las siguientes notaciones:
✓ Válido: a=b+c;
✓ Válido: a = b + c ;
✓ Válido: a = b + c ;

Transformación de la gramática

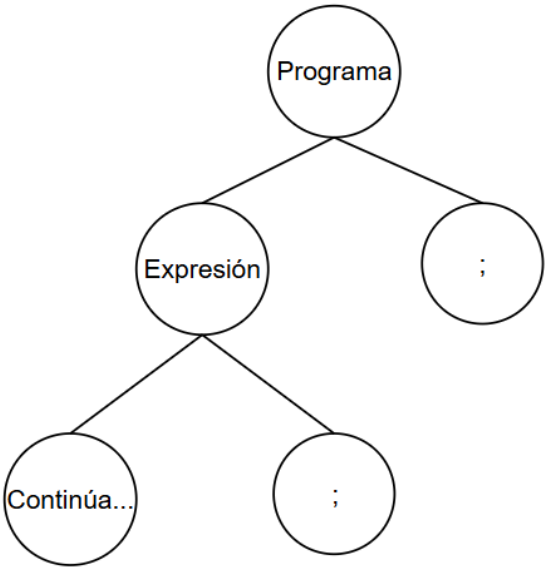
Al intentar realizar el análisis sintáctico con la gramática que está en la anterior página, se detectó errores sintácticos cuando no había, la razón de esto era ciertos detalles en la gramática, principalmente por la repetición de puntos y comas al final de una expresión, puntos y comas que no debían estar ahí y un problema relacionado con expresión

Empezando con la repetición de punto y coma al final de una expresión, se tiene que:

programa -> expresion **;** { programa } **;**

expresion -> identificador = expresion | termino { (+|-) termino } **;**

Si se observan los puntos y coma en negrita, se puede notar que si se tiene sólo programa, se produciría lo siguiente:



Esto generaría que haya dos punto y coma al final de una expresión, algo que no estaría correcto, por esta razón se realiza el siguiente cambio:

programa -> expresion **;** { programa }

expresion -> identificador = expresion | termino { (+|-) termino }

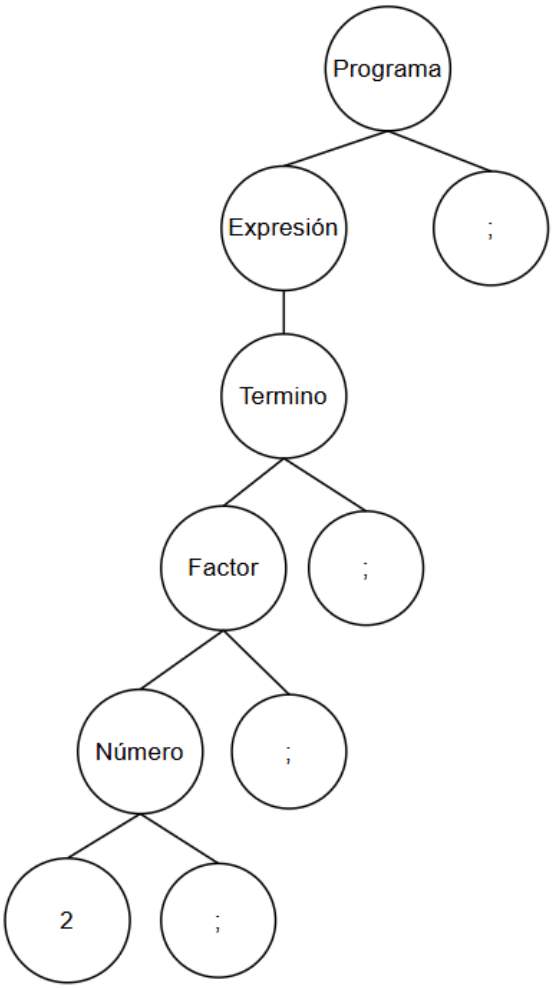
Ahora, hay otros punto y coma en la gramática que no deberían estar ahí, pues generan problemas similares, estos son:

termino -> factor { (*| /) factor } **;**

factor -> identificador | numero | (expresion) **;**

numero -> digito { digito } **;**

La principal razón es que las producciones anteriores terminan siendo números, identificadores o un conjunto de operaciones, es decir, no habría razón para que exista un punto y coma ahí, por ejemplo:



Para evitar que ocurra esto se quitan los punto y coma del final de cada uno, quedando de esta manera

termino -> factor { (*| /) factor }

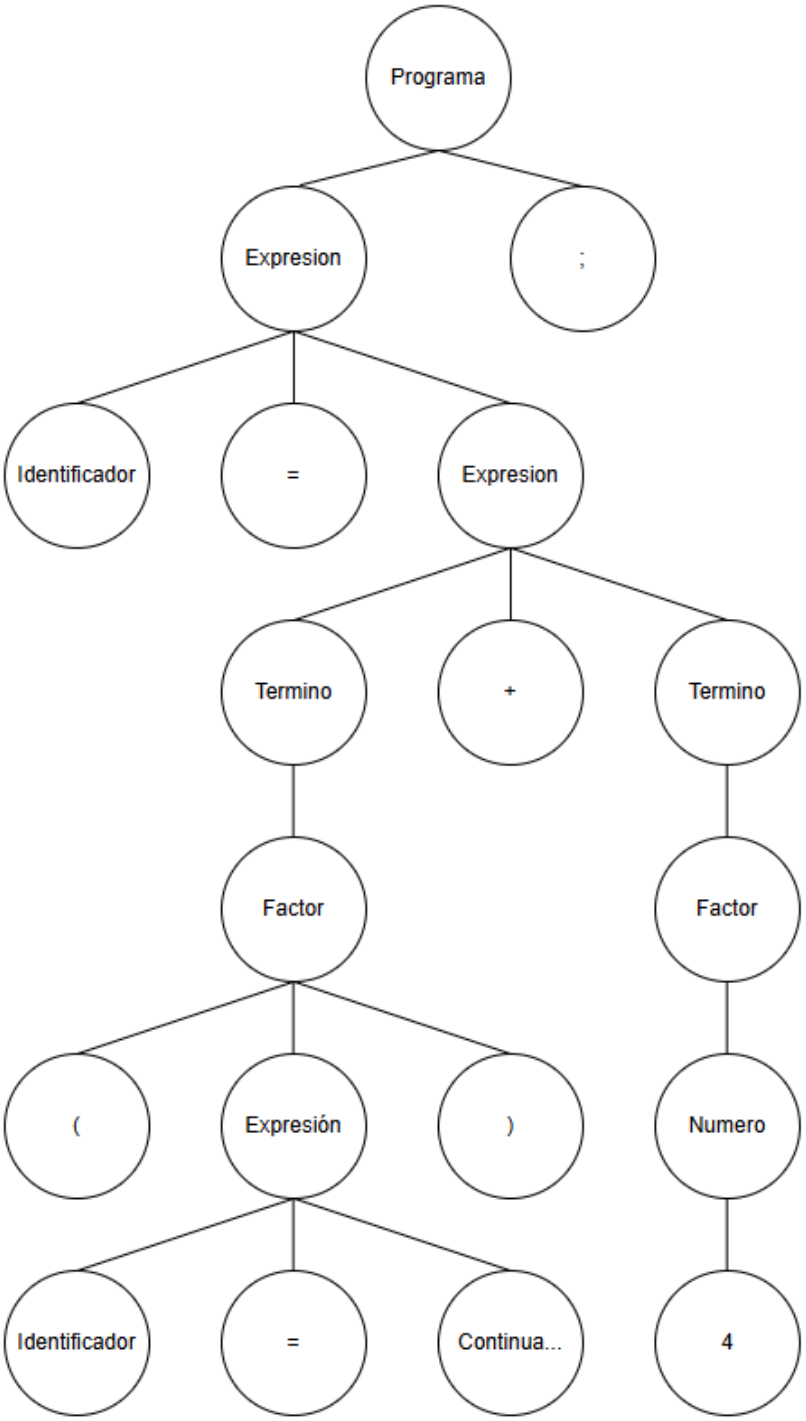
factor -> identificador | numero | (expresion)

numero -> digito { digito }

Por último, como se mencionó anteriormente, en la producción de “expresión” se debe corregir un detalle, actualmente se tiene que:

expresion -> identificador = expresion | termino { (+|-) termino }

Es decir, expresión puede derivar en “identificador” y “=”, esto generaría problemas cuando factor deriva en (expresión), por ejemplo:



Generando algo como:

Identificador = (Identificador = …..) + 4;

Para evitar esto se toma la decisión de realizar el siguiente cambio:

expresion -> identificador = expresión’
expresion’ -> termino { (+|-) termino }
factor -> identificador | numero | (expresion’)

Finalmente, la gramática transformada quedaría de la siguiente forma:

programa -> expresion ; { programa }
expresion -> identificador = expresión’
expresion’ -> termino { (+|-) termino }
termino -> factor { (*| /) factor }
factor -> identificador | numero | (expresion’)
numero -> digito { digito }
digito -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Analizador sintáctico utilizado

Para este avance se decidió hacer uso y programación del analizador descendente recursivo. Esto ya que los estudiantes consideraron que es una forma muy práctica e intuitiva de ver cómo se desarrollan las expresiones y también de ver cómo se viaja por el AST. Gracias a esta implementación también se pudo observar los errores que sufría la gramática originalmente, por esto se hicieron los cambios. A continuación se presentan los métodos de las producciones (no terminales) usadas para construir el parser.

Programa

Es el método principal para llamar al analizador sintáctico. Por medio de este inicia el análisis, llama a expresión, verifica que hay un punto y coma al final de cada expresión y se vuelve a llamar a la expresión en caso de ser necesario. Los diferentes errores llevan a borrar el id incorrecto de la tabla de símbolos.

```
public class FaseSintactica{

    public void Programa(List<Token> tokensitos){

        if(Expresion(tokensitos)){

                if(posicion < tokensitos.size()){

                    if(match(esperado:"PUNTO_COMA", tokensitos.get(posicion).getAtributo())){

                        System.out.println(x:"Encontre ;");
                        nroLinea++;
                        posicion++;

                        if(posicion == tokensitos.size()){

                            return;

                        }
                        else{

                            Programa(tokensitos);

                        }
                    }
                    else{

                        tabla.borrar(ultimoId);
                        System.out.println("Error [Fase Sintactica]: La linea " + nroLinea + " contiene un error en su gramatica, fal
                        return;

                    }
            else{

                tabla.borrar(ultimoId);
                System.out.println("Error [Fase Sintactica]: La linea " + nroLinea + " contiene un error en su gramatica, falta t
                return;

            }
    }
    else{

        tabla.borrar(ultimoId);
        System.out.println("Error [Fase Sintactica]: La linea " + nroLinea + " contiene un error en su gramatica, falta Expresiór
        return;

    }
}
```

Expresión

Este método se encarga de reconocer las expresiones que se forman en el lenguaje. Primero se hace match con el identificador. Luego, se busca el símbolo de asignación. Posteriormente se busca la Delta expresión que hay después del símbolo "=". En caso de que eso falle, se saca el identificador de la tabla de símbolos.

```
public boolean Expresion(List<Token> tokensitos){

    if(match(esperado:"IDENTIFICADOR", tokensitos.get(posicion).getAtributo()) ){

        ultimoId = tokensitos.get(posicion).getValor().toString();
        System.out.println("Encontre id: " + ultimoId);
        posicion++;

        if(match(esperado:"ASIGNACIÓN", tokensitos.get(posicion).getAtributo())){

            System.out.println(x:"Encontre =");
            posicion++;

            if(!DeltaExpresion(tokensitos)){

                System.out.println("Hay que sacar el id: "+ultimoId);
                tabla.borrar(ultimoId);
                return false;

            }else{

                return true;

            }
        }
        else{

            System.out.println("Error [Fase Sintactica]: La linea " + nroLinea + " contiene un error en su gramatica, falta token =");
            return false;

        }
    }

    return false;
}
```

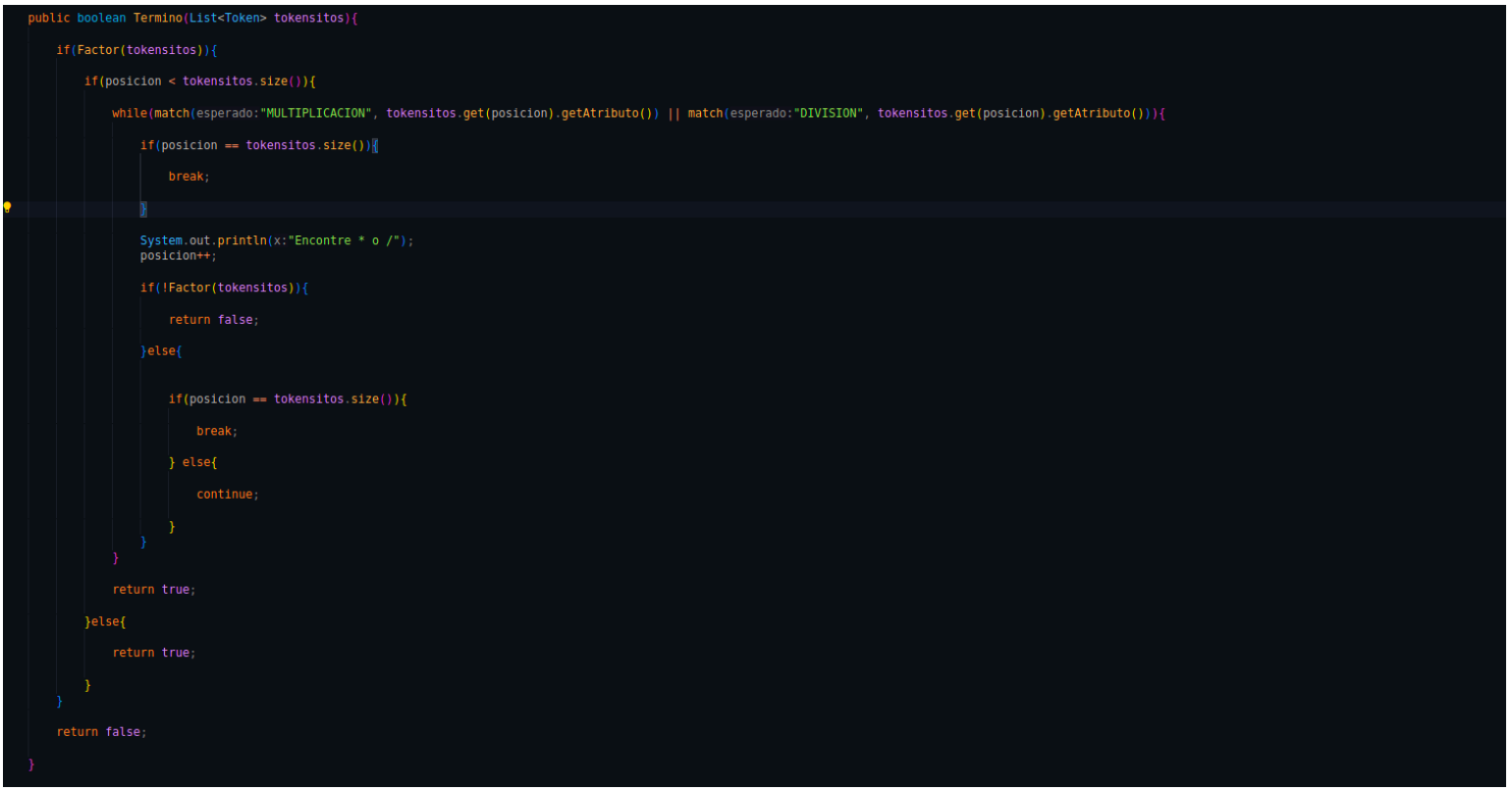
Delta expresión

Esto proviene de la gramática modificada. Por lo explicado anteriormente tener esto provee muchos beneficios sobre el manejo del análisis sintáctico. Este método primero busca por un Término, si lo encuentra procede a hacer match repetitivamente (en caso de haber) símbolos de suma o resta ya que según la gramática esto puede ser 0 o más veces. Luego, se procede a buscar otro término después del símbolo que se encontró. Si no hay, se devuelve un error y se retorna un false. Si no hay un término desde el inicio también se devuelve false y se da un error.

```
public boolean DeltaExpresion(List<Token> tokensitos){

    if(Termino(tokensitos)){

        if(posicion < tokensitos.size()){

            while(match(esperado:"SUMA", tokensitos.get(posicion).getAtributo()) || match(esperado:"RESTA", tokensitos.get(posicion).getAtributo())){

                if(posicion == tokensitos.size()){

                    break;

                }

                System.out.println(x:"Encontre + o -");
                posicion++;

                if(!Termino(tokensitos)){

                    return false;

                }else{

                    if(posicion == tokensitos.size()){

                        break;

                    } else{

                        continue;

                    }

                }
            }

            return true;

        }
        else{

            return true;

        }
    }

    return false;

}
```

Término

Término es un método bastante similar al anterior, nada más que este primero busca un factor y luego busca por multiplicación o división. En caso de encontrar alguno de los anteriores, busca otro factor nuevamente. Todo esto sigue el principio ya mencionado desde delta expresión. Recordemos que la búsqueda de multiplicaciones o divisiones por otro factor también es de 0 o más entonces se tiene que hacer repetidamente la búsqueda.



```
public boolean Termino(List<Token> tokensitos){

    if(Factor(tokensitos)){

        if(posicion < tokensitos.size()){

            while(match(esperado:"MULTIPLICACION", tokensitos.get(posicion).getAtributo()) || match(esperado:"DIVISION", tokensitos.get(posicion).getAtributo())){

                if(posicion == tokensitos.size()){

                    break;

                }

                System.out.println(x:"Encontre * o /");
                posicion++;

                if(!Factor(tokensitos)){

                    return false;

                }else{


                    if(posicion == tokensitos.size()){

                        break;

                    } else{

                        continue;

                    }
                }
            }

            return true;

        }else{

            return true;

        }
    }

    return false;

}
```

Factor

La última de las producciones para este analizador. Factor es clave porque ya empieza a reconocer otros token de tipo terminal, como lo son números, paréntesis y otros identificadores. Su funcionamiento se basa en buscar alguno de los siguientes casos: Identificador, un número o un paréntesis izquierdo. Este último caso conlleva analizar si después del paréntesis de apertura se tiene una delta expresión, dado esto, si es encontrada se busca el paréntesis de cierre o derecho. Si no se cumplen los casos especificados y sus subcasos se encuentra y se devuelve un error.

```java
public boolean Factor(List<Token> tokensitos){

    if(match(esperado:"IDENTIFICADOR", tokensitos.get(posicion).getAtributo())){

        String idEncontrado = tokensitos.get(posicion).getValor().toString();
        System.out.println("Encontre id: " +idEncontrado);
        posicion++;
        return true;
    }
    else if(match(esperado:"NUMERO", tokensitos.get(posicion).getAtributo())){

        String num = tokensitos.get(posicion).getValor().toString();
        System.out.println("Encontre numero: "+num);
        posicion++;
        return true;
    }
    else if(match(esperado:"PARENTESIS_IZQ", tokensitos.get(posicion).getAtributo())){

        posicion++;
        System.out.println(x:"Encontre (");
        if(DeltaExpresion(tokensitos)){

            if(posicion < tokensitos.size() && match(esperado:"PARENTESIS_DER", tokensitos.get(posicion).getAtributo())){

                System.out.println(x:"Encontre )");
                posicion++;
                return true;

            }
            else{

                System.out.println("Error [Fase Sintactica]: La linea " + nroLinea + " contiene un error en su gramatica, falta token )");
                return false;

            }
        }
        else{

            return false;

        }
    }
    else{

        System.out.println("Error [Fase Sintactica]: La linea " + nroLinea + " contiene un error en su gramatica, falta token id o numero o (");
        return false;

    }
}
```

Match

Este método permite hacer match de un token buscado con el que se ingresó. Básicamente se evalúa el string de atributo de un token con un string esperado.

```java
public boolean match(String esperado, String entrada){

    if(esperado == entrada){ return true;}
    else{return false;}
}
```

Pruebas funcionales

1. prueba1.txt:

```
a = 42;
b = a - 7
c = (a + b * 2;
```

En la primera línea, a = 42;, a es un identificador. La expresión a la derecha es válida porque 42 es un número y la asignación termina correctamente con punto y coma.

En la segunda línea, b = a - 7, b es un identificador. La expresión a la derecha es válida porque a es un identificador y 7 es un número pero falta el punto y coma al final lo que genera un error sintáctico.

En la tercera línea, c = (a + b * 2;, c es un identificador. La expresión a la derecha contiene una operación válida pero falta un paréntesis de cierre lo que significa un error sintáctico.

```
marco@marco-B660M-AORUS-PRO-AX-DDR4:~/CompiProyecto1/miCompilador$ java -cp out App prueba1.txt

 --- Fase Lexica ---

 Valor: a, Tipo: IDENTIFICADOR
 Valor: =, Tipo: ASIGNACION
 Valor: 42, Tipo: NUMERO
 Valor: ;, Tipo: PUNTO_COMA
 Valor: b, Tipo: IDENTIFICADOR
 Valor: =, Tipo: ASIGNACION
 Valor: a, Tipo: IDENTIFICADOR
 Valor: -, Tipo: RESTA
 Valor: 7, Tipo: NUMERO
 Valor: c, Tipo: IDENTIFICADOR
 Valor: =, Tipo: ASIGNACION
 Valor: (, Tipo: PARENTESIS_IZQ
 Valor: a, Tipo: IDENTIFICADOR
 Valor: +, Tipo: SUMA
 Valor: b, Tipo: IDENTIFICADOR
 Valor: *, Tipo: MULTIPLICACION
 Valor: 2, Tipo: NUMERO
 Valor: ;, Tipo: PUNTO_COMA

 --- Tabla de simbolos antes de la fase sintactica ---

 Nombre token: a
 Informacion:
 Tipo: entero
 Ambito: global
 Números de linea: [1, 2, 2]

 Nombre token: b
 Informacion:
 Tipo: entero
 Ambito: global
 Números de linea: [2, 2]

 Nombre token: c
 Informacion:
 Tipo: entero
 Ambito: global
 Números de linea: [2]
```

```
--- Fase Sintactica ---

Encontre id: a
Encontre =
Encontre numero: 42
Encontre ;
Encontre id: b
Encontre =
Encontre id: a
Encontre + o -
Encontre numero: 7
Error [Fase Sintactica]: La linea 2 contiene un error en su gramatica, falta token ;

--- Tabla de simbolos despues de la fase sintactica ---

Nombre token: a
Informacion:
Tipo: entero
Ambito: global
Números de linea: [1, 2, 2]

Nombre token: c
Informacion:
Tipo: entero
Ambito: global
Números de linea: [2]
```

2. prueba2.txt:

```
q = 5;
w = q + (3 * 2);
e = (w - 4) / 2;
```

En la primera línea, q = 5;, q es un identificador. La expresión a la derecha es válida porque 5 es un número y la asignación termina correctamente con punto y coma.

En la segunda línea, w = q + (3 * 2);, w es un identificador. La expresión a la derecha es válida, con q como identificador y la multiplicación de 3 * 2 correctamente agrupada entre paréntesis. La línea termina correctamente con punto y coma.

En la tercera línea, e = (w - 4) / 2;, e es un identificador. La expresión a la derecha es válida, con la resta de w - 4 agrupada entre paréntesis antes de realizar la división por 2. La línea también termina correctamente con punto y coma.

```
--- Fase Lexica ---

Valor: q, Tipo: IDENTIFICADOR
Valor: =, Tipo: ASIGNACION
Valor: 5, Tipo: NUMERO
Valor: ;, Tipo: PUNTO_COMA
Valor: w, Tipo: IDENTIFICADOR
Valor: =, Tipo: ASIGNACION
Valor: q, Tipo: IDENTIFICADOR
Valor: +, Tipo: SUMA
Valor: (, Tipo: PARENTESIS_IZQ
Valor: 3, Tipo: NUMERO
Valor: *, Tipo: MULTIPLICACION
Valor: 2, Tipo: NUMERO
Valor: ), Tipo: PARENTESIS_DER
Valor: ;, Tipo: PUNTO_COMA
Valor: e, Tipo: IDENTIFICADOR
Valor: =, Tipo: ASIGNACION
Valor: (, Tipo: PARENTESIS_IZQ
Valor: w, Tipo: IDENTIFICADOR
Valor: -, Tipo: RESTA
Valor: 4, Tipo: NUMERO
Valor: ), Tipo: PARENTESIS_DER
Valor: /, Tipo: DIVISION
Valor: 2, Tipo: NUMERO
Valor: ;, Tipo: PUNTO_COMA

--- Tabla de simbolos antes de la fase sintactica ---

Nombre token: q
Informacion:
Tipo: entero
Ambito: global
Números de linea: [1, 2]

Nombre token: e
Informacion:
Tipo: entero
Ambito: global
Números de linea: [3]

Nombre token: w
Informacion:
Tipo: entero
Ambito: global
Números de linea: [2, 3]
```

```
--- Fase Sintactica ---

Encontre id: q
Encontre =
Encontre numero: 5
Encontre ;
Encontre id: w
Encontre =
Encontre id: q
Encontre + o -
Encontre (
Encontre numero: 3
Encontre * o /
Encontre numero: 2
Encontre )
Encontre ;
Encontre id: e
Encontre =
Encontre (
Encontre id: w
Encontre + o -
Encontre numero: 4
Encontre )
Encontre * o /
Encontre numero: 2
Encontre ;

--- Tabla de simbolos despues de la fase sintactica ---

Nombre token: q
Informacion:
Tipo: entero
Ambito: global
Números de linea: [1, 2]

Nombre token: e
Informacion:
Tipo: entero
Ambito: global
Números de linea: [3]

Nombre token: w
Informacion:
Tipo: entero
Ambito: global
Números de linea: [2, 3]
```

3. prueba3.txt:

```
x = 2 * 3 + 4 / 2;
y = x - (x * 2) + (4 / 2);
z = y + (x + 3);
```

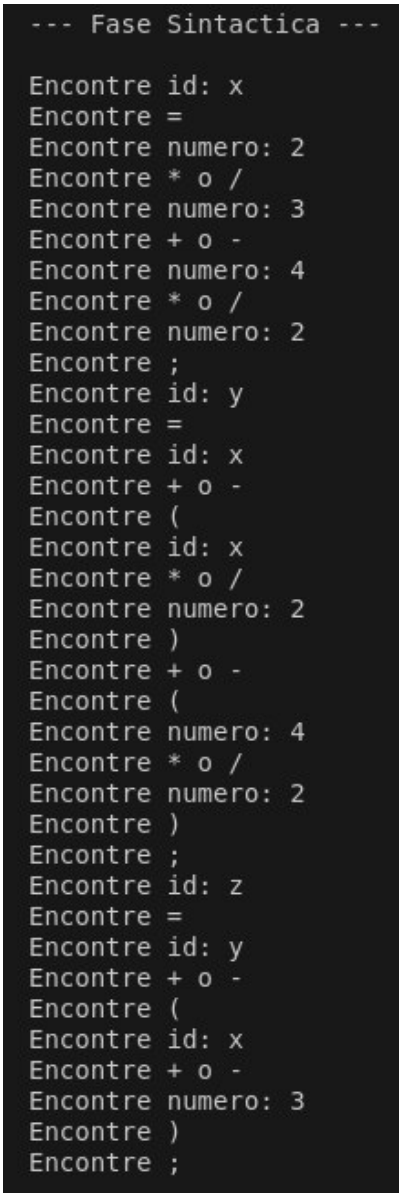
En la primera línea, x = 2 * 3 + 4 / 2;, x es un identificador. Las operaciones de suma, multiplicación y división son válidas. La línea termina correctamente con punto y coma.

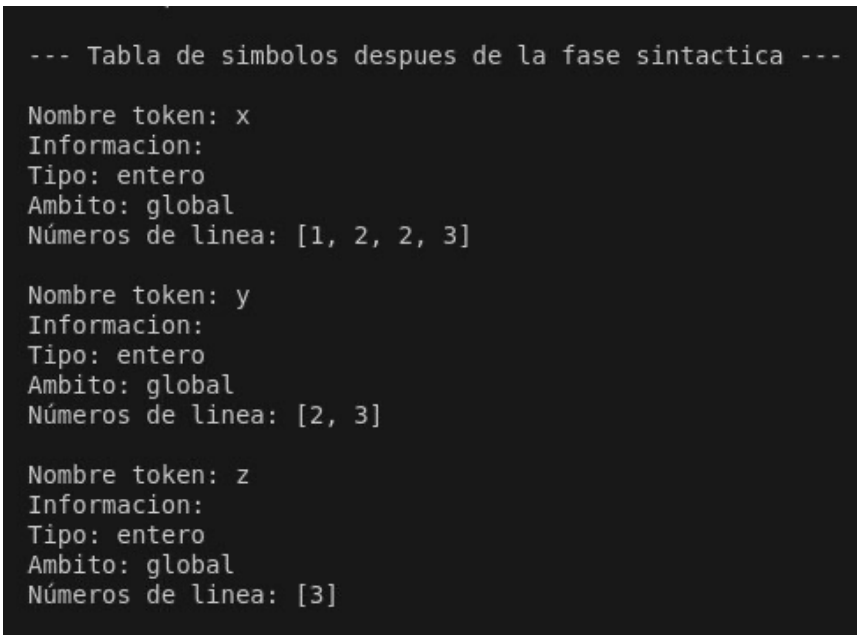
En la segunda línea, y = x - (x * 2) + (4 / 2);, y es un identificador. La expresión a la derecha es válida, con las operaciones agrupadas correctamente entre paréntesis para su evaluación antes del resto de operaciones. La línea termina correctamente con punto y coma.

En la tercera línea, z = y + (x + 3);, z es un identificador. La expresión a la derecha es válida con las sumas correctamente agrupadas entre paréntesis. La línea termina correctamente con punto y coma.

```
--- Fase Lexica ---

Valor: x, Tipo: IDENTIFICADOR
Valor: =, Tipo: ASIGNACION
Valor: 2, Tipo: NUMERO
Valor: *, Tipo: MULTIPLICACION
Valor: 3, Tipo: NUMERO
Valor: +, Tipo: SUMA
Valor: 4, Tipo: NUMERO
Valor: /, Tipo: DIVISION
Valor: 2, Tipo: NUMERO
Valor: ;, Tipo: PUNTO_COMA
Valor: y, Tipo: IDENTIFICADOR
Valor: =, Tipo: ASIGNACION
Valor: x, Tipo: IDENTIFICADOR
Valor: -, Tipo: RESTA
Valor: (, Tipo: PARENTESIS_IZQ
Valor: x, Tipo: IDENTIFICADOR
Valor: *, Tipo: MULTIPLICACION
Valor: 2, Tipo: NUMERO
Valor: ), Tipo: PARENTESIS_DER
Valor: +, Tipo: SUMA
Valor: (, Tipo: PARENTESIS_IZQ
Valor: 4, Tipo: NUMERO
Valor: /, Tipo: DIVISION
Valor: 2, Tipo: NUMERO
Valor: ), Tipo: PARENTESIS_DER
Valor: ;, Tipo: PUNTO_COMA
Valor: z, Tipo: IDENTIFICADOR
Valor: =, Tipo: ASIGNACION
Valor: y, Tipo: IDENTIFICADOR
Valor: +, Tipo: SUMA
Valor: (, Tipo: PARENTESIS_IZQ
Valor: x, Tipo: IDENTIFICADOR
Valor: +, Tipo: SUMA
Valor: 3, Tipo: NUMERO
Valor: ), Tipo: PARENTESIS_DER
Valor: ;, Tipo: PUNTO_COMA
```

```
--- Tabla de simbolos antes de la fase sintactica ---

Nombre token: x
Informacion:
Tipo: entero
Ambito: global
Números de linea: [1, 2, 2, 3]

Nombre token: y
Informacion:
Tipo: entero
Ambito: global
Números de linea: [2, 3]

Nombre token: z
Informacion:
Tipo: entero
Ambito: global
Números de linea: [3]
```


--- Fase Sintactica ---
Encontre id: x
Encontre =
Encontre numero: 2
Encontre * o /
Encontre numero: 3
Encontre + o -
Encontre numero: 4
Encontre * o /
Encontre numero: 2
Encontre ;
Encontre id: y
Encontre =
Encontre id: x
Encontre + o -
Encontre (
Encontre id: x
Encontre * o /
Encontre numero: 2
Encontre)
Encontre + o -
Encontre (
Encontre numero: 4
Encontre * o /
Encontre numero: 2
Encontre)
Encontre ;
Encontre id: z
Encontre =
Encontre id: y
Encontre + o -
Encontre (
Encontre id: x
Encontre + o -
Encontre numero: 3
Encontre)
Encontre ;


--- Tabla de simbolos despues de la fase sintactica ---
Nombre token: x
Informacion:
Tipo: entero
Ambito: global
Números de linea: [1, 2, 2, 3]
Nombre token: y
Informacion:
Tipo: entero
Ambito: global
Números de linea: [2, 3]
Nombre token: z
Informacion:
Tipo: entero
Ambito: global
Números de linea: [3]

4. prueba4.txt

```
a = (2 + 3) * (4 - 1);
b = a / (5 + (2 * 3));
c = (a + b) - (2 / (1 + 1));
```

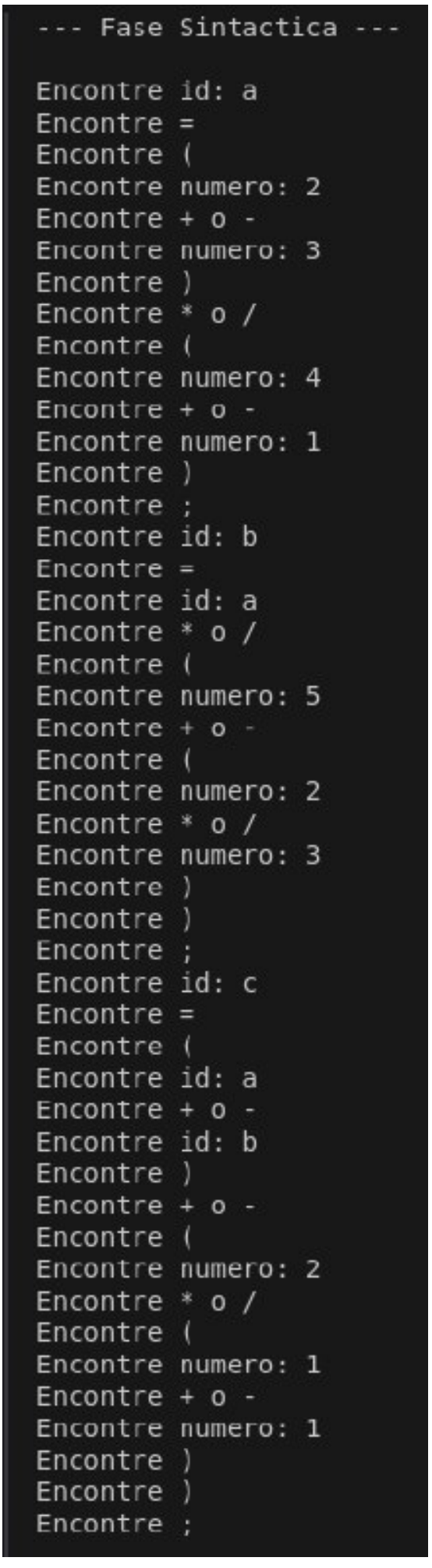
En la primera línea, a = (2 + 3) * (4 - 1);, a es un identificador. La expresión a la derecha es válida con las operaciones de suma y resta correctamente agrupadas entre paréntesis y el resultado de la suma multiplicado por el resultado de la resta. La línea termina correctamente con punto y coma.

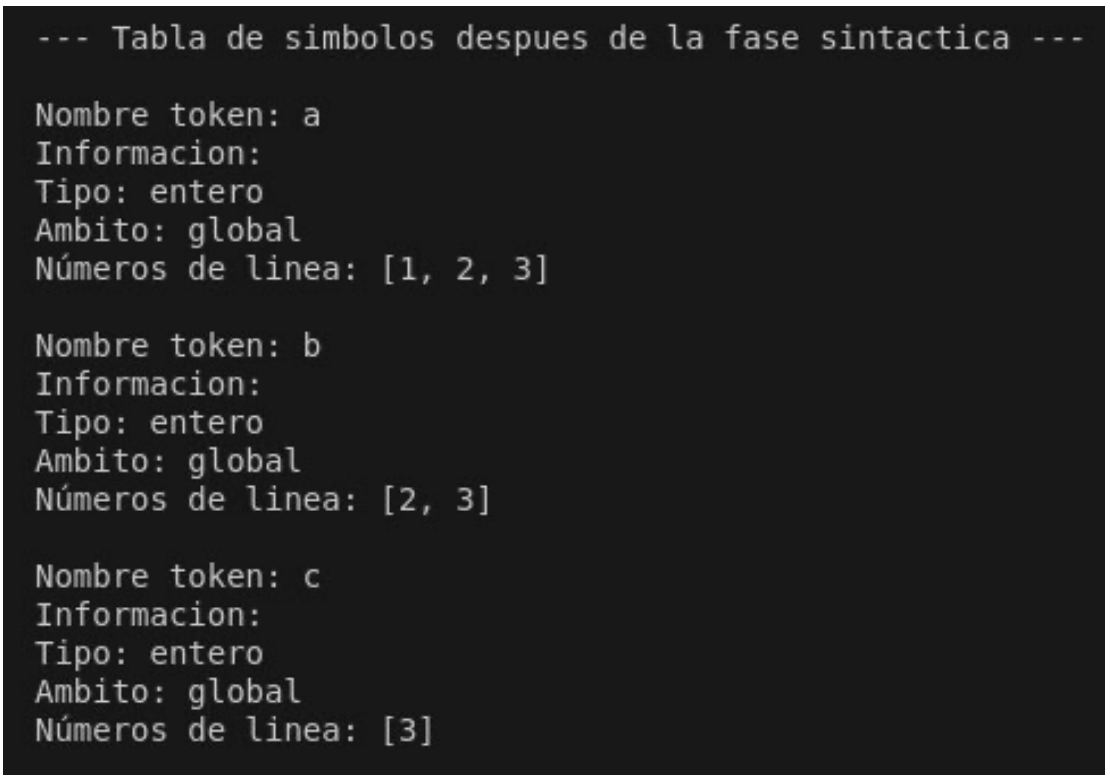
En la segunda línea, b = a / (5 + (2 * 3));, b es un identificador. La expresión a la derecha es válida con la multiplicación de 2 * 3 correctamente agrupada dentro de los paréntesis seguida de la suma con 5. El resultado de esa operación se usa para dividir el valor de a. La línea termina correctamente con punto y coma.

En la tercera línea, c = (a + b) - (2 / (1 + 1));, c es un identificador. La expresión a la derecha es válida con la suma de a y b agrupada entre paréntesis y la división de 2 / (1 + 1) también correctamente anidada entre paréntesis. La resta entre los dos resultados está correctamente estructurada. La línea termina correctamente con punto y coma.

```
--- Fase Lexica ---

Valor: a, Tipo: IDENTIFICADOR
Valor: =, Tipo: ASIGNACION
Valor: (, Tipo: PARENTESIS_IZQ
Valor: 2, Tipo: NUMERO
Valor: +, Tipo: SUMA
Valor: 3, Tipo: NUMERO
Valor: ), Tipo: PARENTESIS_DER
Valor: *, Tipo: MULTIPLICACION
Valor: (, Tipo: PARENTESIS_IZQ
Valor: 4, Tipo: NUMERO
Valor: -, Tipo: RESTA
Valor: 1, Tipo: NUMERO
Valor: ), Tipo: PARENTESIS_DER
Valor: ;, Tipo: PUNTO_COMA
Valor: b, Tipo: IDENTIFICADOR
Valor: =, Tipo: ASIGNACION
Valor: a, Tipo: IDENTIFICADOR
Valor: /, Tipo: DIVISION
Valor: (, Tipo: PARENTESIS_IZQ
Valor: 5, Tipo: NUMERO
Valor: +, Tipo: SUMA
Valor: (, Tipo: PARENTESIS_IZQ
Valor: 2, Tipo: NUMERO
Valor: *, Tipo: MULTIPLICACION
Valor: 3, Tipo: NUMERO
Valor: ), Tipo: PARENTESIS_DER
Valor: ), Tipo: PARENTESIS_DER
Valor: ;, Tipo: PUNTO_COMA
Valor: c, Tipo: IDENTIFICADOR
Valor: =, Tipo: ASIGNACION
Valor: (, Tipo: PARENTESIS_IZQ
Valor: a, Tipo: IDENTIFICADOR
Valor: +, Tipo: SUMA
Valor: b, Tipo: IDENTIFICADOR
Valor: ), Tipo: PARENTESIS_DER
Valor: -, Tipo: RESTA
Valor: (, Tipo: PARENTESIS_IZQ
Valor: 2, Tipo: NUMERO
Valor: /, Tipo: DIVISION
Valor: (, Tipo: PARENTESIS_IZQ
Valor: 1, Tipo: NUMERO
Valor: +, Tipo: SUMA
Valor: 1, Tipo: NUMERO
Valor: ), Tipo: PARENTESIS_DER
Valor: ), Tipo: PARENTESIS_DER
Valor: ;, Tipo: PUNTO_COMA
```

```
--- Tabla de simbolos antes de la fase sintactica ---

Nombre token: a
Informacion:
Tipo: entero
Ambito: global
Números de linea: [1, 2, 3]

Nombre token: b
Informacion:
Tipo: entero
Ambito: global
Números de linea: [2, 3]

Nombre token: c
Informacion:
Tipo: entero
Ambito: global
Números de linea: [3]
```


--- Fase Sintactica ---
Encontre id: a
Encontre =
Encontre (
Encontre numero: 2
Encontre + o -
Encontre numero: 3
Encontre)
Encontre * o /
Encontre (
Encontre numero: 4
Encontre + o -
Encontre numero: 1
Encontre)
Encontre ;
Encontre id: b
Encontre =
Encontre id: a
Encontre * o /
Encontre (
Encontre numero: 5
Encontre + o -
Encontre (
Encontre numero: 2
Encontre * o /
Encontre numero: 3
Encontre)
Encontre)
Encontre ;
Encontre id: c
Encontre =
Encontre (
Encontre id: a
Encontre + o -
Encontre id: b
Encontre)
Encontre + o -
Encontre (
Encontre numero: 2
Encontre * o /
Encontre (
Encontre numero: 1
Encontre + o -
Encontre numero: 1
Encontre)
Encontre)
Encontre ;


--- Tabla de simbolos despues de la fase sintactica ---
Nombre token: a
Informacion:
Tipo: entero
Ambito: global
Números de linea: [1, 2, 3]
Nombre token: b
Informacion:
Tipo: entero
Ambito: global
Números de linea: [2, 3]
Nombre token: c
Informacion:
Tipo: entero
Ambito: global
Números de linea: [3]

5. errores.txt:

```
x = 5 + ;
y = (2 * 3;
z = y + 4) / ;
w = 7 + 3
```

En la primera línea, x = 5 + ;, x es un identificador. La expresión a la derecha está incompleta ya que falta un número o identificador después del operador +. Esto provoca un error sintáctico.

En la segunda línea, y = (2 * 3;, y es un identificador. La expresión contiene un error porque falta el paréntesis de cierre lo que significa un error sintáctico.

En la tercera línea, z = y + 4) / ;, z es un identificador. La expresión tiene varios errores como el paréntesis de cierre que está mal ubicado y falta un operando después del operador /. Esto significa varios errores sintácticos.

En la cuarta línea, w = 7 + 3, w es un identificador. La expresión es válida pero falta el punto y coma al final de la línea, lo que significa un error sintáctico.

```
marco@marco-B660M-AORUS-PRO-AX-DDR4:~/CompiProyecto1/miCompilador$ java -cp out App errores.txt

--- Fase Lexica ---

Valor: x, Tipo: IDENTIFICADOR
Valor: =, Tipo: ASIGNACION
Valor: 5, Tipo: NUMERO
Valor: +, Tipo: SUMA
Valor: ;, Tipo: PUNTO_COMA
Valor: y, Tipo: IDENTIFICADOR
Valor: =, Tipo: ASIGNACION
Valor: (, Tipo: PARENTESIS_IZQ
Valor: 2, Tipo: NUMERO
Valor: *, Tipo: MULTIPLICACION
Valor: 3, Tipo: NUMERO
Valor: ;, Tipo: PUNTO_COMA
Valor: z, Tipo: IDENTIFICADOR
Valor: =, Tipo: ASIGNACION
Valor: y, Tipo: IDENTIFICADOR
Valor: +, Tipo: SUMA
Valor: 4, Tipo: NUMERO
Valor: ), Tipo: PARENTESIS_DER
Valor: /, Tipo: DIVISION
Valor: ;, Tipo: PUNTO_COMA
Valor: w, Tipo: IDENTIFICADOR
Valor: =, Tipo: ASIGNACION
Valor: 7, Tipo: NUMERO
Valor: +, Tipo: SUMA
Valor: 3, Tipo: NUMERO
```

```
--- Tabla de simbolos antes de la fase sintactica ---

Nombre token: w
Informacion:
Tipo: entero
Ambito: global
Números de linea: [4]

Nombre token: x
Informacion:
Tipo: entero
Ambito: global
Números de linea: [1]

Nombre token: y
Informacion:
Tipo: entero
Ambito: global
Números de linea: [2, 3]

Nombre token: z
Informacion:
Tipo: entero
Ambito: global
Números de linea: [3]
```

```
--- Fase Sintactica ---

Encontre id: x
Encontre =
Encontre numero: 5
Encontre + o -
Error [Fase Sintactica]: La linea 1 contiene un error en su gramatica, falta token id o numero o (
Hay que sacar el id: x
Error [Fase Sintactica]: La linea 1 contiene un error en su gramatica, falta Expresión

--- Tabla de simbolos despues de la fase sintactica ---

Nombre token: w
Informacion:
Tipo: entero
Ambito: global
Números de linea: [4]

Nombre token: y
Informacion:
Tipo: entero
Ambito: global
Números de linea: [2, 3]

Nombre token: z
Informacion:
Tipo: entero
Ambito: global
Números de linea: [3]
```

Referencias

Aho, A., Lam, M., Sethi, R., & Ullman, J. (2006). Compilers: Principles, techniques and tools (2nd. Edition ed.). Addison-Wesley.