Instituto Tecnológico de Costa Rica

Escuela de computación

Compiladores E Intérpretes GR20

Avance 2 Proyecto#1 Compiladores e intérpretes

Estudiantes:

Mariano Oconitrillo Vega- 2021080289

Brandon Rodríguez Vega - 2021152632

Sigifredo Chacón Balladares – 2021067590

Sub-grupo: 2

Profesor:

Roger Emmanuel Ramirez Segura

Centro Académico de Alajuela

Semestre I

2023

INDICE

1. Gramática 2
2. Transformación de la gramática 2
3. Analizador sintáctico utilizado 2
   3.1. Implementación. 3
      3.1.1. Clase de la fase sintáctica. 3
      3.1.2. Funciones auxiliares. 4
      3.1.3. Funciones de las producciones del lenguaje. 8
4. Pruebas funcionales 15

1. Gramática

Lenguaje L1

```
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
```

2. Transformación de la gramática

Para este caso no se requirió transformar la gramática del lenguaje L1 para implementar el analizador sintáctico, esto debido a que al no transformarla se reduce significativamente la complejidad de la implementación, también, al mantener la gramática original, la implementación en código es más fácil de depurar ya que los errores pueden estar directamente relacionados con la sintaxis del lenguaje fuente, por lo que se facilitaría el rastreo de los errores como también su debido arreglo. Además, al evitar transformaciones gramaticales intermedias, se minimiza la introducción de errores sutiles y difíciles de detectar.

3. Analizador sintáctico utilizado

Para este proyecto se eligió el analizador sintáctico descendente recursivo con Back-Tracking, esta elección se justifica en gran medida por la facilidad de implementación que este tipo de analizadores ofrece. En comparación con otros métodos de análisis sintáctico más avanzados y difíciles, como el análisis LL(k) o LR(k), el descendente recursivo con Back-Tracking es relativamente sencillo de diseñar y programar. Esto reduce del tiempo necesario para desarrollar el analizador, lo que es particularmente valioso en este proyecto. Además, la simplicidad de usar el Back-Tracking facilita la comprensión del código del parser. En resumen, la elección del Back-Tracking se basa en su accesibilidad y su capacidad para agilizar la implementación del analizador, lo que lo convierte en la mejor opción para nuestro grupo.

3.1. Implementación.

A continuación, se presentarán todas las partes de la implementación del analizador sintáctico descendente recursivo con Back-Tracking.

3.1.1. Clase de la fase sintáctica.

Para la implementación, se creó la clase en Java llamada “faseSintactica” , esta clase tiene varios atributos, primero se tiene una lista de error en la cual si se llegara a dar un error en esta fase se guardarían ahí para posteriormente mostrarlos, luego se tienen una lista de Tokens, los cuales vienen del analizador léxico desarrollado anteriormente, luego se tiene el valor de la posición el cual nos dice por cual Token o parte del código ingresado se esta leyendo, luego se tiene el número de línea, el cual nos dice por cual línea de código estamos, este atributo más que todo sirve para cuando se indiquen los errores poder decir exactamente en cual línea es, se tiene también un contador de paréntesis izquierdos, este atributo sirve para validar el error cuando hacen falta paréntesis izquierdos, más adelante se explicara como funciona, luego tenemos nuestro hashmap donde tenemos nuestros identificadores que también esto viene del analizador léxico, por ultimo tenemos dos listas las cuales son la lista de las líneas de errores y la lista de identificadores las cuales su uso las vamos a explicar más adelante.

faseSintactica.java

```
1   import ...
3
4   //En esta clase llamada faseSintactica nos encargaremos de hacer como tal el analisis sintactico,
5   //es decir realizamos el proceso de reconocimiento de la sintaxis del codigo y de los errores.
    2 usages
6   public class faseSintactica {
7
        15 usages
8       private ArrayList<String> listaErrores = new ArrayList<>();
        42 usages
9       private ArrayList<Token> listaToken;
        50 usages
10      private int posicion;
        31 usages
11      private int numLinea = 1;
        7 usages
12      private int contadorParentesisIzquierdo = 0;
13
        3 usages
14      private HashMap<String, String> hashMap;
        18 usages
15      private ArrayList<Integer> listaLineasErrores = new ArrayList<>();
        4 usages
16      private ArrayList<Token> listaIdentificadores = new ArrayList<>();
17
```

3.1.2. Funciones auxiliares.

Para la implementación se requirió el uso de funciones auxiliares las cuales sirven para reducir el código, facilitar la lectura y escritura de este, como también cumplir otras funciones.

- **Funciones para modificar la tabla de símbolos:**

Para lograr que la tabla de símbolos este correcta se debe ver que los identificadores que vienen de la fase léxica estén correctos en la fase sintáctica, o sea que en la línea donde están estos identificadores no haya errores sintácticos, esto debido a que si hay errores con estos, en la fase semántica también los habría por lo que hay que quitarlos de la tabla de símbolos.

Eliminar duplicados:

Tenemos primero la función “eliminarDuplicados” la cual lo que hace es eliminar de la lista de errores, que recordando es una lista de enteros que nos dice en que líneas hay errores, pero si hay más de un error en la misma línea, esta línea va a aparecer varias veces en la lista por lo que hay que quitar estas duplicaciones. Esto lo hace primero creando una nueva ArrayList llamada “listaSinDuplicados” que se utilizará para almacenar los números sin duplicados, luego se inicia un bucle for que itera a través de cada elemento en la lista original “listaLineasErrores”, dentro del bucle, se verifica si “listaSinDuplicados” ya contiene el número actual. Si no lo contiene (lo que significa que es la primera vez que se encuentra este número), se agrega a listaSinDuplicados utilizando el método add, después de completar la eliminación de duplicados, la lista original “listaLineasErrores” se borra por completo utilizando clear, y al final, los elementos únicos de “listaSinDuplicados” se agregan nuevamente a listaLineasErrores utilizando addAll, lo que actualiza la lista original con los elementos únicos.

```
public void eliminarDuplicados(){
    ArrayList<Integer> listaSinDuplicados = new ArrayList<>();

    for (Integer numero : listaLineasErrores) {
        if (!listaSinDuplicados.contains(numero)) {
            listaSinDuplicados.add(numero);
        }
    }
    listaLineasErrores.clear();
    listaLineasErrores.addAll(listaSinDuplicados);
}
```

Eliminar identificadores:

Luego se tiene la función “eliminarIdentificadores” esta compara la lista de identificadores en la cual hay tokens y la lista de líneas de errores, pero la lista de identificadores no guarda tokens exactamente como se ha estado haciendo, porque en este caso se guarda el valor del identificador y la línea de este identificador, en vez del tipo del token, ese es el único cambio, estos identificadores se van guardando cuando se está ejecutando el analizador pero solo se guardan los que están antes de un igual y no identificadores que estén después, por ejemplo “x = 24;” en este caso se guarda la “x”, en el caso que sea por ejemplo “x = b + 12;” solo se guardaría la “x” pero no la “b” y por ultimo si tenemos “c + 23;” no se guardaría la “c” por qué está en una expresión que no tienen un igual. Ya aclarado esto, la función primero llama al método eliminarDuplicados, que como se mencionó anteriormente, se encarga de eliminar duplicados en la lista “listaLineasErrores”. Luego, se declara una cadena de caracteres llamada “simbolo”, que se utilizará para almacenar temporalmente el valor de un identificador que se va a eliminar. El código contiene dos bucles anidados, el bucle exterior itera a través de los elementos de “listaLineasErrores”, el bucle interior itera a través de los elementos de “listaIdentificadores”. Dentro del bucle interior, el código compara si el valor entero almacenado en el campo "tipo" (que en este caso es el número de línea del identificador) del objeto, en la posición j de “listaIdentificadores”, es igual al valor en la posición i de “listaLineasErrores”. Si se cumple la condición de igualdad, significa que el identificador actual está en una línea donde hay un error, por lo que no es válido. En este caso, se obtiene el valor del identificador y se almacena en la variable “simbolo”. Por último, se utiliza el método “remove” en el hashMap, que es la tabla de símbolos, para eliminar el identificador correspondiente.

```
public void eliminarIdentificadores(){
    eliminarDuplicados();
    String simbolo="";

    for(int i=0; i<listaLineasErrores.size(); i++){
        for(int j=0; j<listaIdentificadores.size(); j++) {
            if (Integer.parseInt(listaIdentificadores.get(j).getTipo()) == listaLineasErrores.get(i)) {
                simbolo = listaIdentificadores.get(j).getValor();
                hashMap.remove(simbolo);
            }
        }

    }

}
```

- **Funciones auxiliares de errores:**

Primero se mencionarán dos funciones que no tiene que ver con errores, pero son necesarias en el programa.

Contar líneas vacías:

Como se ve en el nombre esta función lo único que hace es contar líneas vacías, ya que nuestro compilador soporta que el usuario deje líneas vacías, esta función es necesaria para que los errores concuerden con la línea en la que están, ya que si no contáramos estas líneas vacías se desfasarían los errores. La función lo que hace es entrar en un bucle "while" que pregunta si el token actual es de tipo "VACIA", estos tokens se agregan en la fase léxica indicando que es una línea vacía, si esto es correcto se aumenta en 1 el número de línea y también se pasa a la siguiente posición en la lista de tokens con "posicionPlus".

```
public void contarLineasVacias(){
    while (listaToken.get(posicion).getTipo().equals("VACIA") && posicion<listaToken.size()-1) {
        numLinea++;
        posicionPlus();
    }
}
```

PosicionPlus:

Lo que hace esta función es básicamente aumentar en 1 la posición de la lista de tokens que estemos analizando, pero siempre revisando que no nos estemos saliendo de esta, para que así no nos de errores de desbordamiento. Funciona preguntando por medio de un "if" si la posición actual no sobre pasa el tamaño de la lista de tokens menos 1, si esto se cumple se puede incrementar la posición.

```
public void posicionPlus(){
    if (posicion < listaToken.size() - 1) {
        posicion++;
    }
}
```

Error de operadores repetidos:

La siguiente función lo que hace es validar si existen uno o muchos operadores seguidos, lo cual sería un error, por lo que cada vez que se encuentra cualquier simbolo de operación como: “+”, “-”, “*”, “/”, se mostrara el mensaje de error correspondiente y como se está dentro de un bucle, esta validación se hará cuantas veces sea necesario. Además de eso el error se agregar a la lista de líneas de errores y se hace un “posicionPlus” para seguir con los tokens.

```
public void errorOperadorRepetido(){
    while(listaToken.get(posicion).getTipo().equals("MULTIPLICACION") || listaToken.get(posicion).getTipo().equals("DIVISION")
            || listaToken.get(posicion).getTipo().equals("SUMA") || listaToken.get(posicion).getTipo().equals("RESTA")){
        listaErrores.add("Error [Fase Sintáctica]: La línea " + numLinea + " contiene un error en su gramática, falta un factor en la expresión");
        listaLineasErrores.add(numLinea);
        posicionPlus();
    }

}
```

Error de paréntesis izquierdos:

La siguiente función lo que hace es validar si en la expresión que se está analizando faltan paréntesis izquierdos, lo que hace la función es ver si el token actual es un paréntesis derecho y además si el contador que se mencionó anteriormente de paréntesis izquierdos está vacío, eso significa que en ningún momento se abrió ese paréntesis por lo que se mostrará el error correspondiente, como también se agregaría a la lista de líneas de errores y se pasaría al siguiente token.

```
public void errorParentesisIzquierdo(){
    while(listaToken.get(posicion).getTipo().equals("PARENTESIS_DER") && contadorParentesisIzquierdo==0){
        listaErrores.add("Error [Fase Sintáctica]: La línea " + numLinea + " contiene un error en su gramática, falta token: (");
        listaLineasErrores.add(numLinea);
        posicionPlus();
    }
}
```

Error si el siguiente token es incorrecto:

La siguiente función, lo que hace primero es validar mediante un condicional si el token actual es un identificador o un número, posteriormente a eso, en un bucle pregunta si el siguiente token es igualmente un token o un número, lo cual si es verdadero es un error debido a que entre un numero y un identificador o viceversa debe de haber un simbolo de operación. Luego se verifica que si lo que se está leyendo actualmente es un paréntesis izquierdo y anteriormente no era un simbolo de operación significa, que hay un error porque antes de cualquier paréntesis izquierdo y cualquier otra expresión debe de haber un simbolo de operación.

```
public void errorSiguienteIncorrecto(){

    if(listaToken.get(posicion).getTipo().equals("NUMERO")|| listaToken.get(posicion).getTipo().equals("IDENTIFICADOR")){
        while(listaToken.get(posicion).getTipo().equals("NUMERO") ||  listaToken.get(posicion).getTipo().equals("IDENTIFICADOR")){
            listaErrores.add("Error [Fase Sintáctica]: La línea " + numLinea + " contiene un error en su gramática, falta simbolo de Operación");
            listaLineasErrores.add(numLinea);
            posicionPlus();
        }
    }
    else if(listaToken.get(posicion).getTipo().equals("PARENTESIS_IZQ")){
        listaErrores.add("Error [Fase Sintáctica]: La línea " + numLinea + " contiene un error en su gramática, falta simbolo de Operación");
        listaLineasErrores.add(numLinea);
        termino();
    }
}
```

3.1.3. Funciones de las producciones del lenguaje.

Como se mencionó antes la gramática del lenguaje L1, no fue transformada por lo que para la implementación de las producciones se representaron exactamente igual en el código, también se agregaron al interior de las producciones los diferentes errores que se pueden producir.

Programa:

Al empezar la función de programa, que es la primera producción, se llama a la función contar líneas vacías, que se menciono anteriormente, para ver si existe alguna línea vacía antes de leer los tokens del programa, luego se pregunta si el token actual es un punto y coma ya que si esto es correcto significa que hay un error, por que un programa no puede empezar con un punto y coma, por lo que se guardaría la línea del error correspondiente y también se pregunta si el siguiente token es “FIN” el cual indica que ahí se acaba una línea, si esto es correcto, se sigue con el siguiente token, se aumenta el contador de línea y también se ve si hay líneas vacías.

```
public void programa(){

    contarLineasVacias();//Con esta funcion nos saltamos las lineas vacias
    //Este error es en el caso de que solo exista un punto coma en la linea es decir este solo
    while(listaToken.get(posicion).getTipo().equals("PUNTO_COMA")){
        listaErrores.add("Error [Fase Sintáctica]: La línea " + numLinea +" contiene un error en su gramática, Expresion incorrecta");
        listaLineasErrores.add(numLinea);
        posicionPlus();

        if(listaToken.get(posicion).getTipo().equals("FIN")){
            posicionPlus();
            numLinea++;
        }
        contarLineasVacias();
    }
```

Ahora se procedería con el flujo de la producción (expresion “;” {programa}) preguntando si el programa empieza con una expresión, esta es otra función de la producción con el mismo nombre que se explicara más adelante, si es correcto se pregunta si el token actual, o sea, el que esta después de la expresión es un punto y coma, así como viene en la producción, si no es un punto y coma y si este token también es uno de “FIN” significa que hay un error por lo que se guardaría el error correspondiente, se añadiría la línea del error a la lista, se aumentaría la línea, se aumentara la posición de la lista de tokens y se llamaría a programa recursivamente. Si lo anterior no ocurrió, se preguntaría si el token actual es un punto y coma, si esto es correcto se aumentará la posición, se preguntaría si el token actual es uno de “FIN”, si esto es así se aumenta la línea, luego se haría otro “posicionPlus” y se llamaría a programa recursivamente como en la producción.

```
    }
    //Llamamos a expresion para ver si es una expresion
    if(expresion()) {
        //Este error es para que si no hay punto y coma nos diga que falta un punto y coma
        if (listaToken.get(posicion).getTipo().equals("FIN") && !(listaToken.get(posicion - 1).getTipo().equals("PUNTO_COMA"))) {
            listaErrores.add("Error [Fase Sintáctica]: La línea " + numLinea +" contiene un error en su gramática, falta token ;");
            listaLineasErrores.add(numLinea);
            numLinea++;
            posicionPlus();
            programa();
        }
        //Si hay punto y coma entonces verificamos que puede volver a haber un programa
        else if (listaToken.get(posicion).getTipo().equals("PUNTO_COMA")) {
            posicionPlus();
            if (listaToken.get(posicion).getTipo().equals("FIN")){
                numLinea++;
            }
            posicionPlus();
            programa();
        }
    }
}
```

Expresión:

La función inicia cuando se declara una variable checkpoint que almacena la posición actual (posición) en el análisis del código. Esto se usa para realizar un "rollback" en caso de que se detecte un error y se necesite retroceder en la posición del análisis. Luego, se verifica una condición para determinar si existe un error en la gramática. La condición se divide en dos partes: La primera parte verifica si la posición actual (posición) es 0 y si el tipo del token en esa posición es "ASIGNACION". Esto se hace para detectar si el código comienza con una asignación sin un identificador previo, y la segunda parte verifica si el tipo del token en la posición actual es "ASIGNACION", y si el tipo del token anterior (posición actual - 1) no es "IDENTIFICADOR". Esto se hace para detectar si hay una asignación sin un identificador antes.

Si se cumple cualquiera de las condiciones anteriores (lo que significa que se encontró un error gramatical), se agregan mensajes de error a la lista listaErrores y el número de la línea a listaLineasErrores. Luego, se llama recursivamente al método expresion para continuar analizando la expresión después del error. Esto se hace para intentar identificar cualquier otro error en la expresión, si no es una expresión, se agregaría el error correspondiente y se guardaría la línea del error y se retornaría true.

```
public boolean expresion(){
    int checkpoint = posicion;

    // No hay identificador?
    if((posicion == 0 && (listaToken.get(posicion).getTipo().equals("ASIGNACION"))) ||
            (listaToken.get(posicion).getTipo().equals("ASIGNACION") && !(listaToken.get(posicion-1).getTipo().equals("IDENTIFICADOR")))){
        listaErrores.add("Error [Fase Sintáctica]: La línea  " + numLinea + " contiene un error en su gramática, falta un identificador");
        listaLineasErrores.add(numLinea);
        posicionPlus();
        if (!expresion()) { //Lo que sigue no es expresion?
            listaErrores.add("Error [Fase Sintáctica]: La línea " + numLinea + " contiene un error en su gramática, falta un factor en la expresión");
            listaLineasErrores.add(numLinea);
        }
        return true;
    }
```

Ahora vamos con la segunda parte del código en la cual se encuentra la primera parte de la producción de expresion (identificador “=” expresion), para esto se restaura la posición del análisis (posicion) a la posición guardada previamente en la variable checkpoint. Esto se hace para aplicar back-tracking en caso de que la primera parte del análisis no tenga éxito. Luego, se verifica si el token en la posición actual es un "IDENTIFICADOR". Si es así, el código procede a realizar más análisis. Si se encuentra un "IDENTIFICADOR", se incrementa la posición actual (posicionPlus) para avanzar al siguiente token. A continuación, se verifica si el siguiente token es una "ASIGNACION". Si es así, el código procede a realizar más análisis. Si se encuentra una "ASIGNACION", se agrega el

"IDENTIFICADOR" encontrado previamente (token en la posición posicion-1) a la lista listaIdentificadores como se mencionó anteriormente, solo estos identificadores se guardan para posteriormente revisar si la tabla de símbolos está correcta. Luego, se incrementa nuevamente la posición actual (posicionPlus) para avanzar al siguiente token. Se llama al método expresion para analizar la expresión que debe seguir después de una asignación. Si expresion devuelve true, esto significa que se ha analizado con éxito una expresión. Si expresion devuelve false, se agrega un mensaje de error a las listas listaErrores y listaLineasErrores. El mensaje indica que falta un factor en la expresión. Si no se encuentra un "IDENTIFICADOR" o una "ASIGNACION" después de restaurar la posición con el checkpoint, el código continúa. La posición del análisis se restaura nuevamente al checkpoint. Se llama la función “expresionTermino”, el cual es la segunda parte de la producción, para validar si lo que se esperaba como una expresión era en realidad un término.

```
    posicion= checkpoint;//Los checkpoint para aplicar backtraking

    //Verificamos si es un identificador y luego si sigue un igual en el caso de que no entonces mas abajo hacemos un checkpoint para volver
    if(listaToken.get(posicion).getTipo().equals("IDENTIFICADOR")){
        posicionPlus();
        if(listaToken.get(posicion).getTipo().equals("ASIGNACION")){
            listaIdentificadores.add(new Token(listaToken.get(posicion-1).getValor(),Integer.toString(numLinea)));
            posicionPlus();

            if(expresion()){//Es una expresion lo que sigue?
                return true;
            }
            else{
                listaErrores.add("Error [Fase Sintáctica]: La línea " + numLinea + " contiene un error en su gramática, falta un factor en la expresión");
                listaLineasErrores.add(numLinea);
                return true;
            }
        }
    }

    posicion= checkpoint;
    //Hacemos checkpoint en el caso de que no sea identificador mas igual
    return expresionTermino();// Llamamos a expresion termino para validar si en realidad expresion era termino

}
```

ExpresionTermino:

La siguiente función es la continuación de la producción de expresion (termino {(+,-) termino}), se separó para mayor facilidad en la comprensión del código y solucionar ciertos errores. Esta función comienza llamando a la función termino. Después de llamar a termino, el código verifica si el token en la posición actual (posicion) es una "ASIGNACION". Si lo es, se agrega un mensaje de error a las listas listaErrores y listaLineasErrores, indicando que hay un símbolo de asignación mal colocado en la línea actual. Luego, se avanza a la siguiente posición con posicionPlus. Si no se encuentra un "ASIGNACION", el código verifica si el token en la posición actual es una "SUMA" o

"RESTA". Si es cualquiera de estos operadores, se avanza a la siguiente posición (posicionPlus) y luego se llama recursivamente a expresionTermino, para analizar el término siguiente en la expresión. Después de analizar el término siguiente (ya sea un término o una expresión) el código vuelve a llamar recursivamente a expresionTermino, sin verificar nuevamente el token en la posición actual. Finalmente, el método expresionTermino devuelve true si se ha analizado con éxito la expresión o el término. Si en algún punto del análisis no se cumple la gramática, se devuelve false.

```
public boolean expresionTermino(){
    if(termino()){//Llamamos a la funcion termino
        if (listaToken.get(posicion).getTipo().equals("ASIGNACION")){//En el caso de que haya un igual en medio que no pudo ser dado por medio de expresion
            listaErrores.add("Error [Fase Sintáctica]: La linea " + numLinea + " contiene un error en su gramática, simbolo de asignacion mal colocado");
            listaLineasErrores.add(numLinea);
            posicionPlus();
        }

        else if(listaToken.get(posicion).getTipo().equals("SUMA") || listaToken.get(posicion).getTipo().equals("RESTA")){
            posicionPlus();
            if(!expresionTermino()){
                return false;
            }
        }
        expresionTermino();
        return true;
    }
    return false;
}
```

Termino:

La siguiente función corresponde a la producción de termino (factor {(+,-) factor}), se comienza llamando a varias funciones que verifican diferentes condiciones antes de analizar un término. Estas funciones ya se mencionaron anteriormente, son: errorOperadorRepetido: Verifica si hay operadores repetidos y agrega un error si se encuentran, errorParentesisIzquierdo: Verifica si falta un paréntesis izquierdo y agrega un error si es necesario. Luego, la función verifica si la función de factor devuelve true. Esto indica que se ha analizado con éxito un factor en la expresión. Después de analizar un factor, el código verifica el tipo del token en la posición actual (posicion), para determinar qué sigue en la expresión. Hay varios casos posibles: Si el token en la posición actual es "NUMERO", "IDENTIFICADOR" o "PARENTESIS_IZQ", significa que hay un error gramatical, ya que debería haber un operador entre dos factores. Se llama a errorSiguienteIncorrecto para informar sobre este error, si el token en la posición actual es "PARENTESIS_DER" y el contador contadorParentesisIzquierdo es igual a 0, significa que falta un paréntesis izquierdo correspondiente. En este caso, se agrega un mensaje de error indicando que falta el token "(" y se avanza a la siguiente posición hasta que se resuelva el desequilibrio de paréntesis. Si el token en la posición actual es "MULTIPLICACION" o "DIVISION", indica que hay una multiplicación o división entre

factores. En este caso, se avanza a la siguiente posición y se llama recursivamente a termino, para analizar el siguiente término. Si solo se ha encontrado un factor la función devuelve true. En caso contrario, devuelve false ya que no se cumple con la producción de termino.

```
public boolean termino(){
    errorOperadorRepetido();//Verificamos si hay un operador repetido para añadir ese error
    errorParentesisIzquierdo();//Verificamos si falta un parentesis izquierdo para añadir ese error

    if(factor()){

        if (listaToken.get(posicion).getTipo().equals("NUMERO")|| listaToken.get(posicion).getTipo().equals("IDENTIFICADOR") || listaToken.get(posicion).getTipo().equals("PARENTESIS_IZQ")) {
            errorSiguienteIncorrecto();//Error en el caso de que haya un factor pero antes de el no haya un simbolo de operación
        }
        //Volvemos a verificar si falta el parentesis izquierdo y ejecutamos el error
        else if (listaToken.get(posicion).getTipo().equals("PARENTESIS_DER") && contadorParentesisIzquierdo==0){
            while(listaToken.get(posicion).getTipo().equals("PARENTESIS_DER") && contadorParentesisIzquierdo==0){
                listaErrores.add("Error [Fase Sintáctica]: La linea " + numLinea + " contiene un error en su gramática, falta token: (");
                listaLineasErrores.add(numLinea);
                posicionPlus();
            }
        }
        //si no hay errores entonces sigue una multiplicacion o division y seguimos con el ciclo normal llamando otra vez a termino
        else if(listaToken.get(posicion).getTipo().equals("MULTIPLICACION") || listaToken.get(posicion).getTipo().equals("DIVISION")){
            posicionPlus();
            return termino();
        }

        return true;
    }

    return false;
}
```

Factor:

Esta función corresponde con la producción de factor (identificador|numero|(expresion)), se comienza con la declaración de una variable checkpoint, que almacena la posición actual (posicion) en el análisis del código. Esto se hace para realizar un "rollback" en caso de que no se cumplan las condiciones y se necesite volver a la posición original. Luego, el código verifica tres casos posibles para determinar si la entrada es un factor: El primer caso verifica si el token en la posición actual (posicion) es un "IDENTIFICADOR". Si es así, se avanza a la siguiente posición (posicionPlus) y se devuelve true. El segundo caso verifica si el token es un "NUMERO". Si es así, se avanza a la siguiente posición y se devuelve true. El tercer caso verifica si el token es un "PARENTESIS_IZQ". Si es así, se avanza a la siguiente posición y se incrementa el contador contadorParentesisIzquierdo. Luego, se llama a expresion para analizar una expresión entre paréntesis. Después de analizar la expresión, el código verifica si el siguiente token es un "PARENTESIS_DER". Si es así, se avanza y se decrementa el contador contadorParentesisIzquierdo. Si no se encuentra un "PARENTESIS_DER", se agrega un error indicando que falta el token “)” y se decrementa el contador. Si no se cumple ninguno de los tres casos anteriores, se devuelve false para indicar que la entrada no es un factor válido. El método maneja los errores de la siguiente manera: Si se encuentra un "IDENTIFICADOR" o un "NUMERO" pero falta un operador

entre dos factores, no se registra un error en este método, ya que se manejará en otro lugar. Si se encuentra un "PARENTESIS_IZQ" pero no se completa una expresión o falta un "PARENTESIS_DER", se agrega un error indicando la falta de un cierre de paréntesis o una expresión incompleta.

```java
public boolean factor(){

    int checkpoint = posicion;
    if(listaToken.get(posicion).getTipo().equals("IDENTIFICADOR")){//Verificamos si es un identificador
        posicionPlus();
        return true;
    }

    posicion = checkpoint;
    if(listaToken.get(posicion).getTipo().equals("NUMERO")){//Verificamos si es un numero
        posicionPlus();
        return true;
    }

    posicion = checkpoint;
    if(listaToken.get(posicion).getTipo().equals("PARENTESIS_IZQ")){//Verificamos si es un parentesis izquierdo luego una expresion y luego un parentesis derecho
        posicionPlus();
        contadorParentesisIzquierdo ++;
        if(expresion()){
            if(listaToken.get(posicion).getTipo().equals("PARENTESIS_DER")){
                posicionPlus();
                contadorParentesisIzquierdo --;
            }
            else{
                listaErrores.add("Error [Fase Sintáctica]: La linea " + numLinea + " contiene un error en su gramática, falta token: )");//Error en el caso de que si sea expresion pero fa
                listaLineasErrores.add(numLinea);
                contadorParentesisIzquierdo --;
            }
            return true;
        }
        listaErrores.add("Error [Fase Sintáctica]: La linea " + numLinea + " contiene un error en su gramática, expresion incompleta");//Error en el caso de que no sea una expresion
        listaLineasErrores.add(numLinea);
        contadorParentesisIzquierdo --;
        return true;
    }

    else{
        return false;
    }

}
```

4. Pruebas funcionales

Prueba 1:

a = 42 ;

b = a / 7 + (40 * zzz) - 33 ;

c = (a + b) * 2 ;

b * a + b / (mivariable) ;

En la anterior prueba se ve como sintáctica y léxicamente esta correcta, según la gramática del lenguaje L, primeramente el analizador empieza por la producción de programa preguntado si lo primero que se encuentra es una expresion, por lo que se va a la función de expresion, preguntando la primera parte de la producción, la cual es (identificador = expresion), primero lee la "a" como un identificador por lo que esta correcto y ahora pregunta si el token que sigue es de un "=", por lo que esta correcto, ahora se vuelve a llamar a expresion y esta vez se va a la segunda parte de la expresion para ver si es un termino, ya que cuando se pregunto si el 42 es un identificador, esto fue erróneo por lo que se hizo un back-tracking para seguir con la otra parte de la producción (termino {(+,-) termino}), por lo que ahora, se va a la función termino, que a su vez se va a la función de factor, la cual determina que efectivamente es un factor ya que el "42" es un número y ya con esto la recursión termina y se devuelve a la función de programa, donde se pregunta si el siguiente token es ";" por lo que es correcto y estaría buena esa primera línea y seguiría preguntando si lo que sigue después del punto y coma es un programa, así continua el programa hasta el final, que efectivamente esta correcto en su totalidad.

Nuestro compilador en este caso no imprimiría nada ya que todo está bien, solo imprimiría los tokens del analizador léxico que solo son ilustrativos. Y además tenemos nuestra tabla de símbolos con todos los identificadores ya que no hubo que quitar ninguno, ya que todo está bien.

```
Token: ), Tipo: PARENTESIS_DER
Token: *, Tipo: MULTIPLICACION
Token: 2, Tipo: NUMERO
Token: ;, Tipo: PUNTO_COMA
Token: $, Tipo: FIN
Token: b, Tipo: IDENTIFICADOR
Token: *, Tipo: MULTIPLICACION
Token: a, Tipo: IDENTIFICADOR
Token: +, Tipo: SUMA
Token: b, Tipo: IDENTIFICADOR
Token: /, Tipo: DIVISION
Token: (, Tipo: PARENTESIS_IZQ
Token: mivariable, Tipo: IDENTIFICADOR
Token: ), Tipo: PARENTESIS_DER
Token: ;, Tipo: PUNTO_COMA
Token: $, Tipo: FIN
```

```
Clave: a, Valor: IDENTIFICADOR
Clave: b, Valor: IDENTIFICADOR
Clave: c, Valor: IDENTIFICADOR
Clave: mivariable, Valor: IDENTIFICADOR
Clave: zzz, Valor: IDENTIFICADOR
```

Prueba 2:

f = (a + b) * c ;

pi = 314 ;

radio / pi * 2 ;

circulo = f + 12;

En este caso la prueba también esta correcta, ahora explicaremos otro caso y como se interpreta en nuestro código, primero entra a programa como siempre, porque es nuestra primera producción, luego pregunta si es una expresion lo que se esta leyendo, luego en expresion se lee un identificador (f) por lo que se cumple la primera condición de la primera parte de la producción, luego pregunta si lo que sigue es un "=" y efectivamente, ahora se pregunta si lo que sigue es una expresion, para esto volvemos a la producción y se pregunta si lo que estamos leyendo es un identificador, en este caso no, por que estamos en el paréntesis izquierdo, por lo que haciendo back-tracking, se ve la segunda parte de la producción y se pregunta si terminó, nos vamos a término y se pregunta si es un factor el token actual, ya en factor nos vamos al tercer caso en el cual coincide nuestro paréntesis izquierdo, por lo que ahora se pregunta si lo que esta después de ese paréntesis es una expresion, ya en expresion se valida la segunda parte de la producción siguiendo los mismos pasos, termino → factor → identificador, nuestro token coincide ya que es una "a" la cual es un identificador, luego volvemos a expresion donde nos preguntamos si el siguiente token es un "+" o "-" en este caso es una suma por lo que ahora se pregunta si lo que estamos leyendo es otro termino por lo que siguiendo la misma lógica llegamos a que si ya que llegamos al identificador "b", ya al validar que lo que esta después del paréntesis es una expresion se pregunta si lo que esta después es un paréntesis derecho que sierre esa expresion, en este caso ese correcto por lo que ahora la recursión volvería a la producción de expresion ya que todo esto era un termino y bajo esta misma lógica se valida que el "*" si esta correcto y el identificador "c" también y finaliza el programa con ";" por lo que esta correcto y así continua hasta el final, al igual que en el anterior caso nuestro compilador no devuelve nada solo los tokens lo que significa que todo esta correcto y la tabla de símbolos.

```
Token: ;, Tipo: PUNTO_COMA
Token: $, Tipo: FIN
Token: circulo, Tipo: IDENTIFICADOR
Token: =, Tipo: ASIGNACION
Token: f, Tipo: IDENTIFICADOR
Token: +, Tipo: SUMA
Token: 12, Tipo: NUMERO
Token: ;, Tipo: PUNTO_COMA
Token: $, Tipo: FIN
```

```
Clave: a, Valor: IDENTIFICADOR
Clave: b, Valor: IDENTIFICADOR
Clave: c, Valor: IDENTIFICADOR
Clave: f, Valor: IDENTIFICADOR
Clave: pi, Valor: IDENTIFICADOR
Clave: circulo, Valor: IDENTIFICADOR
Clave: radio, Valor: IDENTIFICADOR
```

Prueba 3:

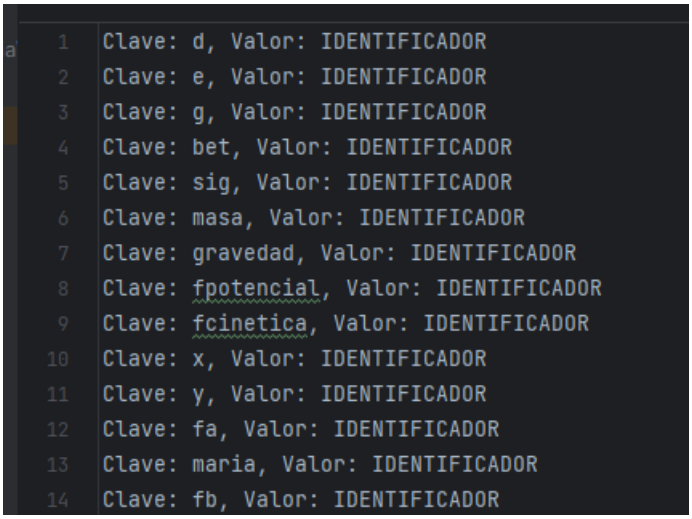
(masa * gravedad) / 2 ;

fa = fb - fcinetica + fpotencial ;

maria = bet * sig ;

g = (d - e) * (x + y) ;

En esta prueba también se encuentra todo correcto según la gramática, vamos a analizar la primera sentencia que es otro caso diferente a los primeros 2, primero se entra a programa como siempre y se evalúa si es una expresion, en expresion se pregunta primero si es un identificador el primer token, en este caso no es así ya que estamos con el paréntesis izquierdo, por lo que medio del back-tracking, se evalúa la segunda parte de la producción donde se pregunta si es termino, ya como vimos anteriormente el código sigue la ruta de termino → factor → (, por lo que se valida que si está correcto, después se evalúa que haya una expresion dentro de los paréntesis, en este caso "masa * gravedad" si cumple con la gramática para ser una expresion y también tiene un paréntesis derecho que cierra la expresion por lo que esta correcto, por ultimo se ve que este paréntesis se divide por un numero lo que cumple con la producción de termino que se buscaba ya que seria " factor (*, /) factor" en este caso escogiendo la división y esto a su vez cumple con expresion por lo que el programa estaría valido, se evalúa que efectivamente se termina con punto y coma. Ya con esto el programa sigue y valida que todo está correcto, otra vez al estar todo correcto el compilador no retorna nada solo los tokens y su respectiva tabla de símbolos.



Prueba 4 (con errores):

x = y * + / - ;

var = y + ;

pal = y (z + w) ;

gas * 43 / 12 ;

a = ((y + z

zzz = y + z)) ;

ejem = y + 1 2 3 4 5 6

x = ;

mivariable = +

En este caso se probarán casos en los que apartir de la gramática una entrada estaria mala, en la primera fila se puede ver cómo hay varios símbolos de operadores pero sin factores en medio, lo que estaria malo según la gramática ya que esta dice que "termino {(+,-) termino}" y "factor {(*,/) factor}" lo que da a entender que cualquier operador tiene que estar en medio de dos factores u términos no pueden seguir el uno del otro. Luego en la segunda fila se ve que le falta igual que en la fila anterior un factor o un termino para completar la expresión, luego se tiene la tercera fila donde se ve que tambien hace falta un símbolo de operación entre "y" y el paréntesis, ya que esto se podría considerar dos términos lo que quiere decir que tiene que existir un operador en medio de ambos, luego tenemos la cuarta fila que esta correcta nuestro analizador va a seguir analizando sin importar si hay fallos, siempre va a llegar al final, para la quinta fila tenemos otro tipo de error ya que en la expresion se están abriendo 2 paréntesis pero nunca se están cerrando y además falta el ";" final, en la sexta tenemos un error parecido ya que el programa encuentra 2 paréntesis derechos, pero estos paréntesis en ningún momento se abrieron por lo que es un error, luego en la séptima fila tenemos el error que tenemos muchos números juntos lo que se podría ver como si hay muchos términos juntos por lo que si nos acordamos la gramática nos dice que siempre tiene que haber un operador entre dos términos por lo que aquí tambien habrían varios errores, en la penúltima fila nos encontramos que después del igual ya se termina nuestra sentencia por lo que aquí tambien habría otro error ya que después del símbolo del igual tiene que venir luego una expresion y en la última fila vemos que después del igual hay un "+" por lo que como en el anterior línea después del símbolo de igual tiene que haber una expresion pero en este caso el símbolo de la suma es una expresion invalida por lo que hay un error, además falta el ";".

Nuestro compilador en efecto nos avisaría que están todos estos errores con su respectiva línea y además la tabla de símbolos actualizada ya que hubo varios errores.

```
Error [Fase Sintáctica]: La línea 1 contiene un error en su gramática, falta un factor en la expresión
Error [Fase Sintáctica]: La línea 1 contiene un error en su gramática, falta un factor en la expresión
Error [Fase Sintáctica]: La línea 1 contiene un error en su gramática, falta un factor en la expresión
Error [Fase Sintáctica]: La línea 1 contiene un error en su gramática, falta un factor en la expresión
Error [Fase Sintáctica]: La línea 2 contiene un error en su gramática, falta un factor en la expresión
Error [Fase Sintáctica]: La línea 3 contiene un error en su gramática, falta simbolo de Operación
Error [Fase Sintáctica]: La línea 5 contiene un error en su gramática, falta token: )
Error [Fase Sintáctica]: La línea 5 contiene un error en su gramática, falta token: )
Error [Fase Sintáctica]: La línea 5 contiene un error en su gramática, falta token ;
Error [Fase Sintáctica]: La línea 6 contiene un error en su gramática, falta token: (
Error [Fase Sintáctica]: La línea 6 contiene un error en su gramática, falta token: (
Error [Fase Sintáctica]: La línea 7 contiene un error en su gramática, falta simbolo de Operación
Error [Fase Sintáctica]: La línea 7 contiene un error en su gramática, falta simbolo de Operación
Error [Fase Sintáctica]: La línea 7 contiene un error en su gramática, falta simbolo de Operación
Error [Fase Sintáctica]: La línea 7 contiene un error en su gramática, falta simbolo de Operación
Error [Fase Sintáctica]: La línea 7 contiene un error en su gramática, falta simbolo de Operación
Error [Fase Sintáctica]: La línea 7 contiene un error en su gramática, falta token ;
Error [Fase Sintáctica]: La línea 8 contiene un error en su gramática, falta un factor en la expresión
Error [Fase Sintáctica]: La línea 9 contiene un error en su gramática, falta un factor en la expresión
Error [Fase Sintáctica]: La línea 9 contiene un error en su gramática, falta un factor en la expresión
Error [Fase Sintáctica]: La línea 9 contiene un error en su gramática, falta token ;
```

Como se ve están todos los errores, los primeros 4 factores de la primera línea, el otro factor que falta en la segunda línea, en la tercera falta un símbolo de operación antes del paréntesis, se salta la cuarta fila que esta correcta, en la quinta línea falan los 2 paréntesis y el “;”, en la sexta también faltan sus 2 respectivos paréntesis, todos los símbolos de operación que faltan en la línea 7, el factor que falta después del igual en la línea 8 y en la línea 9 faltan los dos factores que acompañen al símbolo “+” y el “;”.

```
1  Clave: w, Valor: IDENTIFICADOR
2  Clave: gas, Valor: IDENTIFICADOR
3  Clave: y, Valor: IDENTIFICADOR
4  Clave: z, Valor: IDENTIFICADOR
5
```

En la tabla de símbolos se puede ver que varios identificadores fueron eliminados, ya que había errores en sus respectivas líneas como “x”, “var”, “pal”, “a”, “zzz”, “ejem” y “mivariable”, solo se mantuvieron los que se pueden ver en la tabla de símbolos.