Instituto Tecnológico de Costa Rica

Escuela de computación

Compiladores E Intérpretes GR20

Avance 3 Proyecto#1

Estudiantes:

Brandon Rodríguez Vega - 2021152632

Mariano Oconitrillo Vega- 2021080289

Sigifredo Chacón Balladares – 2021067590

Sub-grupo: 2

Profesor:

Roger Emmanuel Ramirez Segura

Centro Académico de Alajuela

Semestre I

2023

Índice:

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 facilita 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 semántico

```
import java.util.ArrayList;
import java.util.HashMap;

public class faseSemantica {
   private HashMap<String, String> hashMap;
   private ArrayList<Token> listaIdentificadores;
   private ArrayList<Token> listaToken;
   private int numLinea = 1;
   private ArrayList<String> listaErroresSemanticos = new ArrayList<>();

   public faseSemantica(ArrayList<Token> listaIdentificadores, ArrayList<Token>
listaToken,HashMap<String, String> hashMap){
       this.listaIdentificadores = listaIdentificadores;
       this.listaToken = listaToken;
       this.hashMap = hashMap;
   }

   public void analisisSemantico() {
       for (Token token : listaToken) {
           if(token.getTipo().equals("FIN")){
               numLinea++;
           }
           boolean coincide = false;
           if (token.getTipo().equals("IDENTIFICADOR")) {
               for (Token identificador : listaIdentificadores) {
                   if (token.getValor().equals(identificador.getValor())) {
                       coincide = true;
                       break;
                   }
               }

               if (!coincide) {
                   hashMap.remove(token.getValor());
                   listaErroresSemanticos.add("Error [Fase Semantica]: La línea
" + numLinea +" contiene un error en su gramática, no declarado identificador "+
token.getValor());
               }
           }
       }
   }

   public ArrayList<String> getListaErroresSemanticos() {
       return listaErroresSemanticos;
   }

   public void setListaErroresSemanticos(ArrayList<String>
listaErroresSemanticos) {
       this.listaErroresSemanticos = listaErroresSemanticos;
```

```
    }
}
```

En este caso, nuestro grupo no implementó el AST, nuestro programa funciona solamente mediante la clase de faseSemantica, la cual recibe:
ListaIdentificadores:Es la lista que contiene todos los identificadores que están antes de un igual tomandolos como identificadores ya definidos, esta lista se llena en la faseSintactica, cuando verificamos si algo es una expresión guardamos esos identificadores.
ListaToken:Es la lista que devuelve el analisis lexico la cual contiene todos los tokens de la expresión y su respectivo tipo, por ejemplo “Token: ;, Tipo: PUNTO_COMA” .
HashMap: Este hashmap solo contiene los identificadores, no solo los que estén antes del igual sino todos los que existan, este hashMap también se llena en la fase léxica.

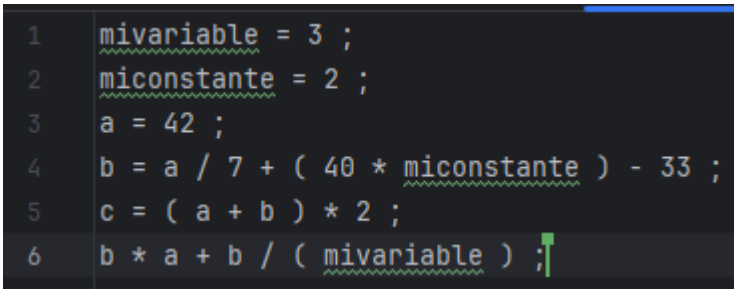
La listaToken y la listaIdentificadores almacenan un dato de tipo TOKEN el cual se compone de un String Tipo y un String Valor los cuales son si es identificador o punto y coma, y el valor correspondiente a ese dato por ejemplo “a” o “;” , respectivamente.

Con estos tres atributos que recibimos, pasamos al método de analisisSemantico(). Primero creamos un for para recorrer todos los token de la listaToken con el objetivo de luego compararlos con otro for para la listaIdentificadores, y así si un token de la listaToken no está en la listaIdentificadores se sabe que este no fue definido anteriormente, luego hacemos un if el cual se encarga de sumar a una variable para saber en qué numero de linea estamos ubicados, definimos un boolean “coincide” y lo inicializamos en false, lo cual nos ayudará para saber cuando un identificador coincide con otro, creamos el for de la listaIdentificadores, y dentro colocamos la interrogante de si el token de la listaToken coincide con alguno de la listaIdentificadores, si es asi, ponemos el coincide en true y hacemos un break para que siga con el otro elemento de la listaToken. Si en algún momento este coincide, queda en false, luego de pasar el for de listaIdentificadores entraría en el if de abajo, el cual removerá del hashmap este identificador que no se definió, luego, hacemos un .add a la lista de errores semánticos del error correspondiente a ese identificador con el número de línea correspondiente.

En el main agregamos el for adecuado para que los errores se agreguen en el archivo de salida.txt.

4. Pruebas funcionales.

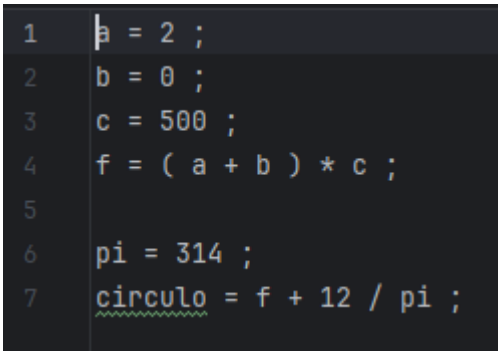
Prueba funcional 1:



```
mivariable = 3 ;
miconstante = 2 ;
a = 42 ;
b = a / 7 + ( 40 * miconstante ) - 33 ;
c = ( a + b ) * 2 ;
b * a + b / ( mivariable ) ;
```

En este ejemplo se crearon diferentes identificadores, los cuales son: “mivariable”, “miconstante”, “a”, “b” (la cual hace uso de “a” y de “miconstante”, pero al estar previamente definidas no nos genera un error) y “c” (la cual hace uso de “a” y de “b”, pero al estar previamente definidas no nos genera un error). Además, por último se creó una expresión que hace uso de varios de estos identificadores, pero al estar previamente definidos tampoco nos generará un error, por lo tanto no se mostrará nada en pantalla a la hora de ejecutar esta prueba.

Prueba funcional 2:

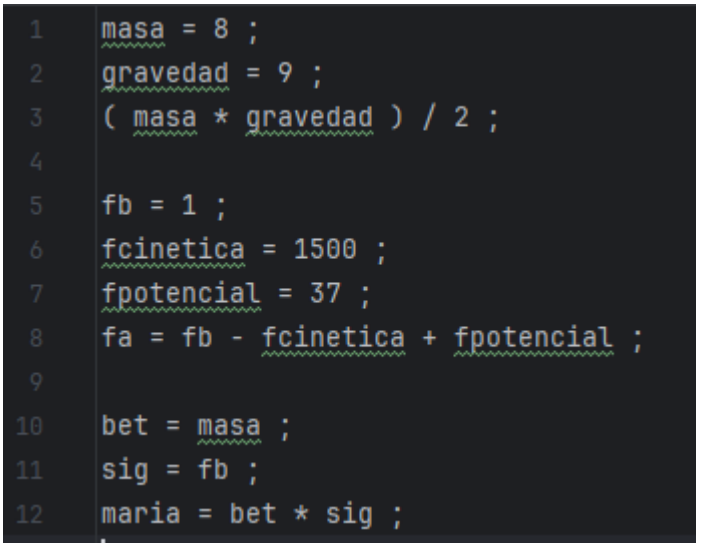


```
a = 2 ;
b = 0 ;
c = 500 ;
f = ( a + b ) * c ;

pi = 314 ;
circulo = f + 12 / pi ;
```

En este ejemplo se crearon diferentes identificadores, los cuales son: “a”, “b”, “c”, “f” (la cual hace uso de “a”, “b” y “c”, pero al estar previamente definidas no nos genera un error) , “pi” y “circulo” (el cual hace uso de “f” y de “pi”, pero al estar previamente definidas no nos genera un error). Dado que todos los identificadores utilizados fueron declarados anteriormente, la prueba se ejecutará con éxito y no se mostrará nada en pantalla.

Prueba funcional 3:



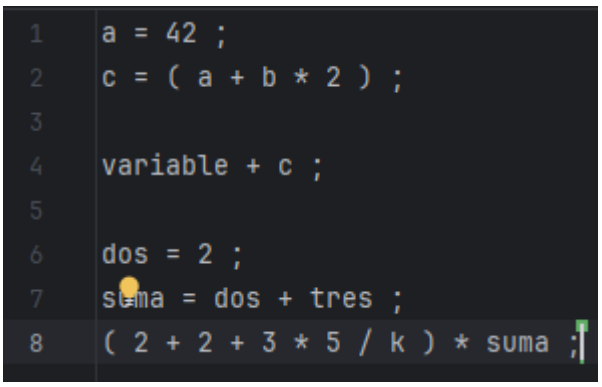
```
masa = 8 ;
gravedad = 9 ;
( masa * gravedad ) / 2 ;

fb = 1 ;
fcinetica = 1500 ;
fpotencial = 37 ;
fa = fb - fcinetica + fpotencial ;

bet = masa ;
sig = fb ;
maria = bet * sig ;
```

En este ejemplo se crearon diferentes identificadores, los cuales son: “masa”, “gravedad”, “fb”, “fcinetica”, “fpotencial”, “fa” (la cual hace uso de “fb”, “fcinetica” y “fpotencial”, pero al estar previamente definidas no nos genera un error), “bet” (el cual hace uso de “masa”, pero al estar previamente definido no nos genera un error), “sig” (el cual hace uso de “fb”, pero al estar previamente definido no nos genera un error), “maria” (el cual hace uso de “bet” y “sig”, pero al estar previamente definidos no nos genera un error). Además, se creó una expresión que hace uso de varios de estos identificadores, pero al estar previamente definidos tampoco nos generará un error, por lo tanto no se mostrará nada en pantalla a la hora de ejecutar esta prueba.

Errores:



```
a = 42 ;
c = ( a + b * 2 ) ;

variable + c ;

dos = 2 ;
suma = dos + tres ;
( 2 + 2 + 3 * 5 / k ) * suma ;
```

Para esta prueba se crearon los identificadores: “a”, “c” (el cual hace uso de “a” y “b”, en este caso deberá mostrar un error ya que “b” no está definido), “dos” y “suma” (el cual hace uso de “dos” y “tres”, en este caso deberá mostrar un error ya que “tres” no está definido). Además, se crearon dos expresiones, la primera (variable + c) deberá mostrar un error, ya que el identificador “variable” no está definido. Y la segunda expresión ((2 + 2 + 3 * 5 / k) * suma) deberá mostrar un error, ya que el identificador “k”no está definido.
Al ejecutar esta prueba, veremos los siguiente errores:

```
Error [Fase Semantica]: La línea 2 contiene un error en su gramática, no declarado identificador b
Error [Fase Semantica]: La línea 3 contiene un error en su gramática, no declarado identificador variable
Error [Fase Semantica]: La línea 5 contiene un error en su gramática, no declarado identificador tres
Error [Fase Semantica]: La línea 6 contiene un error en su gramática, no declarado identificador k
```

5. Experiencia de aprendizaje.

Durante nuestra experiencia en la construcción de este compilador, hemos tenido la oportunidad de poner en práctica y aplicar los conocimientos adquiridos durante el curso a lo largo de este semestre. Desde comprender la gramática libre de contexto en el EBNF hasta realizar la tokenización en la fase léxica utilizando autómatas, cada etapa del proceso ha requerido un esfuerzo significativo. En la fase semántica, aplicamos técnicas como el Backtracking para recorrer la gramática y construir nuestro árbol sintáctico, además de verificar la correcta declaración de identificadores mediante el uso de la tabla de símbolos en la fase semántica.

Si bien el equipo ha enfrentado desafíos significativos en ciertas fases del desarrollo, estas dificultades nos dieron la oportunidad de profundizar en nuestra comprensión y mejorar nuestras habilidades en la implementación de un compilador.Toda esta experiencia ha reforzado la idea de que este proceso requiere una sólida investigación y dedicación para lograr buenos resultados.

6. Bibliografía.

Gajardo, L., & Mateu, L. (2004). Análisis Semántico de programas escritos en Java. Theoria, 13(1), 39-49.

Thain, D. (2020). Introduction to Compilers and Language Design (2da ed., 85-96 del capítulo 6).

Fischer, C. N., Cytron, R. K., & LeBlanc, R. J. (2010). Crafting a compiler. Pearson.