UNIVERSIDAD UTE

FACULTAD DE CIENCIAS DE LA INGENIERIA E INDUSTRIAS

CARRERA COMPUTACIÓN

CUARTO SEMESTRE

LENGUAJES Y COMPILADORES

MANUAL TÉCNICO

REALIZADO POR

STALIN MINDA

MIGUEL MONAR

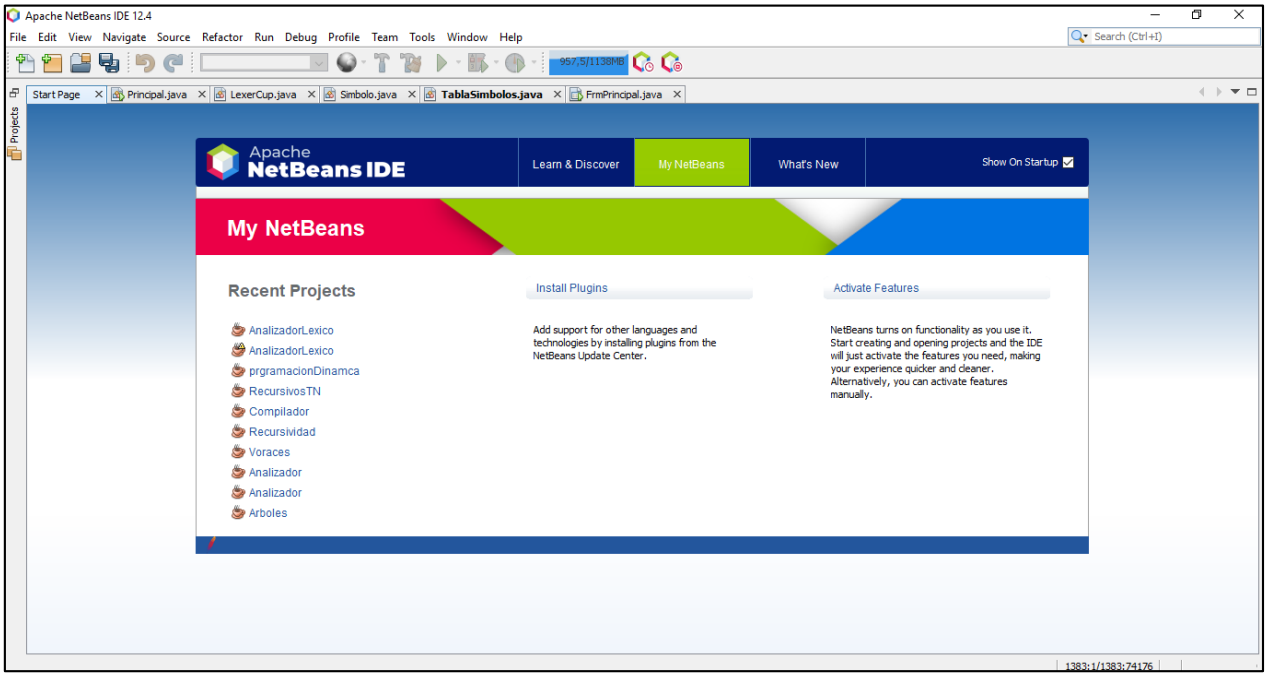
10 de agosto de 2021

Contenido

Requisitos

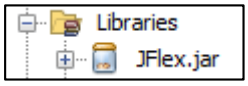
Netbeans

Para el desarrollo del proyecto se utilizó el IDE NetBeans.12.4



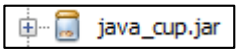
JFlex

Se utilizó la librería JFLex para realizar el analizador sintáctico.



JavaCup

Se utilizaron la librería java_cup para el análisis sintáctico



Clase principal

La clase principal cumple la función de integrar todos los elementos del proyecto y encontrar los archivos y rutas necesarias para generar nuevas clases a partir de las librerías como JFlex y JavaCup.

Por esto, después de cualquier cambio en los archivos de Tokens, gramáticas o lexer, se debe ejecutar de nuevo la clase principal para que el proyecto actualice todos los archivos

que corresponden a estas librerías.

A continuación, se muestra el código de la clase principal

```
public class Principal {
    public static void main(String[] args) throws Exception {
        // C:\Users\USER\Desktop\Final - copia\AnalizadorLexico\src\codigo\Lexer.flex
        String ruta1 = "C:/Users/USER/Desktop/Final - copia/AnalizadorLexico/src/codigo/Lexer.flex";
        String ruta2 = "C:/Users/USER/Desktop/Final - copia/AnalizadorLexico/src/codigo/LexerCup.flex";
        String[] rutaS = {"-parser", "Sintax", "C:/Users/USER/Desktop/Final - copia/AnalizadorLexico/src/codigo/Sintax.cup"};
        generar(ruta1, ruta2, rutaS);
    }
    public static void generar(String ruta1, String ruta2, String[] rutaS) throws IOException, Exception{
        File archivo;
        archivo = new File(ruta1);
        JFlex.Main.generate(archivo);
        archivo = new File(ruta2);
        JFlex.Main.generate(archivo);
        java_cup.Main.main(rutaS);

        Path rutaSym = Paths.get("C:/Users/USER/Desktop/Final - copia/AnalizadorLexico/src/codigo/sym.java");
        if (Files.exists(rutaSym)) {
            Files.delete(rutaSym);
        }
        Files.move(
                Paths.get("C:/Users/USER/Desktop/Final - copia/AnalizadorLexico/sym.java"),
                Paths.get("C:/Users/USER/Desktop/Final - copia/AnalizadorLexico/src/codigo/sym.java")
        );
        Path rutaSin = Paths.get("C:/Users/USER/Desktop/Final - copia/AnalizadorLexico/codigo/Sintax.java");
        if (Files.exists(rutaSin)) {
            Files.delete(rutaSin);
        }
        Files.move(
                Paths.get("C:/Users/USER/Desktop/Final - copia/AnalizadorLexico/Sintax.java"),
                Paths.get("C:/Users/USER/Desktop/Final - copia/AnalizadorLexico/src/codigo/Sintax.java")
        );
    }
}
```

Como se ha mencionado anteriormente, podemos ver en la imagen que esta clase además de identificar las rutas de los analizadores léxico y sintáctico, también genera archivos utilizando métodos como el **JFlex.Main.generate(archivo)** y el **java_cup.Main.main(rutaS).**

Análisis Léxico

El análisis léxico hace uso de la librería JFlex.jar.

Para definir el Análisis léxico es necesario crear un archivo **lexer.flex** que contiene las reglas léxicas.

lexer.flex

Compuesto por las reglas léxicas, se describe de la siguiente forma:

```
package codigo;

import static codigo.Tokens.*;

%%

%class Lexer

%type Tokens

L=[a-zA-Z_]+

D=[0-9]+

espacio=[ ,\t,\r]+

%{

    public String lexeme;

%}

%%

int {lexeme=yytext(); return Int;}

double {lexeme=yytext(); return Double;}

float {lexeme=yytext(); return Float;}

if {lexeme=yytext(); return If;}

else {lexeme=yytext(); return Else;}

while {lexeme=yytext(); return While;}

boolean {lexeme=yytext(); return Boolean;}

break {lexeme=yytext(); return Break;}

default {lexeme=yytext(); return Default;}

do {lexeme=yytext(); return Do;}

switch {lexeme=yytext(); return Switch;}

try {lexeme=yytext(); return Try;}

void {lexeme=yytext(); return Void;}
```

```
char {lexeme=yytext(); return Char;}

String {lexeme=yytext(); return String;}

catch {lexeme=yytext(); return Catch;}

{espacio} {/*Ignore*/}

"//".* {/*Ignore*/}

"\n" {lexeme=yytext(); return Linea;}

"=" {lexeme=yytext(); return Igual;}

"+" {lexeme=yytext(); return Suma;}

"-" {lexeme=yytext(); return Resta;}

"*" {lexeme=yytext(); return Multiplicacion;}

"/" {lexeme=yytext(); return Division;}

"(" {lexeme=yytext(); return Parentesis_a;}

")" {lexeme=yytext(); return Parentesis_c;}

"{" {lexeme=yytext(); return Llave_a;}

"}" {lexeme=yytext(); return Llave_c;}

"\"" {lexeme=yytext(); return Comillas;}

"\'" {lexeme=yytext(); return Comilla;}

"main" {lexeme=yytext(); return Main;}

"," {lexeme=yytext(); return Coma;}

"." {lexeme=yytext(); return Punto;}

";" {lexeme=yytext(); return P_coma;}

{L}({L}|{D})* {lexeme=yytext(); return Identificador;}

("(-"{D}+")")|{D}+ {lexeme=yytext(); return Numero;}

("(-"{D}+")")|{D}+\.("(-"{D}+")")|{D}+ {lexeme=yytext(); return Numero;}

 . {return ERROR;}
```

En este archivo primero se importa la clase enum de **Tokens.** Esta clase se describe mas adelante en su propio apartado.

Luego se le indica que los tokens pueden tener la forma de un identificador o de un numero. Para el identificador se le indica que este puede estar comprendido por letras de la a a la z minúsculas o mayuscaulas concatenadas.

Para los dígitos estos pueden ser del 0 al 9 concatenados.

Tambien se identifican los espacios en blanco, los tabuladores, y los retornos de carro, que también pueden ir concatenados.

Luego se declara públicamente el **String lexeme.** Y a continuación se describe de qué forma serán los lexemas.

```
int {lexeme=yytext(); return Int;}
```

esta línea le indica que se puede encontrar con el lexema “int” y que deberá retornar el token **Int.** Este mismo procedimiento se repite para el resto de palabras reservadas, tipos de datos, corchetes, paréntesis, llaves, comas, punto y coma, puntos y operadores lógicos entre otros lexemas que queremos que el analizador léxico identifique dentro del código.

lexer.java

Es la clase encargada de recorrer el codigo escrito en la IU e ir separándolo en Tokens para después retornar los mismos.

Este archivo es generado por la librería JFlex en base a las reglas léxicas escritas en el lexer.flex.

La parte de esta clase que mas nos interesa analizar es el final, además de ser una de las principales funciones de la misma, es el algoritmo que identifica los lexemas dentro del codigo que anliza y retorna los tokens que corresponden a estos lexemas.

A continuación se muestra solo una fragmento de este codigo, puesto que lo demás tiene una estructura similar únicamente cambiando el *case* que evalua y el Token que retorna dependiendo del lexema que encuentra.

```
switch (zzAction < 0 ? zzAction : ZZ_ACTION[zzAction]) {
```

```
        case 15:
          { lexeme=yytext(); return Comillas;
          }
        case 35: break;
        case 17:
          { lexeme=yytext(); return P_coma;
          }
        case 36: break;
        case 24:
          { lexeme=yytext(); return Void;
          }
        case 37: break;
        case 27:
          { lexeme=yytext(); return Float;
          }
        case 38: break;
        . . .
```

Tokens.java

Esta es una clase de enumeración y únicamente deberá tener listados los mismos tokens que hemos detallado en la clase **lexer.flex.**

```
public enum Tokens {
    Linea,
    Int,
    Double,
    Float,
    If,
```

```
    Else,

    While,

    Boolean,

    Break,

    Default,

    Do,

    Switch,

    Try,

    Void,

    Char,

    String,

    Catch,

    Igual,

    Suma,

    Resta,

    Multiplicacion,

    Division,

    Parentesis_a,

    Parentesis_c,

    Llave_a,

    Llave_c,

    Comillas,

    Comilla,

    Main,

    Coma,

    Punto,

    P_coma,
```

```
    Identificador,

    Numero,

    ERROR


}
```

Análisis Sintáctico

Clase LexerCup.flex

Para crear el analizador sintáctico haremos uso de 2 clases en la primera clase será llamada ***LexerCup.flex*** en la cual retornaremos los símbolos con su respectivo análisis y cadena.

Como sabemos JCup retorna símbolos en vez de tokens por lo cual importamos su símbolo en la segunda línea de código *(import java_cup.runtime.Symbol);*

Haremos uso de 4 instrucciones (%cup %full %line %char) las cuales se encargan de:

- **%cup:** Regresa el análisis
- **%full:** Retorna la cadena
- **%line:** Regresa la línea
- **%char:** Regresa la columna en la que se encuentra

```
import java_cup.runtime.Symbol;
%%
%class LexerCup
%type java_cup.runtime.Symbol
%cup
%full
%line
%char
```

Posterior identificaremos y retornaremos el símbolo con su valor almacenado, incluyendo la línea y columna en la que se encuentra, resultado que serán utilizados

para saber en qué línea y columna está el error.

```
%{
    private Symbol symbol(int type, Object value){  //Identificamos el tipo del símbolo y su valor.
        return new Symbol(type, yyline, yycolumn, value);  }  //Retornamos el tipo y su posición
    private Symbol symbol(int type){
        return new Symbol(type, yyline, yycolumn);  }
%}
```

Posterior retornaremos los símbolos con su respectiva palabra reservada (Int, char..) acompañado de yychar, yyline, yylex. De esta misma manera retornaremos los operadores que vamos a ocupar.

Todo esto podemos verlo en el código que se muestra en el recuadro de abajo.

```
//P. Reservadas
 int {return new Symbol (sym.Int, yychar, yyline, yytext());}
float {return new Symbol (sym.Float, yychar, yyline, yytext());}

//Operadores
 "=" {return new Symbol (sym.Igual, yychar, yyline, yytext());}
"-" {return new Symbol (sym.Resta, yychar, yyline, yytext());}
```

Viendo todo el código unido

```
1   package codigo;
2   import java_cup.runtime.Symbol;
3   %%
4   %class LexerCup
5   %type java_cup.runtime.Symbol
6   %cup
7   %full
8   %line
9   %char
10  L=[a-zA-Z_]+
11  D=[0-9]+
12  espacio=[ ,\t,\r,\n]+
13  %{
14      private Symbol symbol(int type, Object value){
15          return new Symbol(type, yyline, yycolumn, value);
16      }
17      private Symbol symbol(int type){
18          return new Symbol(type, yyline, yycolumn);
19      }
20  %}
21  %%
22  int {return new Symbol (sym.Int, yychar, yyline, yytext());}
23  double {return new Symbol (sym.Double, yychar, yyline, yytext());}
```

Clase Sintax.cup

Después creamos nuestra segunda clase de tipo Empty File la cual se llamará **Sintax.cup** de la misma manera, retornaremos el símbolo. Crearemos un parser code el cual nos permitirá retornar el símbolo que se está analizando.

```
parser code {:
    private Symbol s;
    public void syntax_error(Symbol s){
        this.s = s; }
    public Symbol getS(){
        return this.s; }
:};
```

En esta clase identificaremos los símbolos terminales y no terminales. Los cuales nos ayudaran a crear nuestras reglas sintácticas.

```
terminal Int,Double, Float, If, Else, While, Boolean, Break, Default, Do,
Switch, Try, Void, Char, String,
non terminal INICIO, SENTENCIA, DECLARACION;
```

Seguido, iniciaremos nuestras reglas sintácticas con start with INICIO; La cual nos permitirá iniciar nuestra clase. Esta sentencia tendrá los símbolos terminales y no terminales, viéndolo a si de la siguiente manera:

```
 start with INICIO;
INICIO:: = Int Main Parentesis_a Parentesis_c Llave_a SENTENCIA
Llave_c |
```

Finalmente definiremos la sentencia que tendrán nuestras reglas la cual tendrá el tipo de dato el identificador y el punto y coma.

```
SENTENCIA ::=
        Int Identificador P_coma | Int Identificador Igual Numero P_coma | Int Identificador
        Igual Identificador P_coma;
```

En nuestra clase principal pasaremos las rutas para que el programa pueda generar todos los documentos necesarios.

```
public class Principal {
    public static void main(String[] args) throws Exception {
        String ruta1="C:/Users/aleja/Desktop/UTE Salin/Cuarto Semestre/Lenguajes y copiladores/Proyecto/AnalizadorLexico/s
        String ruta2="C:/Users/aleja/Desktop/UTE Salin/Cuarto Semestre/Lenguajes y copiladores/Proyecto/AnalizadorLexico/s
        String [] rutaS = {"-parser", "Sintax", "C:/Users/aleja/Desktop/UTE Salin/Cuarto Semestre/Lenguajes y copiladores/
        generar(ruta1, ruta2, rutaS);
    }
    public static void generar(String ruta1, String ruta2, String [] rutaS) throws IOException, Exception{
        File archivo;
        archivo = new File(ruta1);
        JFlex.Main.generate(archivo);
        archivo = new File(ruta2);
        JFlex.Main.generate(archivo);
        java_cup.Main.main(rutaS);

        Path rutaSym = Paths.get("C:/Users/aleja/Desktop/UTE Salin/Cuarto Semestre/Lenguajes y copiladores/Proyecto/Analiz
        if (Files.exists(rutaSym)){
            Files.delete(rutaSym);
        }
```

Impresión

Para la imprimir el resultado del análisis sintáctico nos dirigimos a nuestro JframeForm en

el cual insertaremos un nuevo botón y un tex Área. Crearemos un nuevo método de tipo privado en el cual obtendremos la clase del LexerCup que creamos. El resultado lo enviaremos al TxAre haciendo uso del parse el cual analizara la parte sintáctica.

```
private void btnAnalizarSinActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    String ST = txtResultado.getText();
    Sintax s = new Sintax(new codigo.LexerCup(new StringReader(ST)));

    try {
        s.parse();
        txtSintactico.setText("Analisis realizado correctamente");
        txtSintactico.setForeground(new Color(25, 111, 61));

    } catch (Exception ex) {
        Symbol sym = s.getS();
        txtSintactico.setText("Error de sintaxis. Linea: " + (sym.right) + "Columna: " + (sym.left + 1)
        txtSintactico.setForeground(Color.red);

    }
```

Como resultado el análisis sintáctico chequeara que los tipos de datos estén asignados correctamente, dependiendo de la sentencia que hayamos puesto con ello evitara la pérdida de información o los errores semánticos.

Análisis Semántico

El análisis semántico se realizó dentro del método LlenarTablaSimbolos().

De esta forma se puede controlar un correcto llenado de la tabla de símbolos.

Los errores semánticos que se controlan son los siguientes:

- **Error de tipo de dato**
    - Cuando se intenta asignar un valor a una variable y los tipos de dato no coinciden.
- **Error de redefinir la variable.**
    - Cuando se intenta definir una variable que ya se ha definido con anterioridad.
- **Error de definir una variable con un valor que no existe.**
    - Cuando se intenta definir una variable en función de otra, pero esta última no se ha definido todavía.

Esto se controla al ir leyendo el código token por token como si se tratara de un análisis léxico.

A continuación, se explica cómo se realiza el análisis semántico, únicamente con el ejemplo del tipo de dato entero, ya que esta misma idea puede extrapolarse para los demás tipos de datos (String, char, float, double).

Primero se debe identificar si el usuario ha escrito la siguiente sentencia

```
int a = b;
```

si es asi, entonces para que no existan errores semánticos debemos asegurarnos que se cumplan las tres condiciones siguientes:

1. que la variable b haya sido definida con anterioridad, por lo que el símbolo b existe en la tabla de símbolos
2. que el tipo de dato de b, también sea int y;
3. que la variable a no haya sido definida con anterioridad.

Si se cumplen las tres condiciones, entonces basta con buscar el símbolo b en la tabla de símbolos para para poder definir a, en función del valor de b.

```
// guardamos el lexema de la variable, para usarlo como su nombre en la tabla de
// símbolos.
String nombre = lexE.lexeme;
if (Tabla.existe(nombre)) {
    // error de redefinicion de variable
} else {
    // creamos el simbolo con su nombre
    Simbolo simbEntero = new Simbolo(nombre);
```

```
    // insertamos el simbolo en la tabla
    Tabla.insertar(nombre, simbEntero);
    // verificamos si lo que le sigue al identificador es otro
    // identificador e.j. a = b
    String banderaNumero = "Identificador";
    if (tokSigE.toString().equals(banderaNumero)) {
        // verificamos que b exista en la table
        if (Tabla.existe(lexSigE.lexeme)) {
            //b existe, verificamos que sea del tipo int
            if(!Tabla.buscar(lexSigE.lexeme).getTipo().toString().equals("integer")) {
                 // b no es del tipo integer
                 // error de tipo de dato
            } else {
                // b si es del tipo de dato integer
                // buscamos a, y le asignamos el valor de b
            }
        } else {
            // b no existe en la tabla de símbolos
            //  Error de definir una variable con un valor que no existe.
        }
    }
}
```

Este proceso se lleva a cabo de la misma forma para los tipos de dato:

- float
- double
- char
- String

Y la tabla de símbolos no se actualiza cuando estos errores ocurren.

En caso contrario, es decir cuando no hay error semántico, el símbolo se crea, se guarda el símbolo en la tabla de símbolos, y se accede a el en la tabla para actualizar su valor.

Tabla de Símbolos

FrmPrincipal.java

La tabla de símbolos está compuesta por dos clases:

- TablaSimbolos.java
- Simbolo.java.

Pero es en el FrmPrincipal, que se va llenando la tabla de símbolos, pues es aquí en donde tras realizar el análisis léxico, hemos decidido repetir un segundo pseudo análisis léxico, que nos sirve para ir creando símbolos e ir insertándolos en la tabla, aprovechando el lexer que nos permiten analizar tokens.

Esto se realiza en el método **LlenarTablaSimbolos().**

Este algoritmo recorre el código escrito o leído desde un archivo .txt, y va creando los tokens. Ignora todo aquel token que no sea relevante para ser almacenado en la tabla de símbolos, como son las palabras reservadas, corchetes, paréntesis, punto y coma, puntos, comas, igual y operadores matemáticos.

Sin embargo, detiene y afina el análisis cuando se encuentra con los token:

- Identificador
- Int
- Float
- Double
- Char
- String

Ya que es después de estos tokens, que se va a encontrar la definición de variables.

De forma que el llenado de la tabla de símbolos es relevante únicamente en los casos mencionados.

El código es el siguiente:

```
private void LlenarTablaSimbolos() throws IOException{
       int cont = 1;
        TablaSimbolos Tabla= new TablaSimbolos();
        String expr = (String) txtResultado.getText();
        Lexer lexer = new Lexer(new StringReader(expr));
        String resultado = "LINEA N°    " + cont + "\t\tSIMBOLO\n";
        while (true) {
            Tokens token = lexer.yylex();
            if (token == null) {
                return;
            }
            switch (token) {
                case Linea:
                    cont++;
                    resultado += "LINEA N° " + cont + "\n";
                    break;
                case Int:
                     // se obtiene el token identificador
                     // se crea el símbolo
                     // se inserta el símbolo en la tabla
                     // se asigna el valor al símbolo, accediendo primero a el desde
```

```
                    // la tabla
                    break;
                . . .
}
```

TablaSimbolos.java

Esta clase contiene el HashMap que hemos utilizado para implementar la tabla de símbolos.

```
HashMap <String,Simbolo> t;
```

El hashmap usa como Key un String, que será el nombre de la variable.

En cambio, para el Value del hashmap, usara Objetos de tipo Simbolo.

El símbolo es el tipo de dato abstracto que se ha definido para almacenar nombre, valor y tipo de cada variable definida en el código. Su implementación se detalla mas adelante.

Tambien dentro de esta clase hemos definido los métodos:

Constructor

```
public TablaSimbolos() {
    t = new HashMap();
}
```

Insertar

```
```

```
public void insertar(String nombre, Simbolo s) {
    t.put(nombre, s);
}
```

este método inserta en el hashmap <t> el símbolo, insertándolo con la clave nombre; que es el nombre de la variable del símbolo, y valor s, que es el objeto símbolo.

Buscar

```
public Simbolo buscar(String nombre) {
    return  (t.get(nombre));
}
```

Este método obtiene un símbolo del hasmap, buscándolo con su clave “nombre”, que es el nombre de la variable del símbolo. Retorna un objeto de tipo Simbolo.

Existe

```
public boolean existe(String nombre){
    Object simboloBandera = t.get(nombre);
    if (simboloBandera == null)
        return false;
    else
        return true;
}
```

Este método averigua si un símbolo existe dentro de la tabla de símbolos. recibe el nombre del símbolo, que es el nombre de la variable. Si encuentra el símbolo retorna true, y si no retorna false.

setMagnitud

```
public void setMagnitud (String nombre, Object valor){
        // Objeto valor es instancia de Intger, Float o Double
        if (valor instanceof Integer)
        {
           t.get(nombre).setValorEntero((Integer)valor);
           t.get(nombre).setTipo("integer");
        }
        else if (valor instanceof Double)
        {
           t.get(nombre).setValorDouble((Double)valor);
           t.get(nombre).setTipo("double");
        }
        else if (valor instanceof Float)
        {
           t.get(nombre).setValorFloat((Float)valor);
           t.get(nombre).setTipo("float");
        }
        else if (valor instanceof String)
        {
           t.get(nombre).setValorString((String)valor);
           t.get(nombre).setTipo("String");
        }
        else if (valor instanceof Character){
           t.get(nombre).setValorChar((Character)valor);
           t.get(nombre).setTipo("char");
```

```
        }
}
```

Este es sin duda el método más importante dentro de la clase tabla de símbolos, pues es el responsable de asignarle un valor al símbolo, de esta forma el símbolo está casi completo, pues ahora ya tiene un nombre de variable y su valor respectivo que puede ser de tipo int, float, doublé, String o char.

Recibe dos elementos, uno es el nombre de la variable, para poder ubicar al símbolo dentro del hashmap, y otro es un object llamado valor.

Este Object puede ser una instancia de las clases:

- Integer
- Float
- Double
- Character
- String

De esa manera podemos reconocer qué tipo de dato se le debe asignar.

Simbolo.java

La clase símbolo es el TDA que se ha implementado para representar cada variable definida dentro del código que se analiza.

Sus atributos son:

```
String nombre;
Integer valorEntero;
Double valorDouble;
Float valorFloat;
```

```
String valorString;

Character valorChar;

String tipo;
```

nombre → nombre de la variable que se ha defindo.

(valorEntero, valorDouble, valorFloat, valorString, valorChar)→todos serán null, excepto aque valor al cual pertenezca la variable definida en el código, en cuyo caso guardará la información que se le ha asignado a la variable.

tipo → aquí se guarda como String el tipo de dato al que pertenece la variable.

Por ejemplo si en el código se ha definido una variable de tipo int llamada a y de valor 345, entonces estos atriubutos y sus respectivos valores serian.

nombre → a

valorEntero → 345

valorDouble → null

valorFloat → null

valorString → null

valorChar → null

tipo → integer

Para asignar el nombre al símbolo se hace uso de su constructor. Los demás atributos se modifican con sus respectivos setter por defecto.

Constructor de la clase símbolo

```
public Simbolo(String nombre){
    this.nombre = nombre;
}
```

La clase Simbolo también comprende los getter y setter por defecto para cada atributo, por lo que su descripción resulta trivial, aun asi se muestra el código a continuación.

```
public void setValorEntero(Integer valorEntero) {
        this.valorEntero = valorEntero;
    }

    public void setValorDouble(Double valorDouble) {
        this.valorDouble = valorDouble;
    }

    public void setValorFloat(Float valorFloat) {
        this.valorFloat = valorFloat;
    }

    public void setValorString(String valorString) {
        this.valorString = valorString;
    }
```

```
public void setValorChar(Character valorChar) {
    this.valorChar = valorChar;
}

public void setTipo(String tipo) {
    this.tipo = tipo;
}

public String getNombre() {
    return nombre;
}

public Integer getValorEntero() {
    return valorEntero;
}

public Double getValorDouble() {
    return valorDouble;
}

public Float getValorFloat() {
    return valorFloat;
}

public String getValorString() {
    return valorString;
```

```
    }


    public Character getValorChar() {

        return valorChar;

    }


    public String getTipo() {

        return tipo;

    }
```