**Instituto Tecnológico de Costa Rica**

**Escuela de Ingeniería en Computación**

**Proyecto programado #3**

Greivin Mauricio Fernández Brizuela c.2022437510

Daniel Alonso Garbanzo Carvajal c.2022117129 

IC-5701 – Compiladores e Interpretes

21 de enero de 2023

Contenido

[Manual de usuario 3](#_Toc153660203)

[Pruebas de funcionalidad 5](#_Toc153660204)

[Descripción del problema 9](#_Toc153660205)

[Diseño del programa 10](#_Toc153660206)

[Librerías usadas 11](#_Toc153660207)

[Análisis de resultados 12](#_Toc153660208)

# Manual de usuario

1. Construcción inicial del proyecto:
   * Realiza un "build" del proyecto para inicializar todas las librerías necesarias.
   * Asegúrate de que los plugins definidos en el archivo pom.xml de la carpeta programa estén correctamente configurados. Esto es crucial para una depuración adecuada.
2. Estructura del proyecto:
   * El proyecto se compone de tres paquetes principales: Analizadores, Generadores y Programa.
   * Los archivos jflex y cup en el paquete Analizadores son esenciales para generar los archivos .java correspondientes.
3. Generación de código:
   * Ejecuta los archivos GeneradorLexer y GeneradorParser para generar el código necesario.
   * Se debe tener en cuenta los siguientes detalles durante este proceso:
4. Modificaciones en App.java:
   * + En la carpeta Programa, dentro del archivo App.java, encontrarás algunos errores.
     + Comenta los imports import Analizadores.Lexer e import Analizadores.sym.
     + También comenta las líneas de la 32 al 62 y de la 80 a la 88.

b. Ejecución de GeneradorParser:

* + - Después de ejecutar GeneradorParser y generar Parser.java, comenta las líneas del 100 al 112.
    - Esto es necesario porque el Lexer aún no ha sido creado.

c. Ejecución de GeneradorLexer:

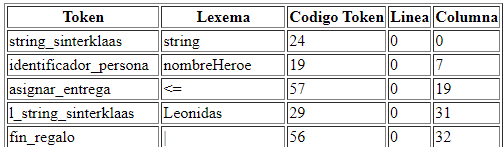
* + - Se debe comentar cualquier linea en Parser.java, donde exista algun error.
    - Una vez que GeneradorParser haya finalizado y se hayan comentado las lineas, ejecuta GeneradorLexer.

1. Finalización y ejecución:
   * Descomenta todas las líneas que previamente comentaste.
   * Ejecuta el archivo App.java del paquete Programa para utilizar la aplicación.
2. Resultado de la ejecución:
   * Al correr la aplicación, se debe crear un archivo .html.
   * Este archivo contendrá un análisis de todas las líneas del archivo test.txt, mostrando el token, lexema, código del token, línea y columna.

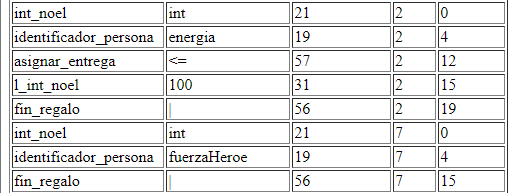
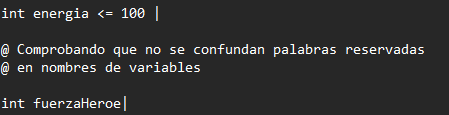
# Pruebas de funcionalidad

En esta parte se valida el uso de un string, un identificador, asignador, literal y comentario, en la tabla se muestra su respectivo código, línea y columna.

string nombreHeroe <= "Leonidas"| @ Inicialización de cadena



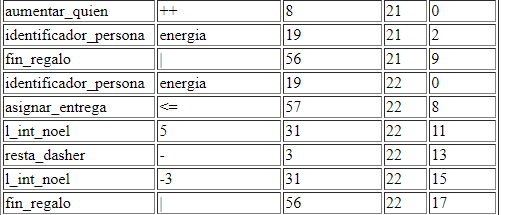
Se valida el uso de un integer, identificador, asignador y literal entero, además de un intero sin su literal.



Se valida la operación unaria de la variable energía anteriormente definida y seguidamente se le asigna un nuevo valor.

++energia|

energia <= 5 - -3|



Se valida el uso de los floats, se le asignan valores distintos para validar los negativos.

float agilidad <= 20.5 - 3.1 + -5|

float resistencia <= 2.3|

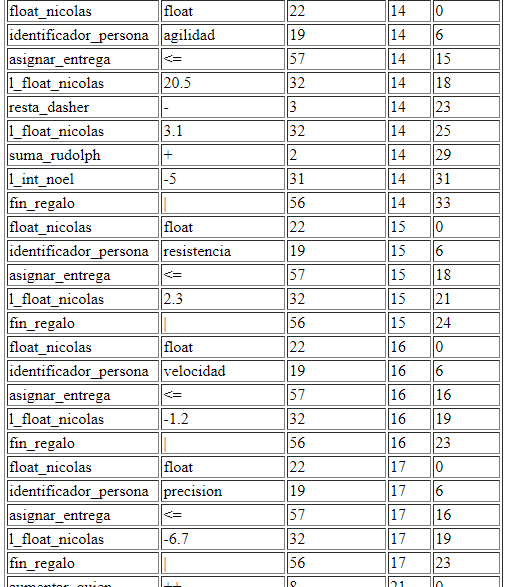
float velocidad <= -1.2|

float precision <= -6.7|

/\_

Comentario multilinea sobre las estadísticas

\_/



Se valida el uso de do y del until, además de la operación unaria negativa y el uso del print.

do {

--energia|

print("Energía actual: ", energia)|

} until (energia == 0)

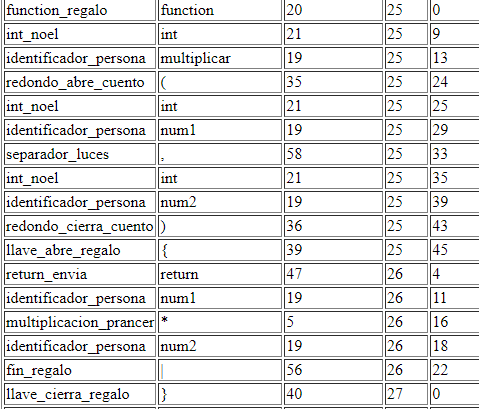


@ Definiendo nuevas funciones

function int multiplicar(int num1, int num2) {

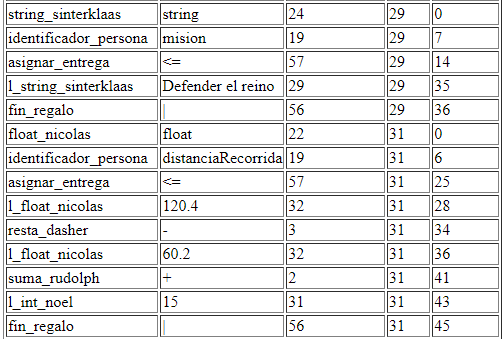
return num1 \* num2|

}



string mision <= "Defender el reino"| @ Definiendo el objetivo

float distanciaRecorrida <= 120.4 - 60.2 + 15|

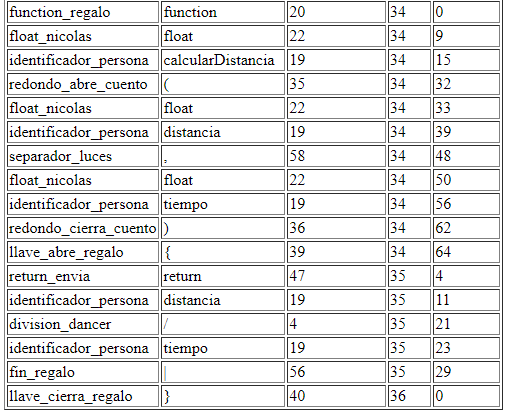


@ Función para cálculo de distancias

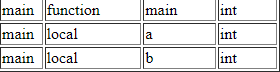
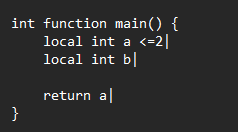
function float calcularDistancia(float distancia, float tiempo) {

return distancia / tiempo|

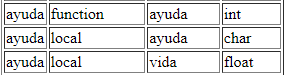
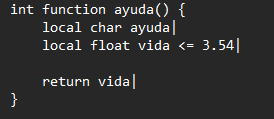
}



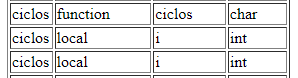
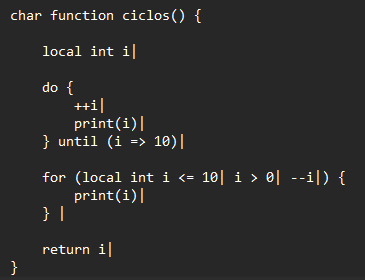
En esta parte se valida el uso de la función main que retorna un int, también se valida el uso de las variables locales en int, tanto con asignación y declaración.



En esta parte se valida el uso de una función que retorna int, también se valida el uso de las variables local char y float con asignación.



En esta parte se valida el uso de una función que retorna un char, también se utiliza una declaración de un local int, también se hace el uso de ciclos do-until y for, en estos se incluye el uso de las variables local int para el uso de estos ciclos



A screenshot of a computer code

Description automatically generated

Ejemplo de corrida con siguiente código

/\_ esto es un comentario inicial \_/

float function func1(char x22,char x22){@semantico borrar parametro

local char x22 <= '!'| /\_semantico x22\_/

local char miChar<='!'|

@ Restringir el largo en semantico, para que char largo = 1

local char miChar2<='!'| @sintactico

local string str1<="Hola $%&/#$&) mundo"|

local float fl1|

local float fl1<=56.6| @semantico fl1

local int in1<= false|

local char arr[2] <= {'c','d'}| @sintatico lit

@arr[2] <= "c"|

local boolean mibool <= true|

return -5.6|@cambio en retorno genera semantico

}

float function miFunc(int x, float z){

for(i <= 10| i<30\*2| ++i|)

{

do{

var <= var - 1| @semantico

return var|@sintactico

}until(var2>12.2 # (34+33)>12)| @semantico

if (var==0)

{

print("entra al if")|

break|

}

elif(true)

{

print(var)|

}

else

{

for(j <= 10| j<30\*2| ++j|){

for(k <= 10| k<30\*2| ++k|){

local int navidad|

return 1+3|

}|

}|

}|

if (var==0){

break|

} elif(true) {

print(var)|

}|

}|

return 1| @semantico

}

string function func3(string b1){ @semantico string

return b1|

}

int function main(){ @function

/\_

Comentario 1

\_/

@comentario 2

local boolean b1|

local string s1|

read(s1)| @semantico

print(b1)| @semantico

print(true)| @semantico

print(-6.7)|

miFunc(miFunc(),"a")| @semantico miFunc, hola

local boolean bl0 <= 6.7 != 8.9| @ok

local boolean bl1 <= in1 => fl1 ^ false # !(func2(3,in1) > 56)| @semantico in1 >= fl1, func2

return bl1| @semantico

}

# Descripción del problema

El avance del proyecto a diseñar es la fase de Análisis Léxico para gramática. Algunos aspectos relevantes del problema son los siguientes: leer un archivo fuente (.txt), escribir en un archivo todos los tokens encontrados (.html), identificador asociado con el lexema y debe de reportar, manejar los errores léxicos y sintácticos encontrados, por cada token se debe indicar en cuál tabla de símbolos va, además cual información almacenará e indicar si el archivo puede ser generado por la grámatica.

La herramienta que se tiene que usar es JFlex para realizar el análisis léxico junto con la librería java-cup. Se deben crear las definiciones regulares correspondientes para identificar los diferentes tokens con la siguiente información: tipo de token, lexema, línea de aparición y columna, también para la tabla de símbolos se debe de tomar en cuenta la siguiente información: tabla, tipo de entrada, tipo de dato y nombre.

Adicionalmente, se debe realizar el analisis semantico, para verificar que las funciones existan, el retorno sea el correcto, las variables son correctas, las expresiones se hagan de manera adecuada y que todos los elementos del código existan y funcionen de manera correcta. También, se debe de realizar el código ensamblador para la ejecución de lo escrito por el programador.

# Diseño del programa

La solución propuesta para el programa es la siguiente:

Se contará con 3 paquetes diferentes, los cuales son: Analizadores, Generadores y Programa. Analizadores cuenta con los archivos .jflex y .cup, adicionalmente en este paquete se crearán los archivos .java para las clases sym, Parser (java-cup) y Lexer (JFlex), también se utiliza una clase SymbolObject para facilitar la creación de la tabla de símbolos. Generadores cuenta con 2 archivos .java, los cuales se encargan de generar las diferentes clases para el funcionamiento correcto del programa. Por último, Programa cuenta con la clase App.java, para correr el programa (leer archivo fuente, guardar resultados en archivo .html, guardar tabla de símbolos en archivo .html y mostrar errores de tokens en consola), adicionalmente, se cuenta con la clase HTMLTable.java para generar los archivos .html.

El archivo lexer.jflex cuenta con todas las definiciones regulares para identificar los diferentes tokens. Al mismo tiempo, el archivo parser.cup cuenta con todos los nombres de los diferentes terminales y no terminales, también cuenta con las producciones necesarias.

Por último, al ser un proyecto creado en Maven todas las instalaciones de las dependencias se almacenan en un archivo pom.xml. De esta manera, se lleva un control exacto de las versiones de las librerías que se están utilizando y el compilar es más amigable.

# Librerías usadas

Las librerías que se utilizaron fueron JFlex para realizar el análisis léxico, adicionalmente se utilizó java-cup para hacer el parsing del archivo y trabajar junto con JFlex. Las librerías mencionadas anteriormente se instalaron con Maven, por lo tanto, se cuenta con un pom.xml para mantener un control de las diferentes dependencias y simplificar la compilación al usuario.

# Análisis de resultados



1. Permitir la creación de funciones, y dentro de ellas, estructuras de control, bloques de código ({}) y sentencias de código.



1. Manejar los tipos de variables enteras, flotantes, booleanas, caracteres, cadenas de caracteres (string) y arreglo estático.



1. Se permite crear arreglos de tipo entero o char. Además, se permite obtener y modificar sus elementos, y ser utilizados en expresiones.



1. Permitir sentencias para creación de variables, creación y asignación de expresiones y asignación de expresiones a variables, y algunos casos, sólo expresiones sin asignación. La asignación se hará por medio de “<=”.



1. Las expresiones permiten combinar literales, variables y/o funciones, de los tipos reconocidos en la gramática.



1. Debe permitir operadores y operandos, respetando precedencia (usual matemática) y permitiendo el uso de paréntesis.



1. Permitir expresiones aritméticas binarias de suma (+), resta (-), división (/) –entera o decimal según el tipo--, multiplicación (\*), módulo (~) y potencia (\*\*). Para enteros o flotantes.



1. Permitir expresiones aritméticas unarias de negativo (-), ++, -- , antes del operando (preorden); esto para enteros, el negativo adicionalmente se puede aplicar a flotantes. El negativo a literales y el ++ y -- a variables.



1. Permitir expresiones relacionales (sobre enteros y flotantes) de menor, menor o igual, mayor, mayor o igual, igual y diferente. Los operadores igual y diferente permiten adicionalmente tipo booleano. Sólo permiten un operando y 2 operadores.



1. Permitir expresiones lógicas de conjunción (^), disyunción (#) y negación (esta debe ser de tipo caracter (!)).



1. Debe permitir sentencias de código para las diferentes expresiones mencionadas anteriormente y su combinación, el delimitador de final de expresión será el carácter pipe (|). Además, dichas expresiones pueden usarse en las condicionales y bloques de las siguientes estructuras de control.



1. Debe permitir el uso de tipos y la combinación de expresiones aritméticas (binarias y unarias), relacionales y lógicas, según las reglas gramaticales, aritméticas, relacionales y lógicas del Paradigma Imperativo, por ejemplo, tomando como referencia el lenguaje C.



1. Debe permitir las estructuras de control if-[elif]-[else], do-until y for, además, permitir return y break. Las expresiones de las condiciones deberán ser valores booleanos combinando expresiones aritméticas, lógicas y relacionales.



1. Debe permitir las funciones de leer (enteros y flotantes) y escribir en la salida estándar (cadena carácter, enteros y flotantes), se pueden escribir literales o variables, se lee a identificadores.



1. Debe permitir la creación y utilización de funciones, definición clásica, estos deben retornar valores (entero, flotantes, char o booleanos) y recibir parámetros (con tipo).



1. Debe existir un único procedimiento inicial main, por medio de la cual se inicia la ejecución de los programas.



1. Además, debe permitir comentarios de una línea (@) o múltiples líneas (/\_ \_/).



Consideraciones especiales de sintaxis:

* La creación de funciones deberá ir precedida por la palabra reserva function.



* La creación de variables deberá ir precedida por la palabra reserva local.



Consideraciones especiales de gramática:

* El símbolo inicial se llamará navidad.



* Todas las producciones tendrán un nombre alusivo a la navidad, por ejemplo:
  + La producción para funciones podría llamarse bolsanavideña.



* + La producción para expresiones podría llamarse regalo.



* + La producción para expresiones aritméticas podría llamarse regaloprin.



* + La producción para expresiones relacionales podría llamarse regalocomprado.



* + La producción para expresiones lógicas podría llamarse regalomanual.

