# RustLex: Desarrollo de un Analizador léxico, sintáctico y semántico basado en Rust

## Introducción

Rust es un lenguaje de programación que empodera a los desarrolladores para escribir código más seguro y eficiente en una variedad de dominios, desde el trabajo a nivel de sistema hasta aplicaciones web y proyectos para dispositivos como Raspberry Pi. Elimina los riesgos tradicionales asociados con la programación a bajo nivel, ofreciendo herramientas amigables y confiables, y está diseñado para guiar a los programadores hacia un código confiable y eficiente. Además, Rust es lo suficientemente expresivo y ergonómico para ser agradable en la creación de aplicaciones de línea de comandos y servidores web, permitiendo a los programadores mejorar sus habilidades y confianza en general (Matsakis, N. & Turon, A.)

Rust se puede caracterizar en 3 criterios de evaluación de un lenguaje de programación:

* Simplicidad: A pesar de ser un lenguaje poderoso y flexible, Rust mantiene una simplicidad en su diseño que facilita la comprensión y el uso del lenguaje, incluso para desarrolladores principiantes.
* Diseño de sintaxis: Rust posee una sintaxis moderna y clara, lo que contribuye a la legibilidad y mantenibilidad del código. La sintaxis está diseñada para minimizar errores y mejorar la productividad del desarrollador.
* Expresividad: Rust permite a los desarrolladores expresar ideas complejas de manera clara y concisa. Su capacidad para soportar paradigmas de programación funcional y de sistemas hace que sea un lenguaje altamente versátil y expresivo.

## Objetivo

Implementar un analizador léxico, sintáctico y semántico basado en el lenguaje de programación Rust, con el propósito de proporcionar una herramienta para el procesamiento y análisis de código fuente.

## Analizador léxico

### Descripción general

Nuestro analizador léxico en esencia debe transformar una cadena de caracteres de entrada en una secuencia de tokens. Esto se logrará definiendo los tipos de tokens (como palabras claves, identificadores, literales, operadores y símbolos de puntuación), implementando mecanismos para avanzar y recoger caracteres de la entrada, y funciones para identificar y clasificar los segmentos de entrada en tokens. Además, debe manejar los errores léxicos.

### Componentes principales

De acuerdo con Matsakis N. y Turon A. en la documentación de este lenguaje de programación podemos encontrar los siguientes componentes:

#### Variables

Las variables en Rust pueden contener letras, dígitos y guión bajo. Deben empezar con letras o guión bajo (“\_”), y siguen la propiedad de Case Sensitive, por lo que se diferenciarán variables escritas en mayúsculas o minúsculas.

#### Tipos de datos

Para la implementación se usarán los siguientes tipos de datos:

1. Enteros

En Rust, se tratan de valores numéricos enteros sin punto ni decimales.

1. Punto flotante

Valores decimales, en Rust, la separación de decimales es con el punto (“.”).

1. Booleanos

En Rust, el tipo de dato bool, toma los valores de: true, false. En minúsculas.

#### Operadores

1. Operadores aritméticos

Para realizar operaciones entre números:

|  |  |
| --- | --- |
| Operador | Descripción |
| + | Adición |
| - | Substracción |
| \* | Multiplicación |
| / | División |
| % | Módulo |

1. Operadores relacionales

Definir la relación entre dos valores:

|  |  |
| --- | --- |
| Operador | Descripción |
| > | Mayor que |
| < | Menor que |
| >= | Mayor o igual que |
| <= | Menor o igual que |
| == | Igualdad |
| != | Diferente de |

1. Operadores lógicos

Para combinar dos o más condiciones:

|  |  |
| --- | --- |
| Operador | Descripción |
| && | AND |
| || | OR |
| ! | NOT |

1. De asignación

Rust utiliza el símbolo = para asignar valores a las variables.

#### Palabras reservadas

Rust define las palabras reservadas como *keywords*, de las cuales se usarán:

1. Strict keywords

Solo se pueden usar con su contexto correcto, no pueden usarse en otro, como variables.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| as | break | const | continue | crate | else |
| enum | extern | false | fn | for | if |
| impl | in | let | loop | match | mod |
| move | mut | pub | ref | return | self |
| Self | static | struct | super | trait | true |
| type | unsafe | use | where | while | async |
| await | dyn |  |  |  |  |

1. Reserved keywords

|  |  |  |  |
| --- | --- | --- | --- |
| Io | Print | Println | Readline |
| Std | stdin |  |  |

#### Delimitadores

Existen diversos delimitadores dentro de Rust.

1. Llaves {}: Delimita bloques de código como funciones, bucles y estructuras de control.
2. Paréntesis (): Agrupa expresiones y para pasar parámetros a funciones.
3. Corchetes []: Delimita arreglos.
4. Punto y coma ;: Finaliza la declaración de variables.

## Analizador Sintáctico

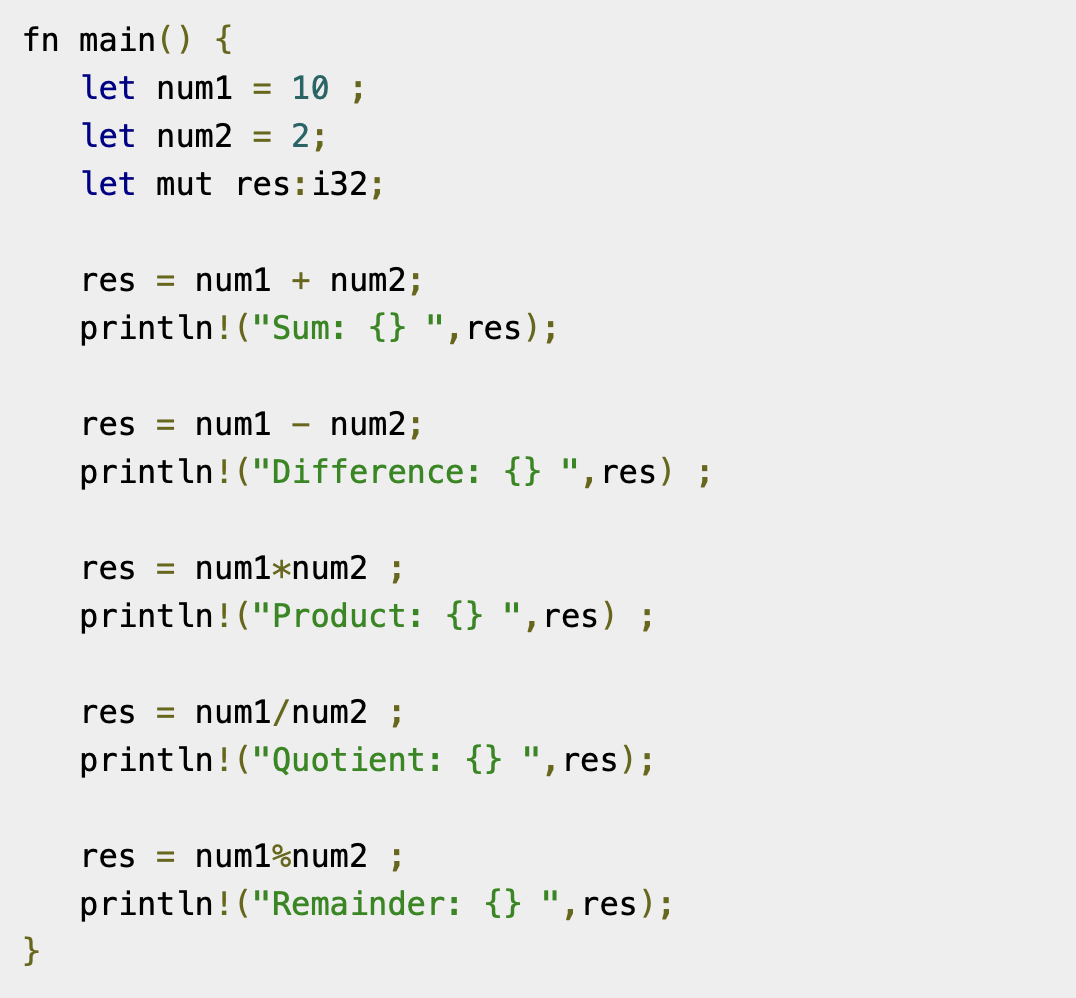
### Descripción general

El analizar sintáctico que se desarrollará tendrá como objetivo interpretar y validar que las expresiones escritas en Rust sigan su correcta estructura. Este componente del compilador recibirá la secuencia de tokens generados por el analizador léxico y verificará que las expresiones tengan la estructura definida en las reglas gramaticales.

### Reglas gramaticales

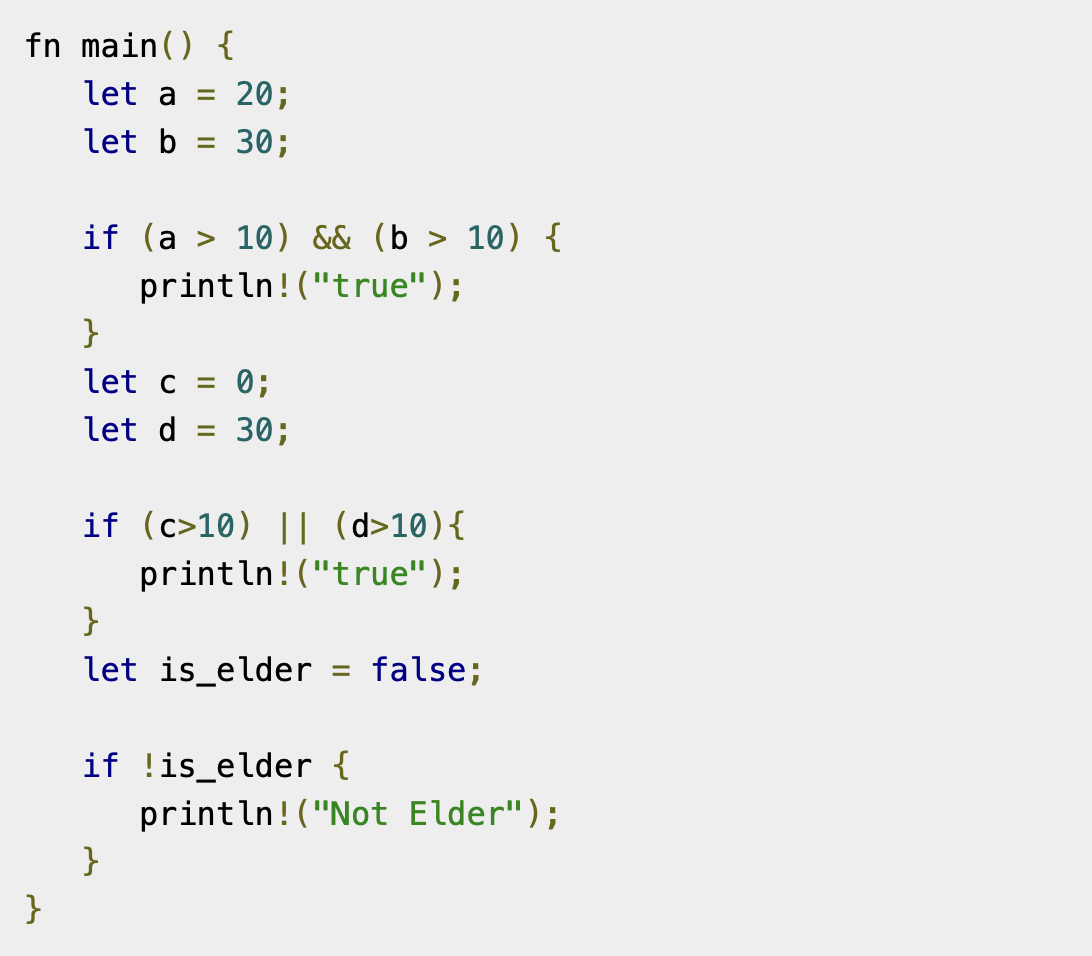
#### Expresiones aritméticas

Para las expresiones aritméticas, debe haber más de un término separado por un operador aritmético. Los términos pueden ser valores, o una expresión entre paréntesis que albergue otra expresión aritmética.



#### Expresiones booleanas

Debe incluir términos separados por un operador relacional.



#### Estructuras de datos (ED)

De manera primitiva, Rust, solo cuenta con dos estructuras de datos:

1. Tupla: Utiliza paréntesis y almacena los datos separados por comas. Puede contener datos de distinto tipo.

A number on a black background

Description automatically generated

1. Array: Todos los elementos deben ser del mismo tipo. Se utilizan corchetes y datos separados por coma.

A number on a black background

Description automatically generated

#### Estructuras de control

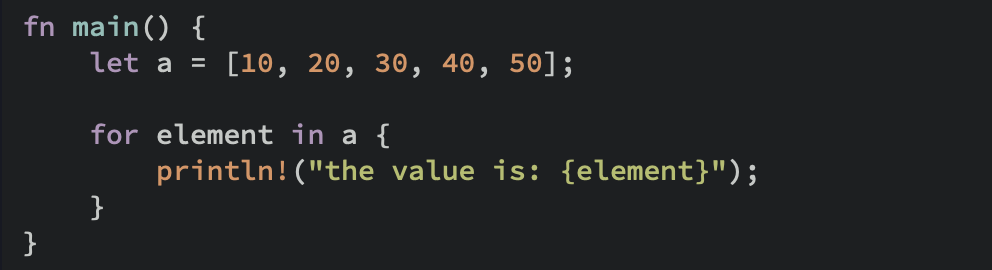
1. While

Este bucle se ejecuta mientras la condición sea verdadera, se utiliza la palabra reservada while junto a la expresión booleana y llaves para definir el bloque de código a repetir.



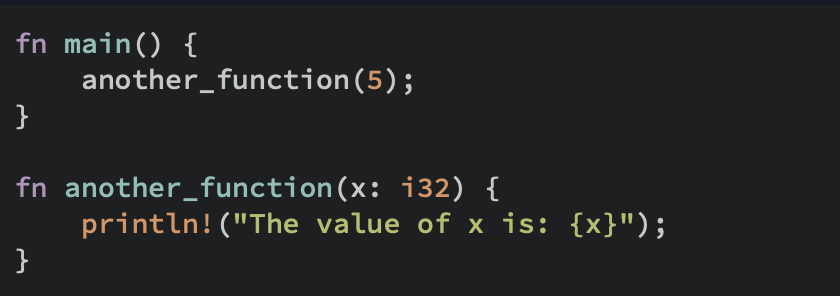
1. For

En Rust, la estructura de control for sirve para iterar dentro de elementos en una lista. Se utiliza la palabra reservada for, junto a la variable que se utilizará para la iteración, palabra reservada in, y el nombre de la colección a iterar.



#### Declaraciones de funciones

En Rust, para declarar una función se debe hacer uso de la palabra reservada fn. Seguido del nombre de la función junto a una pareja de paréntesis que albergará los parámetros, los cuales deben estar junto al tipo de dato. Las reglas para nombrar funciones son similares a las reglas para nombrar variables.



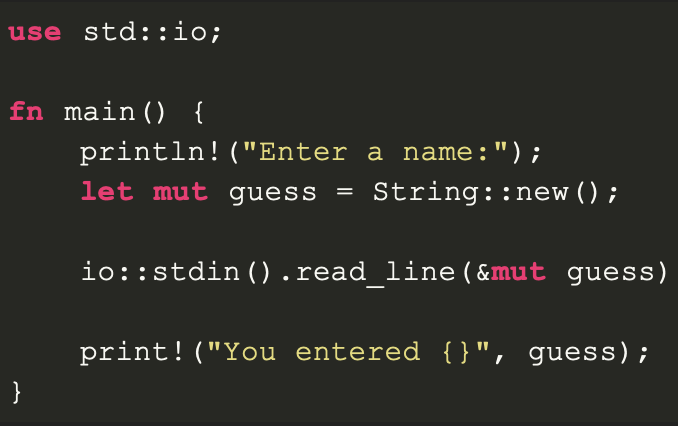
#### Impresión y solicitud de datos

Para la impresión de datos en Rust existen dos macros (funciones):

1. print!(“cadena”): Muestra por consola lo ingresado en el paréntesis dentro de comillas dobles. Sin carácter de finalización de línea.
2. println!(“cadena”): Muestra por consola y finaliza con un salto de línea.

Para imprimir variables dentro de cualquiera de las macros se puede realizar de la siguiente manera:

Para el ingreso de datos:



## Analizador semántico

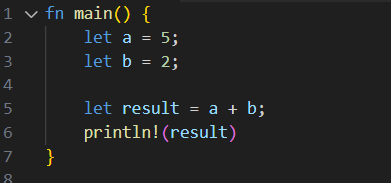
### Descripción general

El analizador semántico tendrá la responsabilidad de verificar que el código fuente tenga sentido dentro del contexto del lenguaje y que cumpla con las reglas semánticas del lenguaje. Las tareas que incluirá son verificación de tipos, chequeo de control del flujo, chequeo de declaraciones y definiciones y verificación de propiedades específicas del lenguaje como lo es la mutabilidad.

### Reglas semánticas

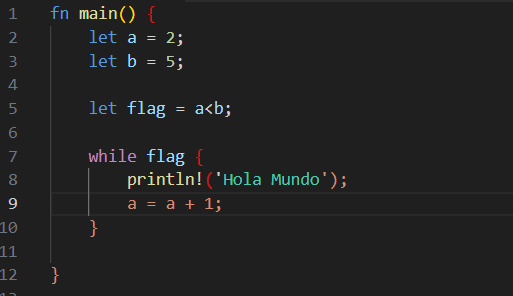
#### Verificación de existencia de variables declaradas

Antes de usar una variable en el código debe verificarse que dicha variable se encuentre declarada, por lo tanto, se debe hacer una revisión de las variables que ya han sido declaradas. Caso contrario, el código no sabría cómo manejar ese dato sin declarar.



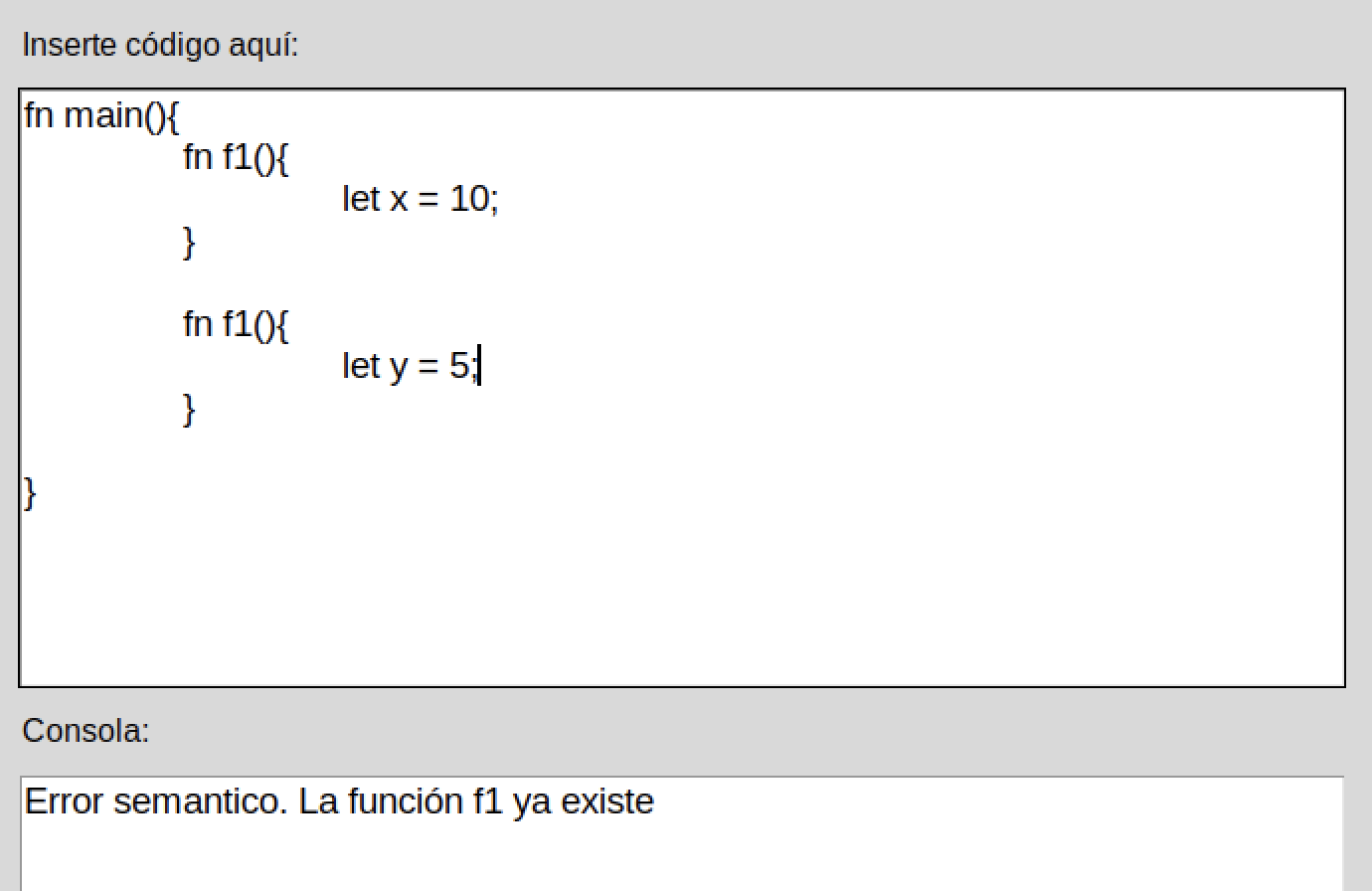
#### Uso correcto de las condiciones

Al momento de utilizar las condiciones ya sea dentro de una estructura de control while o en una expresión booleana, se debe tomar en cuenta que el valor que arroje la condición sea de tipo booleano, de otra manera, no se trataría de un valor booleano.



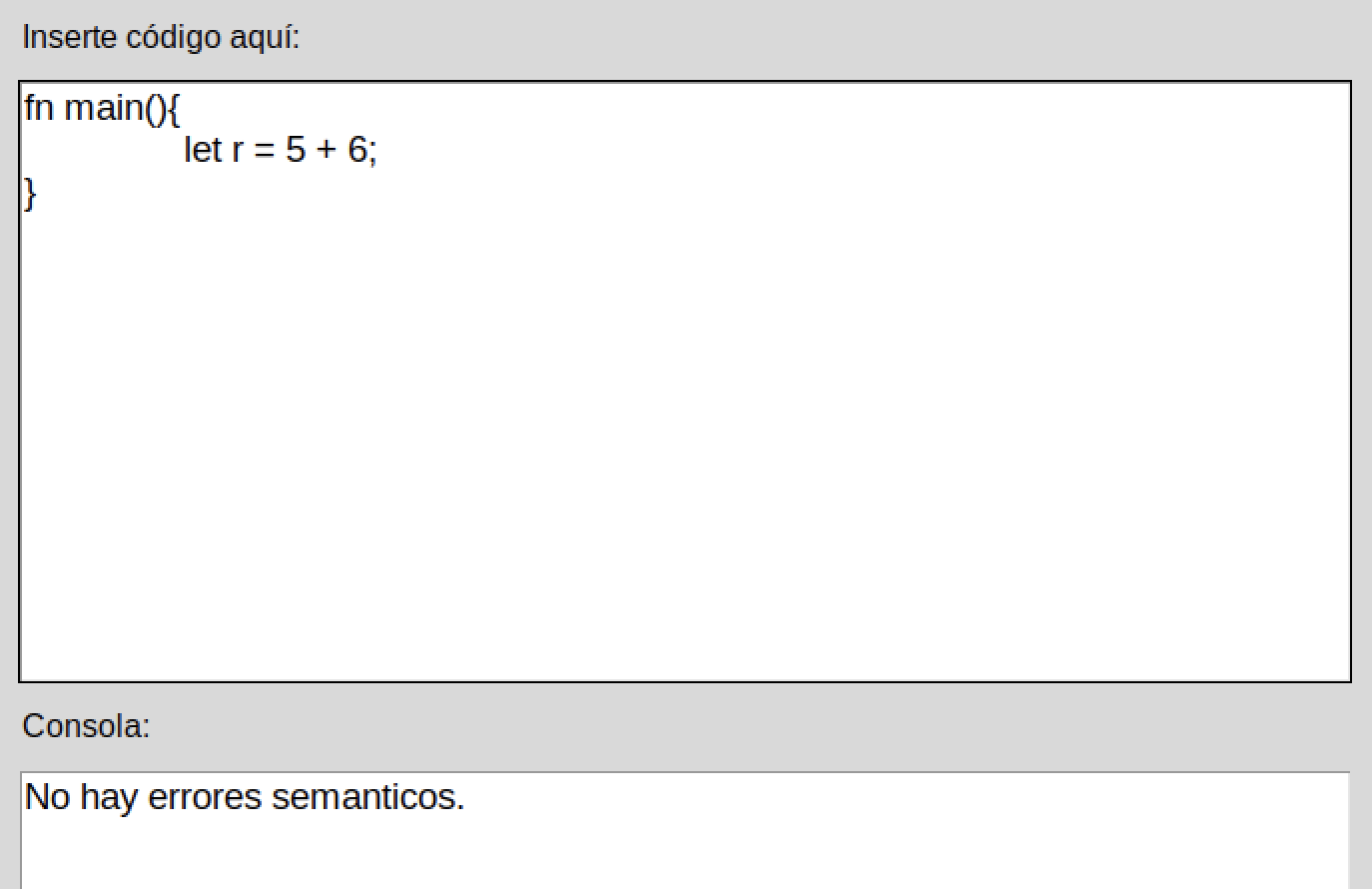
#### Revisión de funciones repetidas

En Rust, al igual que en otros lenguajes de programación, no se puede repetir el nombre que define a una función. En el analizador semántico se analizará si al momento de definir una función el identificador no se encuentra usado al momento.



#### Operaciones aritméticas únicamente entre valores numéricos

Al momento de realizar operaciones aritméticas, es necesario que se realice dicha operación entre valores numéricos. Es decir, si el primer valor es entero o flotante solo se puede operar con otro valor entero o flotante.



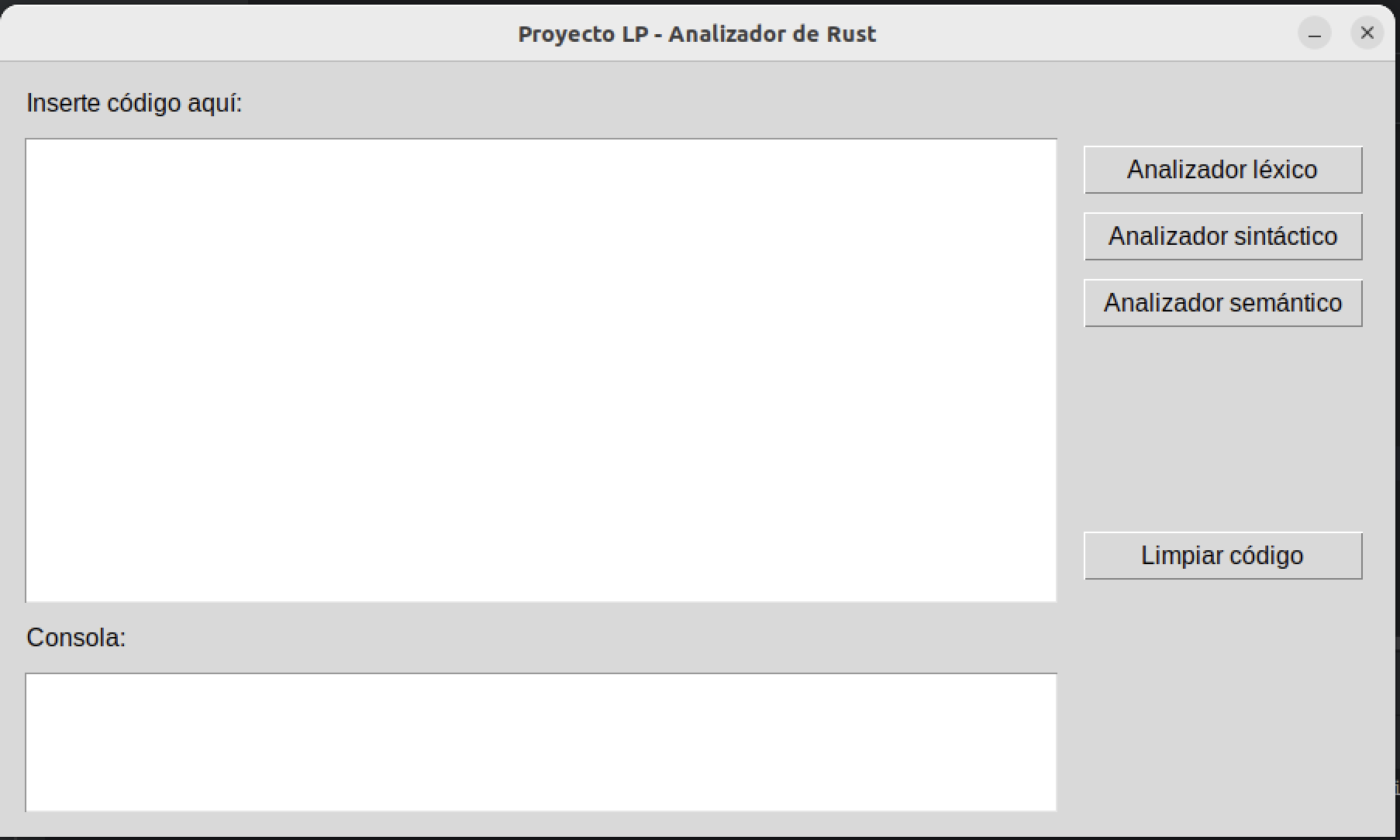
#### Reglas generales

1. Cada programa en Rust debe tener al menos la función main(), que es la que se ejecutará al momento de correr el programa.
2. Todas las variables por usar deben estar declaradas.

## Diseño general

La interfaz gráfica deberá contar con:

* Un editor de código, es decir, un área de texto donde se pueda escribir o pegar el código fuente que se desea analizar.
* Un botón para iniciar el proceso de análisis léxico, sintáctico y semántico respectivamente. Además, incorporar un botón para limpiar el editor de código.
* Una consola de salida, donde se muestre información relevante sobre el proceso de análisis tales como errores y mensajes de logs.



## Algoritmo

El siguiente algoritmo fue usado para probar la implementación:

fn main(){

let x = 1;

let y = 10;

let suma = x + y;

let lista = [1,3,5,7];

let mult = 0;

for impar in lista{

println!(impar);

}

fn funcion1(){

let tuplaPares = (2,4,6,8);

let indice = 0;

while indice < 5{

println!(par);

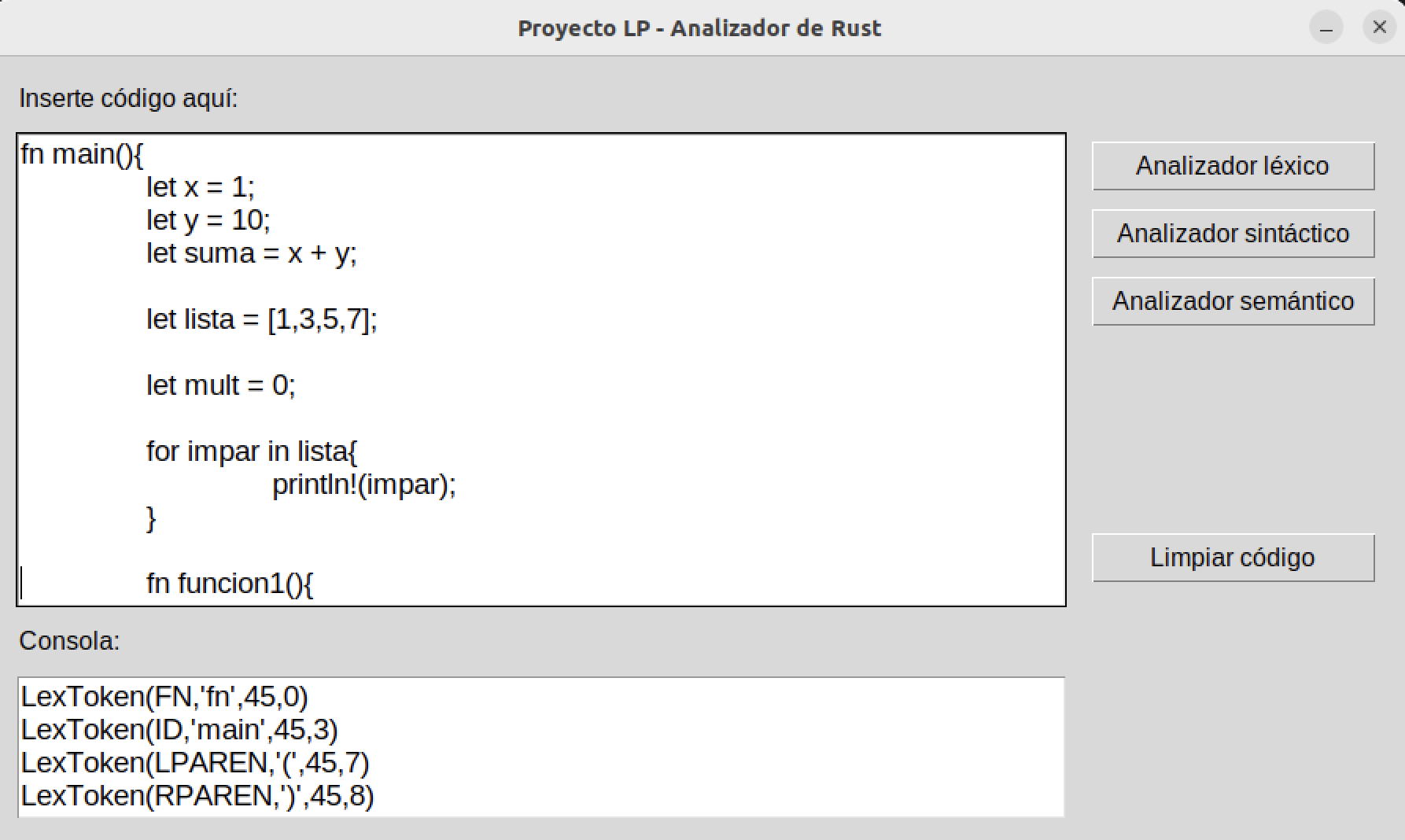
indice +=1;

}

}

}

### Salida del analizador léxico



*(En Anexos se encuentra la lista completa de tokens de la salida de este algoritmo)*

### Salida del analizador sintáctico

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

### Salida del analizador semántico

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

## Conclusiones

* La sintaxis de Rust es clara y precisa, diseñada para facilitar la escritura como la lectura del código. Esto se logra a través de convenciones estrictas y consistentes que reducen la ambigüedad y mejoran la comprensibilidad del código.
* La ejecución del algoritmo de prueba mostró que la implementación de este lenguaje basado en Rust es confiable para detectar errores léxicos, sintácticos y semánticos.
* La experiencia de desarrollo de este lenguaje basado en Rust es altamente positiva. La documentación oficial de Rust es extensa y bien organizada lo que facilita el aprendizaje y la resolución de problemas. Aunque la curva de aprendizaje inicial puede ser empinada debido a conceptos nuevos, una vez superada esa fase se vuelve gratificante.

## Referencias

Matsakis, N., & Turon, A. (n.d.). The rust programming language. *Foreword - The Rust Programming Language.* <https://doc.rust-lang.org/book/foreword.html>

Rust Team (s.f.) The rust programming language. *Data Types- The Rust Programming Language*. Recuperado el 10 de junio de 2024: [Data Types - The Rust Programming Language (rust-lang.org)](https://doc.rust-lang.org/book/ch03-02-data-types.html)

Rust Team (s.f.). The rust programming language. *Control Flow - The Rust Programming Language*. Recuperado el 10 de junio de 2024: [Control Flow - The Rust Programming Language (rust-lang.org)](https://doc.rust-lang.org/book/ch03-05-control-flow.html)

Tutorials Point (s.f.). Rust - arithmetic operators. Recuperado el 10 de junio de 2024: <https://www.tutorialspoint.com/rust/rust_arithmetic_operators.htm>

Tutorials Point (s.f.). Rust – logical operators. Recuperado el 10 de junio de 2024: <https://www.tutorialspoint.com/rust/rust_logical_operators.htm>

Rust Team (s.f.). The rust reference. Keywords – The Rust Reference. Recuperado el 10 de junio de 2024: <https://doc.rust-lang.org/reference/keywords.html>

Rust Team (s.f.). The rust programming language. Control flow – *The Rust Programming Language*. Recuperado el 10 de junio de 2024: <https://doc.rust-lang.org/book/ch03-05-control-flow.html>

Rust Team (s.f.). The rust programming language. Functions – *The Rust Programming Language*. Recuperado el 10 de junio de 2024: <https://doc.rust-lang.org/book/ch03-03-how-functions-work.html>

Programiz (s.f.). Programiz. Rust Print Output (With Examples). Recuperado el 10 de junio de 2024: <https://www.programiz.com/rust/print-output>

GeeksForGeeks (2021). Geeks for Geeks. Standard I/O in Rust – Geeks for Geeks. Recuperado el 10 de junio de 2024: <https://www.geeksforgeeks.org/standard-i-o-in-rust/>

Rust Team (s.f.). Rust by Example. Functions – *Rust by Example*. Recuperado el 10 de junio de 2024: <https://doc.rust-lang.org/rust-by-example/fn.html>

## Anexos

### Listado de tokens

LexToken(FN,'fn',45,0)

LexToken(ID,'main',45,3)

LexToken(LPAREN,'(',45,7)

LexToken(RPAREN,')',45,8)

LexToken(LCURLYBRACKET,'{',45,9)

LexToken(LET,'let',46,12)

LexToken(ID,'x',46,16)

LexToken(ASSIGN,'=',46,18)

LexToken(INTEGER,1,46,20)

LexToken(SEMICOLON,';',46,21)

LexToken(LET,'let',47,24)

LexToken(ID,'y',47,28)

LexToken(ASSIGN,'=',47,30)

LexToken(INTEGER,10,47,32)

LexToken(SEMICOLON,';',47,34)

LexToken(LET,'let',48,37)

LexToken(ID,'suma',48,41)

LexToken(ASSIGN,'=',48,46)

LexToken(ID,'x',48,48)

LexToken(PLUS,'+',48,50)

LexToken(ID,'y',48,52)

LexToken(SEMICOLON,';',48,53)

LexToken(LET,'let',50,57)

LexToken(ID,'lista',50,61)

LexToken(ASSIGN,'=',50,67)

LexToken(LBRACKET,'[',50,69)

LexToken(INTEGER,1,50,70)

LexToken(COMMA,',',50,71)

LexToken(INTEGER,3,50,72)

LexToken(COMMA,',',50,73)

LexToken(INTEGER,5,50,74)

LexToken(COMMA,',',50,75)

LexToken(INTEGER,7,50,76)

LexToken(RBRACKET,']',50,77)

LexToken(SEMICOLON,';',50,78)

LexToken(LET,'let',52,82)

LexToken(ID,'mult',52,86)

LexToken(ASSIGN,'=',52,91)

LexToken(INTEGER,0,52,93)

LexToken(SEMICOLON,';',52,94)

LexToken(FOR,'for',54,99)

LexToken(ID,'impar',54,103)

LexToken(IN,'in',54,109)

LexToken(ID,'lista',54,112)

LexToken(LCURLYBRACKET,'{',54,117)

LexToken(PRINTLN,'println',55,121)

LexToken(NOT,'!',55,128)

LexToken(LPAREN,'(',55,129)

LexToken(ID,'impar',55,130)

LexToken(RPAREN,')',55,135)

LexToken(SEMICOLON,';',55,136)

LexToken(RCURLYBRACKET,'}',56,139)

LexToken(FN,'fn',58,143)

LexToken(ID,'funcion1',58,146)

LexToken(LPAREN,'(',58,154)

LexToken(RPAREN,')',58,155)

LexToken(LCURLYBRACKET,'{',58,156)

LexToken(LET,'let',59,160)

LexToken(ID,'tuplaPares',59,164)

LexToken(ASSIGN,'=',59,175)

LexToken(LPAREN,'(',59,177)

LexToken(INTEGER,2,59,178)

LexToken(COMMA,',',59,179)

LexToken(INTEGER,4,59,180)

LexToken(COMMA,',',59,181)

LexToken(INTEGER,6,59,182)

LexToken(COMMA,',',59,183)

LexToken(INTEGER,8,59,184)

LexToken(RPAREN,')',59,185)

LexToken(SEMICOLON,';',59,186)

LexToken(LET,'let',60,190)

LexToken(ID,'indice',60,194)

LexToken(ASSIGN,'=',60,201)

LexToken(INTEGER,0,60,203)

LexToken(SEMICOLON,';',60,204)

LexToken(WHILE,'while',61,208)

LexToken(ID,'indice',61,214)

LexToken(LESSTHAN,'<',61,221)

LexToken(INTEGER,5,61,223)

LexToken(LCURLYBRACKET,'{',61,224)

LexToken(PRINTLN,'println',62,229)

LexToken(NOT,'!',62,236)

LexToken(LPAREN,'(',62,237)

LexToken(ID,'par',62,238)

LexToken(RPAREN,')',62,241)

LexToken(SEMICOLON,';',62,242)

LexToken(ID,'indice',63,247)

LexToken(PLUS,'+',63,254)

LexToken(ASSIGN,'=',63,255)

LexToken(INTEGER,1,63,256)

LexToken(SEMICOLON,';',63,257)

LexToken(RCURLYBRACKET,'}',64,261)

LexToken(RCURLYBRACKET,'}',65,264)

LexToken(RCURLYBRACKET,'}',66,266)