Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas
Organización de Lenguajes y Compiladores 2
Primer Semestre 2022
Catedráticos: Ing. Bayron López, Ing. Edgar Sabán e Ing. Luis Espino
Tutores académicos: Jhonatan López, Alex Jerónimo y Pablo Roca

DB-Rust

Primer proyecto de laboratorio

1. Competencias

1.1. Competencia general

Que los estudiantes apliquen los conocimientos adquiridos en el curso para la construcción de un intérprete utilizando las herramientas establecidas.

1.2. Competencias específicas

- Que los estudiantes utilicen la herramienta de ANTLR para el análisis léxico y sintáctico.
- Que los estudiantes apliquen sus conocimientos adquiridos en clase magistral y laboratorio de compiladores para desarrollar un intérprete de Rust.
- Que los estudiantes realicen análisis semántico e interpretación del lenguaje DB-Rust

2. Descripción

2.1. Descripción General

Rust es un lenguaje de programación que está tomando fuerza en los últimos años, este cuenta con muchas características que lo hacen un lenguaje muy completo y de gran uso para el desarrollo de servidores backend. Rust se caracteriza por el uso de módulos lo que permite guardar bloques de códigos con una lógica específica.

La idea del proyecto es que se desarrolle un simulador de bases de datos a través de los módulos que ofrece Rust, donde un módulo padre representa una base de datos y sus módulos hijos representan las tablas. Dentro de estas tablas existirán funciones para toda la interacción de los datos, ya sea iniciar la tabla, insertar un dato, obtener un dato por un atributo en específico, etc. No todas las tablas tendrán las mismas funciones, estas pueden cambiar para tener diferentes interacciones con los datos.

2.2. Flujo específico de la aplicación

El proyecto le permitirá a los desarrolladores realizar distintas acciones. Estas acciones seguirán el siguiente flujo:

1. El desarrollador colocará el código fuente en el entorno de desarrollo.
2. El analizador deberá hacer en su primera lectura el guardado de todas las bases de datos (módulos) con sus respectivas tablas y propiedades de cada una.
3. Luego de guardar las bases de datos el analizador deberá guardar las variables globales las cuales se podrán usar para el flujo del código.
4. El analizador deberá guardar las funciones que haya encontrado y ejecutar lo que encuentre en la función “main” del código.
5. Por último el analizador deberá mostrar la salida esperada en consola y los diferentes reportes que se solicitan (tabla de errores, tabla de símbolos, bases de datos existentes, tablas de las bases de datos).

Diagrama del flujo de la aplicación:

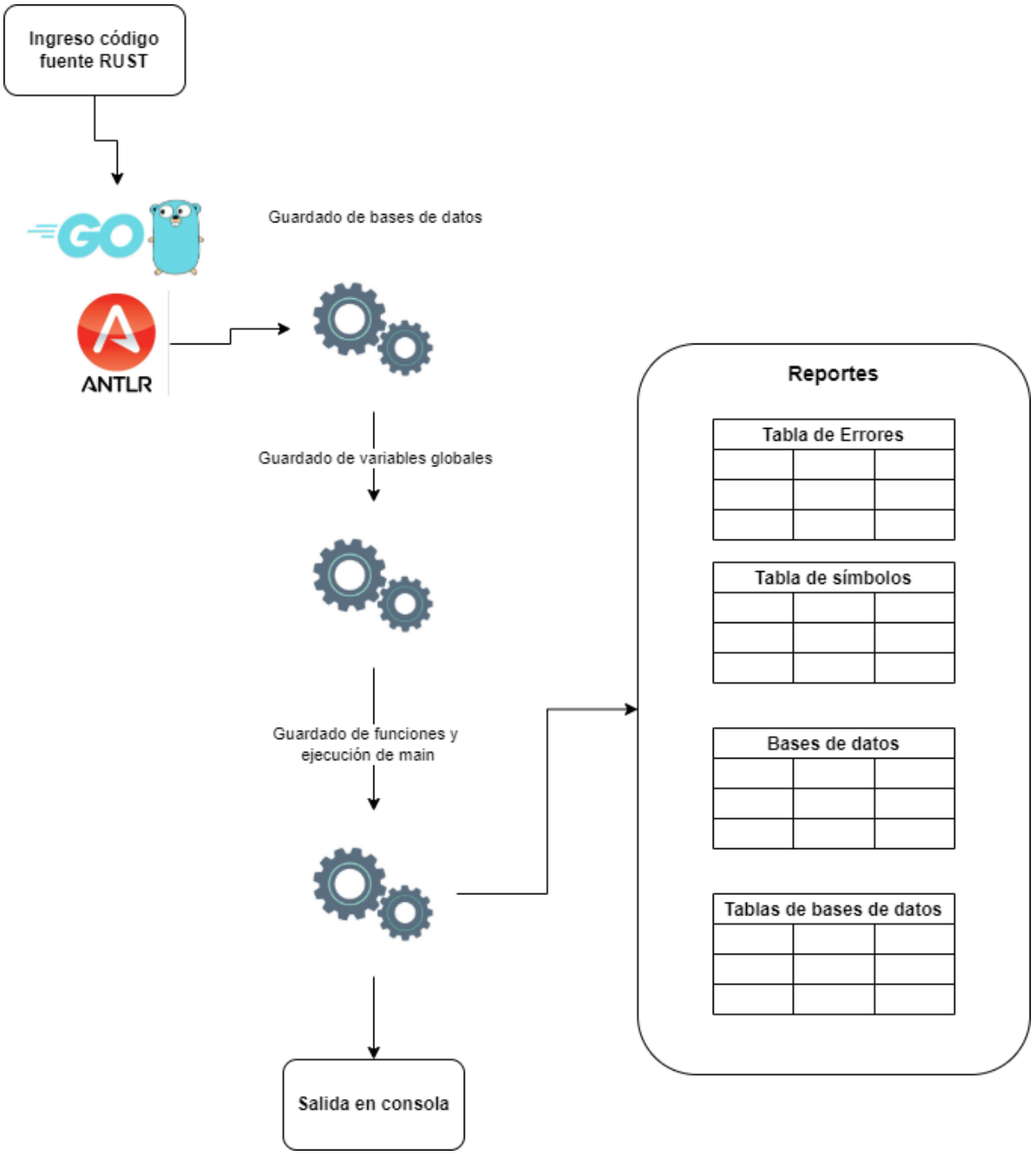

Ilustración 1: Flujo específico de la aplicación

2.3. Ingreso de código fuente

El lenguaje DB-Rust está basado en Rust con instrucciones limitadas. Se deberá de contar con un lugar en la interfaz gráfica donde los desarrolladores puedan programar o copiar código que luego podrá ser ejecutado

2.4. Ejecución

Si el código se ejecuta y encuentra errores semánticos en el mismo entonces se deberá contar con recuperación de errores, los únicos errores de recuperación serán de tipo semántico, esto con la finalidad de que se pueda seguir ejecutando el resto del código de entrada y finalizando mostrar dichos errores en el reporte respectivo, de lo contrario se generará la salida en consola satisfactoriamente.

2.5. Generación de reportes

El código fuente se colocará en la interfaz gráfica, copiandolo de un archivo fuente y pegando en la consola de entrada. El programa se ejecutará y mostrará los resultados en la consola. Luego de eso, el desarrollador podrá ver distintos reportes que se generaron con el código que ingresó. Entre los cuales se encuentran la tabla de errores, la tabla de símbolos, reporte de bases de datos y reporte de las tablas de las bases de datos. Estos reportes serán para que se verifique cómo el estudiante usa las estructuras internas para la interpretación del lenguaje. En la sección de reportes se detallará más al respecto de estos reportes.

3. Componentes de la aplicación

Se solicita la implementación de un editor básico para crear aplicaciones en el lenguaje DB-Rust, la cual contará con opciones de ejecutar, mostrar reportes y mostrar información del estudiante. A continuación, se describirán los componentes principales de la aplicación.

3.1. Página de editor

Este componente es un entorno de desarrollo básico que permite crear aplicaciones escribiendo como entrada el código fuente en un lenguaje de alto nivel y generar su respectiva salida en consola. En el editor solo basta con escribir el código fuente o copiar y pegar desde un archivo de entrada.

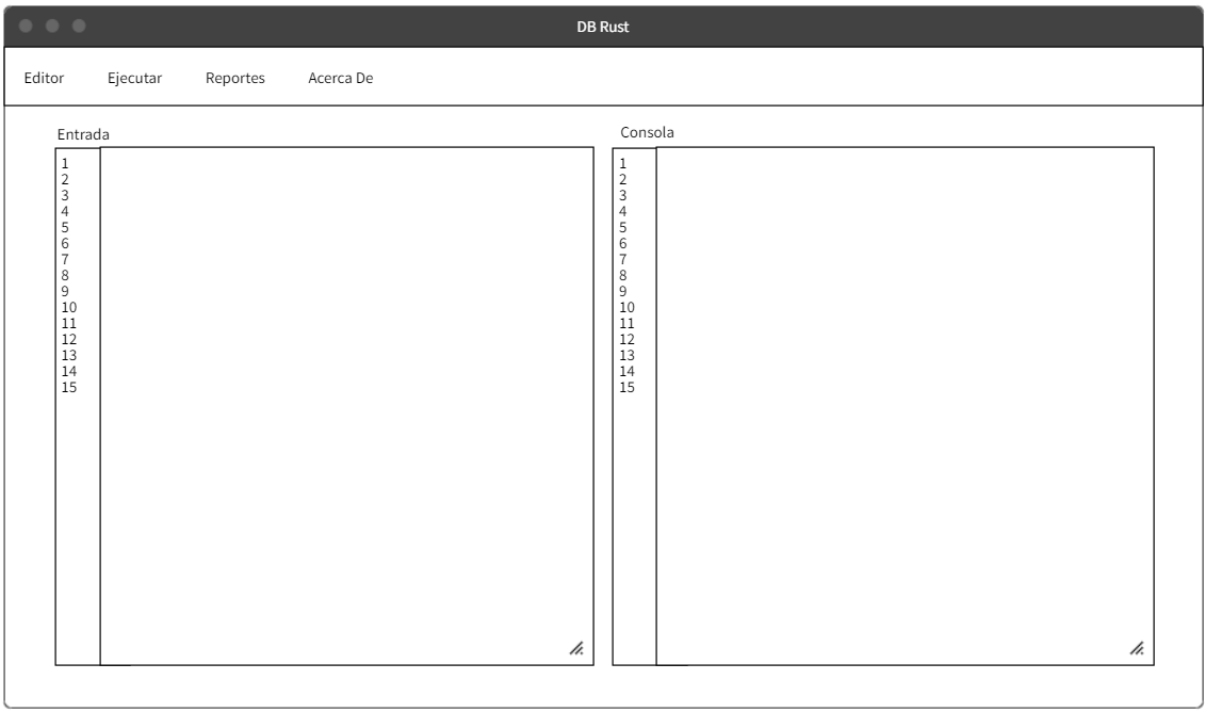

Ilustración 2. IDE básico

El editor tendrá las siguientes opciones:

- **Editor**: Redireccione a la página de editor.
- **Ejecutar**: Realiza la lectura del código fuente y efectúa el análisis mostrando el resultado en la consola.

- **Reportes**: Despliega los distintos reportes que puede generar, se describe en la página de reportes.
- **Acerca de**: Muestra los datos del estudiante tales como registro académico, nombre completo y sección del curso a la que pertenece.

Ilustración 3. Barra menú

3.2. Página de reportes

Este componente consiste en mostrar vistas de los reportes de tabla de símbolos, reporte de errores, reporte de base de datos existente y reporte de tablas de las bases de datos.

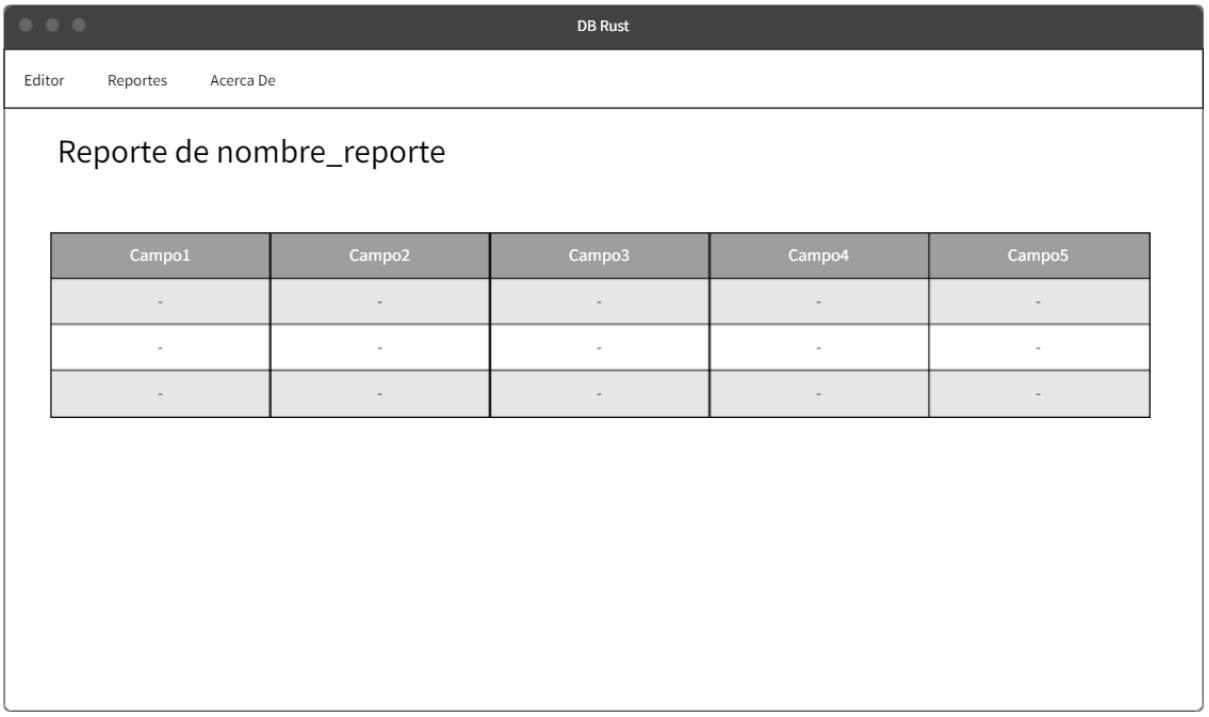

Ilustración 4. Página de reportes

Para mostrar un reporte en específico, la aplicación contará con cuatro opciones, se usará la vista de la ilustración 4 como plantilla para mostrar cada reporte:

Ilustración 5. Opciones de reportes

- **Reporte de tabla de símbolos**: Este mostrará la tabla de símbolos de toda la ejecución. Se espera que únicamente aparezcan todos los símbolos encontrados en el ámbito global y en los ámbitos locales.
- **Reporte de errores**: Este detalla todos los errores encontrados durante el análisis y la ejecución de la entrada.
- **Reporte de base de datos existente**: Este mostrará todas las bases de datos (módulos padre) que haya encontrado dentro del código fuente.
- **Reporte de tablas de base de datos**: Este mostrará las tablas que existen dentro de cada base de datos.

4. Sintaxis de BD-RUST

BD-RUST está basado en el lenguaje Rust, incluyendo únicamente las sentencias y reglas que se especificarán a continuación. El siguiente enlace muestra la documentación oficial de Rust, de donde se recopiló lo necesario para estructurar este proyecto.

http://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/getting-started.html

Puede consultar la siguiente página, para conocer más sobre la sintaxis de rust.

https://dev.to/psychecat/un-tour-rapido-por-rust-15i4

4.1. Generalidades

- Comentarios. Un comentario es un componente léxico del lenguaje que no es tomado en cuenta en el analizador sintáctico. En DB-Rust el único tipo de comentario existente es de una línea (//)
- Case Sensitive. Esto quiere decir que distinguirá entre mayúsculas y minúsculas.
- Identificadores. Un identificador de BD-RUST debe comenzar por una letra [A-Za-z] o guión bajo [_] seguido de una secuencia de letras, dígitos o guión bajo.
- Fin de instrucción. Se hace uso del símbolo ";" para establecer el fin de una instrucción.
- Declaración de variables. Bajo las reglas de Rust, al momento de declarar una variable, se debe especificar si será mutable o no. Si no se usa la palabra reservada 'mut' la declaración será una constante.

4.2. Tipos de datos válidos

BD-RUST únicamente aceptará los siguientes tipos de datos, cualquier otro no se deberá tomar en cuenta:

RUST no maneja Null, ya que busca garantizar la seguridad e integridad en todo momento.

- **i64:** valores numéricos enteros. Por ejemplo: 3,2,-1.
- **f64:** valores numéricos con punto flotante. Por ejemplo: 3.2,45.6,11.2.
- **bool:** valores booleanos, true o false.
- **char:** Literales de caracteres, se definen con comillas simples. Por ejemplo: 'a'.
- **&str o String:** Cadenas de texto definidas con comillas dobles.

- **usize**: La situación principal en la que usaría usize es al indexar algún tipo de colección (arreglo o vector).
- **Arreglos**: Conjunto de valores indexados entre 0 hasta n-1, de valores del mismo tipo. Para más información, consulte la sección 4.9.

```
[10, 20, 30, 40];
["Hola", "Mundo"];
[3.4, 2.0, 5.0];
```

- **Vectores:** Un vector es un puntero a una lista de objetos de un solo tipo, asignados dinámicamente y de tamaño dinámico. Para más información consulte la sección 4.10.

```
// vector mutable
let mut vector: Vec<i32> = Vec::new();
let till_five = vec![1, 2, 3, 4, 5];

// crear un vector con 20 elementos iguales
let ones = vec![<expresion>; 20];
```

- **Struct**: Estos son tipos compuestos definidos por el programador. Existen 2 tipos de Struct, aquellos que son mutables y los inmutables. Para mayor detalle, consulte la sección 4.11.

```
struct Color(u8, u8, u8);

// Forma típica de crear un struct
let c1 = Color(0, 0, 0);

// Especificando el indice de las propiedades
let c2 = Color{0: 255, 1: 127, 2: 0};
```

4.3. Expresiones

4.3.1. Aritméticas

Una operación aritmética está compuesta por un conjunto de reglas que permiten obtener resultados con base en expresiones que poseen datos específicos durante la ejecución. El lenguaje Rust no acepta que se utilicen tipos de datos diferentes para ser operados.

Para el manejo de la operación de tipos se usará el casteo explicito, el cual a una expresión se le agregara “as <tipo>” de esa manera se convertirá la expresión al tipo definido

A continuación se definen las operaciones aritméticas soportadas por el lenguaje.

Suma

La operación suma se produce mediante la suma de número o strings concatenados.

Operandos	Tipo resultante	Ejemplos
f64 + f64	**f64**	1.2 + 5.4 = 6.6 (5 as f64) + 3.4 = 8.4
i64 + i64	**i64**	2 + 3 = 5 (7.8 as i64) + 1 = 8

&str + str		`let string1: String = "hello".to_owned();`
		ó
		`let string1: String = "hello".to_string();`
		`let string2: &str = "world";`
		
		`let string3 = string1 + string2;`
		// para concatenar, se necesita una direccion &str y un bufer (.to_owned() o .to_string())

Resta

La resta se produce cuando se sustraen el resultado de los operadores, produciendo su diferencia.

Operandos	Tipo resultante	Ejemplos
f64- f64	**f64**	1.2 + 5.4 = -4.2 (4 as f64) - 3.45 = 0.55
i64 - i64	i64	2 + 3 = -1

Multiplicación

El operador multiplicación produce el producto de la multiplicación de los operandos.

Operandos	Tipo resultante	Ejemplos
f64* f64	**f64**	1.2 * 5.4 = 6.48
i64 * i64	**i64**	2 * 3 = 6

División

El operador división se produce el cociente de la operación donde el operando izquierdo es el dividendo y el operando derecho es el divisor.

Operandos	Tipo resultante	Ejemplos
f64/ f64	**f64**	1.2 / 5.4 = 0.222
i64 / i64	**f64**	6 / 4 = 1.5

Potencia

El operador de potenciación devuelve el resultado de elevar el primer operando al segundo operando de potencia.

Operandos	Tipo resultante	Ejemplos
i64::pow(i64 , i64)	**i64**	i64::pow(2, 2) = 4
f64::powf(f64 , f64)	**f64**	f64::powf(2.0,2.0) = 4

Módulo

El operador módulo devuelve el resto que queda cuando un operando se divide por un segundo operando.

Operandos	Tipo resultante	Ejemplos
f64 % f64	**f64**	1.0 % 5.0 = 1.0
i64 % i64	**i64**	6 % 3 = 0

4.3.2. Relacionales

Operador	Descripción
>	Mayor que: Compara ambos lados y verifica si el izquierdo es mayor que el derecho
<	Menor que: Compara ambos lados y verifica si el derecho es mayor que el izquierdo
>=	Mayor o igual que: Compara ambos lados y verifica si el izquierdo es mayor o igual que el derecho
<=	Menor o igual que: Compara ambos lados y verifica si el derecho es mayor o igual que el izquierdo
==	Igualación: Compara ambos valores y verifica si son iguales
!=	Distinto: Compara ambos lados y verifica si son distintos

EJEMPLOS:

Operandos	Tipo resultante	Ejemplos
ft64 [>, <, >=, <=] f64 i64 [>, <, >=, <=] i64 String [>, <, >=, <=] String	**Bool**	4.3 <= 4.3 = true 4 >= 4 = true “hola” > “hola” = false

4.3.3. Lógicas

Los siguientes operadores booleanos son soportados en BD-RUST. No se aceptan valores missing values ni operadores bitwise.

Operación lógica	Operador
OR	||
AND	&&
NOT	!

A	B	A && B	A || B	!A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	true	false	true

4.4. Impresión

Para mostrar información en la consola, DB-Rust cuenta con dos instrucciones para imprimir:

- Imprimir con salto de línea. Para eso se utiliza la función para imprimir *println!(formato, expresión)*.

El formato es opcional solamente si se imprime una cadena caso contrario seguir los siguientes formatos:

- Usar el formato *"{}"* para imprimir una expresión.
- Usar el formato *"{:?}"* para imprimir arreglos o vectores.

```
let b = [1, 2, 3, 4];
println!("+ -");                        // Imprime + -
println!("{} {} {}", "a", "a", "a"); // Imprime a a a

println!("El resultado de 2 + 2 es {}", (2+2));
// Imprime El resultado de 2 + 2 es 4

println!("{}", b[1]);                   // Imprime 2
println!("{:?}", b);                    // Imprime [1, 2, 3, 4]
```

4.5. Declaraciones y asignaciones

Una variable, en DB-Rust, es un nombre asociado a un valor. Las variables no pueden cambiar su tipo.

La declaración se puede realizar de la siguiente forma:

```
let mut ID: TIPO = Expresión;
ó
let mut ID = Expresión;
```

El sufijo **: TIPO** es opcional. Su función es asegurar que la expresión sea del tipo deseado. En caso la expresión sea distinta al tipo debe marcar un error.

```
let mut x : i64 = (3*5);          // Correcto, además, los paréntesis en
una expresión son válidos.
let mut str : i64 = "Saludo";     // ERROR: expected i64, got String
let mut var1 : String = true;     // ERROR: expected String, got Bool
let mut var = 1234;               // Correcto
```

Las variables pueden ser mutables o inmutables, si una declaración trae la palabra reservada **mut** esta variable podrá cambiar su valor en cualquier momento, pero si esta no la posee la variable nunca podrá cambiar su valor.

```
let mut x : i64 = 8200;     //Correcto
x = 67 + 90;                //Correcto

let y = "Hola";             //Correcto
y = "Adiós";    //ERROR una variable inmutable no puede cambiar su valor
```

Las asignaciones dentro de DB-Rust se componen del nombre de una variable igualándola a una expresión.

```
ID = Expresión;

// Asignación de arreglos y vectores
ID[Expresión] = ID[Expresión]
ID = ID[Expresión]
ID[Expresión] = Expresión

arr[2+3][2][1] = arr[2*8][2+5][a*0]
```

4.6. Funciones

4.6.1. Creación de funciones

Las funciones en DB-Rust se crean con la palabra reservada *fn* seguida del nombre de la función y, entre paréntesis, los parámetros de entrada de la función y puede que venga o no la declaración del tipo de retorno con la notación “-> TIPO”.

En DB-Rust permite retornar valores con la instrucción *return* y también sin la instrucción *return* permitiendo retornar expresiones. En caso no se utilice o se utilice return sin valor, la función no devuelve ningún dato.

```
fn NOMBRE_FUNCION (LISTA_PARAMETROS)  {
    LISTA_INSTRUCCIONES
}

fn NOMBRE_FUNCION (LISTA_PARAMETROS) -> TIPO {
    LISTA_INSTRUCCIONES
}
```

4.6.2. Funciones nativas

Rust cuenta con una gran variedad de funciones nativas. Sin embargo, DB-Rust contará con solo unas cuantas de las disponibles en Rust para el manejo de datos, las cuales se detallan a continuación:

- abs: devolverá el valor absoluto de una expresión numérica.
- sqrt: devolverá la raíz cuadrada de una expresión numérica.
- to_string(): devolverá la cadena con tipo de dato *“String”*.
- clone(): devolverá una copia de un recurso.

También se incluyen las siguientes funciones nativas para vectores:

- new: crea un nuevo vector vacío.
- len: devuelve el tamaño de un vector o arreglo.
- push: inserta un valor al final del vector.
- remove: remueve un valor en una posición en específico del vector.
- contains: devuelve si el vector o arreglo posee o no un valor en específico.
- insert: inserta un valor en una posición en específico.
- capacity: devuelve la capacidad máxima de un vector.
- with_capacity: utilizado para especificar el tamaño máximo de un vector.

4.6.3. Llamada a funciones

La llamada a funciones se realiza con el nombre de la función, y entre paréntesis, los parámetros a pasar.

```
funcion1(4,"Cadena");

let mut x = suma(2,5,9);
```

4.6.4. Paso por valor o por referencia

En DB-Rust, los únicos tipos que son pasados por referencia son los arreglos y struct, por lo que si se modifican dentro de una función también se modificarán fuera. El resto de tipos son pasados por valor.

```
// En este caso el vector [10,12, 45] se transformará en [3,12, 45]
fn valores(x: &mut [i64]) {
    x[0] = 3;
}

fn main () {
   let mut x: [i64; 3] = [10, 12, 45];
   valores(&mut x);
   println!("{:?}", x);
}
```

4.7. Sentencias de selección

4.7.1. If

El lenguaje DB-Rust cuenta con ramificación if-else que es similar a otros lenguajes, permite que porciones de código se ejecuten si la condición es evaluada a verdadero. La condición booleana no necesita estar entre paréntesis y cada condición va seguida de un bloque de instrucciones. Esta sentencia se define por las instrucciones *if, else if, else*.

Esta instrucción tiene la cualidad que se puede representar como una expresión, y todas las ramas deben devolver el mismo tipo.

Consideraciones:

- Las instrucciones *else if* y *else* son opcionales.
- La instrucción *else if* se puede utilizar tantas veces como se desee.
- Si la instrucción *if* es tratada como una expresión se debe de asegurar que deben devolver el mismo tipo de dato en cada bloque de instrucciones.

Ejemplos:

```
let x: i64 = 90;
let n = 10;

// Ejemplo 1
if x == 5 {
    println!("x es cinco!");
}

// Ejemplo 2
if x < 61 {
    println!("Reprobado con una nota de: {}", x);
}
else {
   println!("Aprobado con una nota de: {}", x);
}

// Ejemplo 3
let operacion =
    if n < 10 {
        10 * n      // Esta expresión devuelve un 'i64'
    } else if n == 10 {
        2 * n       // Esta expresión devuelve un 'i64'
    } else {
        n / 2       // Esta expresión debe devolver un 'i64' también
    }; // <- ¡No olvides poner un punto y coma aquí!

// Ejemplo 4 - otra forma de ver la instrucción if como expresión
let y = if x == 5 { 10 } else { 15 };
```

```
// Ejemplo 5
let bandera = true;
if bandera {
     println!("verdadero");
}
```

4.7.2. Match

DB-Rust permite realizar selecciones múltiples a través de la instrucción *match*, esta instrucción es similar a la sentencia *switch* de cualquier otro lenguaje, *match* toma una expresión y luego bifurca basado en su valor.

De la misma manera que la sentencia *if*, esta sentencia también tiene la cualidad que se pueden representar como expresión, y todas las ramas deben devolver el mismo tipo.

Consideraciones:

- Se debe de cubrir todos los valores posibles.
- Cuando no cubre todas las coincidencias se debe incluir el brazo por defecto.
- Si la instrucción match es tratada como una expresión se debe de asegurar que deben devolver el mismo tipo de dato en cada bifurcación.
- Las coincidencias deben ser del mismo tipo de datos en cada brazo y en la expresión después del match.

Ejemplos:

```
let numero = 15;

/* Ejemplo 1: Match como instrucción */
// Después del match sigue una expresión
match numero {
    // 1 | 2 | 3 estas son coincidencias
    1 | 2 | 3 => {
        let x = 100;
        println!("Rango de 1 a 3");
    }                                       //esto se conoce como brazo
    6 | 7 | 8 => println!("Rango de 6 a 8"),//esto se conoce como brazo

    "9" => println!("Rango de 6 a 8"),      // esto es un error!

    _ => println!("Resto de casos"),        //brazo por defecto
}

/* Ejemplo 2: Match como expresión */
let x = 5;
let numCadena = match x + 60 {
    1 => "uno",
```

```
    2 => "dos",
    3 => "tres",
    4 => "cuatro",
    5 => "cinco",
    _ => "otra cosa",
};
// ^ No olvidar ';'

/* Ejemplo 3: Match como expresión pero con todas las coincidencias
cubiertas */
let booleano = true;
let binario = match booleano {
    false => 0,
    true => 1,
};
// ^ No olvidar ';'
```

4.8. Sentencias Loops

DB-Rust provee de tres instrucciones iterativas, en el cual dos de ellas incluyen un bucle sobre una condición mientras que uno de ellos está en ciclo infinito, las instrucciones iterativas que soporta el lenguaje son las siguientes:

4.8.1. Loop

La sentencia loop es la sentencia más simple en el lenguaje, indica un bucle infinito y es la única instrucción que no requiere de una condición para ejecutar el bloque de instrucciones.

Consideraciones:

- La sentencia también actúa como una expresión, retornando el valor desde un break.

Ejemplos:

```
// Bucle infinito
loop {
    println!("Itera por siempre!");
}

// Expresion loop
let mut cont = 0;

let result = loop {
    cont = cont + 1;
    if cont == 10 {
        break cont * 2;
    }
```

```
};
// ¡No olvidar el punto y coma!

println!("El resultado es {}", result);
```

4.8.2. While

Esta sentencia ejecutará todo el bloque de instrucciones solamente si la condición es verdadera, de lo contrario las instrucciones dentro del bloque no se ejecutarán, seguirá su flujo secuencial.

Consideraciones:

- Si la condición es falsa, detendrá la ejecución de las sentencias de la lista de instrucciones.
- Si la condición es verdadera, ejecuta todas las sentencias de su lista de instrucciones.

Ejemplos:

```
// Ejemplo 1
// Contador del ciclo
let mut var1 = 0;
while var1 < 10 {
   print!("{}", var1);                // imprime 0123456789
   var1 = var1 + 1;
}
println!("");

// Ejemplo 2
let mut x = 5;                        // mut x: i64
let mut completado = false;           // mut completado: bool

while !completado {
    x = x - 3;
    print!("{}", x);                  // imprime 2-1-4-7-10
    if x % 5 == 0 {
        completado = true;
    }
}
```

4.8.3. For in

Esta sentencia es usada para iterar un número particular de veces, puede iterar sobre un rango de expresiones, arreglos y vectores. También puede iterar sobre una cadena de caracteres aunque primero se debe de convertir en un arreglo de caracteres.

Consideraciones:

- Contiene una variable declarativa que se establece como una variable de control, esta variable servirá para contener el valor de la iteración.
- La expresión que evaluará en cada iteración es de tipo rango o arreglo. Aunque también se puede especificar mediante una variable.

Ejemplos:

```
// Ejemplo 1 - Es aplicable para arreglos o vectores
let vector = vec!["Este", "semestre", "si", "sale"];
for valor in vector {
    println!("{}", valor);
}

// Ejemplo 2 - Es aplicable para arreglos o vectores
let arreglo = vec!["Este", "semestre", "si", "sale"];
for valor in 0..arreglo.len() {
    println!("{:?}", arreglo[valor]);
}

// Ejemplo 3
for n in 1..4 {               // Recorre rango de 1:4
   print!("{ } ", n);         // Únicamente se recorre ascendentemente
}                             // Imprime 1 2 3
println!("");

// Ejemplo 3 - cadena
for letra in "Hola Mundo!".chars() { // Recorre las letras de la cadena
    print!("{ } -", letra);   // Imprime H-o-l-a- -M-u-n-d-o-!-
}
println!("");

// Ejemplo 4 - variable cadena
let cadena = "OLC2";
for letra in cadena.chars() {
    print!("{ } -", letra);   // Imprime O-L-C-2-
}
println!("");

// Ejemplo 5 - Es aplicable para arreglos o vectores
for letra in ["perro", "gato", "tortuga"] {
    print!("{} es mi favorito, ", letra);

//Imprime: perro es mi favorito,  gato es mi favorito,  tortuga es mi favorito,
}
```

4.8.4. Sentencias de transferencia

En ocasiones surge la necesidad de detener un ciclo de manera temprana antes que la condición sea falsa o detener un bucle infinito, también está la posibilidad de saltar algunas instrucciones de un ciclo determinado y de salir de un bucle cuando se requiera retornar un valor en una función, para estás circunstancias el lenguaje DB-Rust define las siguientes instrucciones: **Break, Continue y Return.**

Consideraciones:

- Se debe validar que la sentencia *break y continue* se encuentre únicamente dentro de un bucle.
- Es necesario validar que la sentencia *break* detenga la sentencia asociada para el ciclo más interno.
- Es necesario validar que la sentencia *continue* salte a la siguiente iteración asociada a su sentencia cíclica más interna.
- Es requerido validar que la sentencia *return* esté contenida únicamente en una función.
- La instrucción *break* retorna un valor solamente dentro de la instrucción *loop*, en caso de *for* y *while* deberá notificar el error.

Ejemplos:

```
// Ejemplo 1
while true {
    println!("true");            // Imprime solamente una vez true
    break;
}
break;                          // Error

// Ejemplo 2
let mut num = 0;
while num < 10 {
   num = num + 1;
   if num == 5 {
      continue;
   }
   print!("{ }", num);           // Imprime 1234678910
}

// Ejemplo 3 aplica para funciones
// Imprime No: 1 No:2 No: 3 No: 4
// Retorna 5
fn funcion() -> i64 {
   let mut num = 0;
   while num < 10 {
       num = num + 1;
       if num == 5 {
           return 5;
       }
       print!("No: {} ", num);
    }
```

```
    return 0;
}
```

4.9. Arreglos

Un arreglo es una colección de múltiples valores secuenciales que tiene una longitud fija, cada elemento de un arreglo debe ser del mismo tipo de dato. Cada elemento en un arreglo es asignado un número de índice e inicia por el número 0 hasta n-1 donde n es el tamaño de la colección.

4.9.1. Definición de un arreglo

En DB-Rust, los arreglos se definen en una dimensión, dos dimensiones, tres dimensiones o de *"n" dimensiones*, cuando un arreglo tiene más de una dimensión se conoce como arreglos multidimensionales.

Para definir un arreglo se escribe primero la variable, el tipo de dato (opcional) luego los valores como una lista de elementos separadas por comas dentro de corchetes.

Ejemplo:

```
struct Persona {
    id: i64
}

// arreglo de 1 dimensión
let arr0: [Persona; 1] = [Persona{ id: 0 }];
let arr1: [&str; 2] = ["Hola", "Mundo"];
let arr2: [String; 2] = ["Hola".to_string(), "Mundo".to_string()];

println!("{}", arr0[0].id);   // imprime 0

// arreglo de 3 dimensiones
let mut arr3: [[[i64; 4];2]; 2] = [
    [ [ 1,  3,  5,  7], [  9, 11, 13, 15] ],
    [ [ 2,  4,  6,  8], [10, 12, 14, 16] ]
];

arr3[0][1][3] = 50;

println!("{:?}", arr1);         // imprime ["Hola", "Mundo"]
println!("{:?}", arr2);         // imprime ["Hola", "Mundo"]
println!("{:?}", arr3[0][1]);   // imprime [9, 11, 13, 50]
```

Otra alternativa para definir los arreglos es asignar entre corchetes la expresión que estará asociado a cada elemento del arreglo seguido de la cantidad de valores que tendrá.

Ejemplo:

```
// arreglo de 1 dimensión
let arr1: [&str; 4] = ["Hola"; 4]; // ["Hola", "Hola", "Hola", "Hola"]

// arreglo de 3 dimensiones
let mut arr2 = [
    [ [ 1,  3,  5,  7], [  5;4 ] ],
    [ [ 2,  4,  6,  8], [ 10;4 ] ],
    [ [ 2; 4 ], [ 0; 4 ] ]
];

println!("{:?}", arr2[0][1]);    // imprime [5, 5, 5, 5]
```

4.9.2. Acceso de elementos

Los elementos individuales de un arreglo pueden ser accedidos usando su número de índice correspondiente, para el acceso se escribe el nombre del arreglo indicando la posición a la que se desea acceder entre corchetes, si es un arreglo multidimensional se deberá escribir una lista de corchetes indicando la posición del elemento ([indice] [indice]). En el índice puede ser cualquier expresión de tipo entero.

Ejemplo:

```
let arr1 = [90+9, 28*5, 34/2, 56];
println!("{}", arr1[0]);        // imprime 99

let arr2 = [
    [ [ 1,  3,  5,  7], [  9, 11, 13, 15] ],
    [ [ 2,  4,  6,  8], [10, 12, 14, 16] ]
];
println!("{:?}", arr2[0][1]);   // imprime [9, 11, 13, 15]
```

4.10. Vectores

Un vector es una colección de datos homogénea que almacena los datos como una secuencia ordenada de elementos, un vector puede crecer o reducirse en cualquier momento. Cada elemento en un vector es asignado un número de índice e inicia por el número 0 hasta n-1 donde n es el tamaño de la colección.

Ejemplo:

```
// Tamaño del vector n = 5
// Posiciones de acceso desde 0 a 4
let mut vector = vec![1,2,3,4,5];
```

4.10.1. Definición de un vector

Los vectores se pueden definir por un solo vector o vector dentro de vectores.

Ejemplo:

```
let v = vec![vec![1; 10],vec![2; 8],vec![3; 15],vec![5; 2],vec![8; 1]];
```

Para crear un vector vacío, se debe de llamar a la función predefinida *Vec::new()*, tenga en cuenta que se debe de agregar una notación de tipo porque no se está ingresando un valor en el vector y DB-Rust no sabe de qué tipo de elementos se pretende almacenar.

Ejemplo:

```
// Vector vacío
let v: Vec<i64> = Vec::new();
```

DB-Rust puede deducir el tipo de un valor que se quiere almacenar que raramente se especifica el tipo, esta forma es utilizada para crear un vector con valores iniciales y se denota como *vec!*. Otra forma alternativa es cuando se requiere crear un vector con valores repetidos cambiando su sintaxis.

Ejemplo:

```
// Vector con valores iniciales
let v = vec![1,2,3,4,5];

// Valores repetidos
let v = vec![0; 10];
// es un vector de diez 0's   [0,0,0,0,0,0,0,0,0,0]
```

También permite crear vectores especificando la capacidad, si al vector se excediera de la capacidad, DB-Rust cambiaría el tamaño del vector y este cambio consiste básicamente en crear un nuevo vector que tenga el doble de capacidad y copiar sobre el vector anterior.

Ejemplo:

```
// Vector with_capacity
let v: Vec<i32> = Vec::with_capacity(10);
```

4.10.2. Push

Cuando se crea un vector existe la posibilidad de se quiera agregar elementos extra cuando la variable es mutable, DB-Rust cuenta con el método push, la cual permite agregar elementos al final del vector. Y es utilizada especificando primero la variable luego el punto y seguidamente la palabra *push*.

Ejemplo:

```
// Vector vacío
let mut v: Vec<i64> = Vec::new();
v.push(1);
v.push(2);
v.push(3);
v.push(4);
println!("{:?}", v);      // imprime el vector [1,2,3,4]
```

4.10.3. Insert

DB-Rust permitirá tener otra función para agregar elementos y es la función insert, en esta función tiene dos parámetros, la primera permite especificar la posición donde almacenará el nuevo dato en el vector y el segundo parámetro se ingresa el valor que estará en el vector. Tomar en cuenta que la variable debe ser mutable.

Ejemplo:

```
let mut v = vec![2,4,6,8,10];
v.insert(2, 10);
println!("{}", v[2]);        // Imprime 10
// el vector tendrá los siguientes datos [2,4,10,6,8,10]
```

4.10.4. Remove

Así como se pueden agregar elementos también existe la posibilidad de eliminar elementos de un vector, el lenguaje DB-Rust cuenta con el método *remove*, la cual permite remover y retornar el elemento dentro de un vector especificando el índice, desplazando los elementos a la izquierda. Siempre y cuando la variable sea mutable.

Ejemplo:

```
let mut v = vec![2,4,6,8,10];
let num = v.remove(3);
// después de remove el vector queda así [2,4,6,10];
```

```
println!("{}", num);           // imprime 8
```

4.10.5. Contains

Esta función retornará true si el segmento (vector o arreglo) contiene un elemento con el valor dado en caso contrario retorna false, el valor debe ser mediante un referencia como se ve en el ejemplo:

Ejemplo:

```
let v = vec![2,4,6,8,10];
if v.contains(&2) {
    println!("true");
} else {
    println!("false");
}
```

4.10.6. Len

Está función retorna el número de elementos en un vector o arreglo, servirá para conocer la longitud del segmento.

Ejemplo:

```
let v = vec![2,4,6,8,10];
println!("{}", v.len());        // Imprime 5
```

4.10.7. Capacity

Está función retorna el límite de elementos que permite actualmente un vector. En caso de que se utilice está función en un vector que no se inicializó con *with_capacity*, la capacidad debe ser superior al número de elementos que tiene el vector.

Ejemplo:

```
let mut v: Vec<i32> = Vec::with_capacity(10);
v.push(1);
v.push(2);
v.push(3);

println!("{}", v.capacity());      // Imprime 10
```

4.10.8. Acceso de elementos

Conociendo la forma de crear y agregar elementos, ahora se debe conocer la manera de leer cada elemento. Los elementos individuales de un vector pueden ser accedidos usando su número de índice correspondiente.

Ejemplo:

```
let v = vec![1,2,3,4,5];
let num = v[0];
println!("{}", num);      // imprime 1
```

4.11. Structs

Los *structs* son tipos compuestos que se denominan registros, los tipos compuestos se introducen con la palabra clave *struct* seguida un identificador y luego un bloque de nombres de campos, opcionalmente con tipos usando el operador ":".

Consideraciones:

- Los atributos de los *structs* declarados como inmutables no pueden modificar sus atributos después de la construcción.
- Los *structs* también se pueden utilizar como retorno de una función.
- Las declaraciones de los *structs* se pueden utilizar como expresiones.
- Los atributos se pueden acceder por medio de la notación ".".

```
// Struct
struct Personaje {
     nombre: String,
     edad: i64,
     descripcion: String
}

// Struct
struct Carro {
     placa: String,
     color: String,
     tipo: String
}

fn main(){
 // Construcción Struct
 let mut p1 = Personaje { nombre:"Fer".to_string(), edad:18,
descripcion:"No hace nada".to_string() };

 let mut p2 = Personaje { nombre:"Fer".to_string(), edad:18,
descripcion:"Maneja un carro".to_string()};

 let mut c1 = Carro { placa:"090PLO".to_string(),
color:"gris".to_string(), tipo:"mecanico".to_string() };

 let mut c2 = Carro { placa:"P0S921".to_string(),
color:"verde".to_string(), tipo:"automatico".to_string() };

 // Asignación Atributos
 p1.edad = 10;                          // Cambio aceptado
 p2.edad = 20;                          // Cambio aceptado
 c1.color = "cafe".to_string();         // Cambio aceptado
 c2.color = "rojo".to_string();         // Cambio aceptado
 // Acceso Atributo
 println!("{}", p1.edad);                    // Imprime 18
 println!("{}", c1.color);                    // Imprime "cafe"
}
```

4.12. Módulos

Los módulos son el principal componente de DB-Rust, estos son utilizados para almacenar bloques de código los cuales serán utilizados para la simulación de bases de datos y tablas de bases de datos.

Dentro de DB-Rust los módulos se declaran con la palabra reservada *mod* seguido de un identificador y luego un bloque de código entre dos llaves "{}", dentro de los módulos pueden declararse otros módulos hijos declarados de la misma forma "*mod ID {...}*", y también pueden declararse funciones.

Consideraciones:

- Los atributos de los *módulos* declarados como privados no pueden ser accedidos fuera del módulo.
- Los atributos de los *módulos* declarados como públicos sí pueden ser accedidos fuera del módulo.
- Los atributos públicos de los módulos son declarados con la palabra reservada *pub* seguido de la palabra reservada *mod* o *fn*, dependiendo si se está declarando un módulo o una función.
- Los atributos se pueden acceder por medio de la notación "::".

```
// Ejemplo de módulo simple
mod Cine{
    pub fn direccion() -> String {
        return "6ta calle 4-67, avenida cuadro".to_string();
    }

    fn empleados() -> i64 {
        return 23;
    }
}

//Ejemplo de un módulo anidado
mod Parque {
    pub mod Juego {
        pub fn nombre() -> String {
            return "Columpio".to_string();
        }
    }
}

fn main() {
    println!("{}", Cine::direccion());//Instrucción aceptada
```

```
    println!("{}", Cine::empleados());//ERROR, no es un atributo público
    println!("{}", Parque::Juego::nombre());//Instrucción aceptada
}
```

4.13. Tabla de Resumen de instrucciones

Instrucción	Descripción
Println!	Imprime el resultado de la expresión agregando el salto de línea.
Declaración	Crea una nueva variable dentro del entorno con un valor en específico.
Asignación	Altera el valor de una variable dentro del entorno.
Llamada a funciones	Llama a una función para que sea ejecutada, luego regresa al flujo normal donde se realizó la llamada.
Funciones	Son secuencias de instrucciones definidas en un bloque para ser ejecutadas en una llamada de la función.
If	La instrucción condicional ejecuta el bloque de instrucciones si la condición es verdadera.
Else If	Si la condición en if es falsa, esta instrucción evaluará una nueva condición si es verdadera ejecuta las instrucciones del bloque.
Else	Si las condiciones en if y else if son falsas se ejecuta las instrucciones que estén en else.
Match	Permite hacer la elección de un bloque de código a través de una condición.
While	Esta instrucción ejecuta el bloque de instrucciones si la condición es verdadera.
For	La instrucción for puede iterar sobre tipos iterables como lo son rangos, arreglos y cadenas.
Break	Esta instrucción termina la ejecución de una instrucción cíclica antes de que se alcance el final del objeto iterable. Solamente en el loop puede retornar un valor.
Continue	Esta instrucción salta a la siguiente iteración sin ejecutar las instrucciones restantes.
Return	La instrucción return se utiliza para devolver un resultado en las funciones.

5. Simulación de bases de datos

El objetivo principal de DB-Rust es la simulación de bases de datos, cabe recalcar que los datos **no serán persistentes**, todos los datos que se utilicen solo serán usados durante la ejecución del programa fuente que se ingrese en el analizador.

Para la creación de las bases de datos se hará uso de los **módulos**, como se mencionó en la sección de módulos, estos serán utilizados para almacenar todas las instrucciones necesarias para la simulación.

El analizador deberá ser capaz de leer el código de entrada y guardar todos los módulos antes de ejecutar las instrucciones, ya que se podrá hacer referencia a los módulos en cualquier parte del programa. La existencia de módulos dentro del programa fuente debe ser **obligatoria,** si el analizador no detecta ningún módulo este deberá ser un error de semántica y no continuar con la ejecución ya que sin estos se perdería el objetivo del proyecto.

La manera en que se simularán las bases de datos se describe de la siguiente manera:

- Todos los módulos globales representarán las bases de datos.
- Los módulos que se encuentren dentro de los módulos globales representarán las tablas de las base de datos
- Las funciones dentro de los módulos harán toda la interacción para el flujo de los datos: inserción, obtener, eliminar, etc.

A continuación se muestra un ejemplo de cómo se define una base de datos con los módulos:

```
mod tienda {                        //Base de datos

    pub mod ventas {                //Tabla ventas

        pub struct Venta {
            pub total: i64,
            pub cliente: String
        }

        // Definición del método de creación de la tabla
pub fn crear_tabla(mut _tabla: Vec<Venta>, tamanio: usize) -> Vec<Venta>
{
    _tabla = Vec::with_capacity(tamanio);

    println!("La tabla ventas ha sido creada");

    return _tabla;
}
```

```
        //Definición de la inserción de datos en la tabla
pub fn insertar_tabla(mut _tabla: Vec<Venta>, total: i64, cliente:
String) -> Vec<Venta> {
            if _tabla.len() < _tabla.capacity() {

                //Insertar un valor nuevo
                let valor: Venta = Venta {
                    total: total,
                    cliente: cliente.to_string()
                };

                _tabla.push(valor);
                println!("Valor nuevo agregado a tabla ventas");
            }
            else {
                println!("La tabla ha llegado a su maxima capacidad");
            }
            return _tabla;
    }

         // Definición de la obtención de un dato según su índice
pub fn select_venta_por_id(mut _tabla: Vec<Venta>, id: usize) -> Venta {
            let value: Venta = Venta {
                total: _tabla[id].total,
                cliente: _tabla[id].cliente.clone()
            };
            return value;
        }
    }
}
```

Uso del módulo dentro de la función main:

```
fn main() {
    let mut vector: Vec<tienda::ventas::Venta> = Vec::new();

    //Iniciar la tabla
    vector = tienda::ventas::crear_tabla(vector,10);

    //Insertar valor
vector = tienda::ventas::insertar_tabla(vector,15,"Hector".to_string());

    //Obtener un valor
    let getValue: tienda::ventas::Venta =
tienda::ventas::select_venta_por_id(vector,0);

println!("total: {} cliente: {}", getValue.total, getValue.cliente);
}
```

Salida esperada:

```
La tabla ventas ha sido creada

Valor nuevo agregado a tabla ventas

total: 15, cliente: Hector
```

Revisar el repositorio https://github.com/pablorocad/DB-Rust, ahí estará publicado otro ejemplo con más detalles.

6. Reportes generales

6.1. Tabla de símbolos

En este reporte se solicita mostrar la tabla de símbolos después de la ejecución del archivo. Se deberán mostrar todas las variables, funciones y struct reconocidas, junto con su tipo y toda la información que el estudiante considere necesaria. En las funciones, deberá mostrar el nombre de sus parámetros, en caso tenga. El reporte debe tener como mínimo la siguiente información:

- Identificador.
- Tipo de símbolo.
- Tipo de dato.
- Ámbito al que pertenece.
- Fila.
- Columna.

ID	Tipo Símbolo	Tipo de dato	Ámbito	Fila	Columna
x	Variable	i64	Global	2	18
valores	Funcion	i64	Global	2	1
arr	Arreglo	vec	Global	6	1
x	Arreglo	vec	Local	7	1

6.2. Tabla de errores

La aplicación deberá ser capaz de detectar y reportar todos los errores semánticos que se encuentren durante la ejecución. Su reporte debe contener como mínimo la siguiente información.

- Descripción del error.
- Ámbito donde se encontró el error.
- Número de línea donde se encontró el error.
- Número de columna donde se encontró el error.
- Fecha y hora en el momento que se produce un error.

No.	Descripción	Ámbito	Línea	Columna	Fecha y hora
1	El struct Persona no fue declarado	Global	112	15	14/8/2021 20:16
2	El tipo string no puede multiplicarse con un real	Local	80	10	14/8/2021 20:16
3	No se esperaba que la instrucción break estuviera fuera de un ciclo.	Local	1000	5	14/8/2021 20:16

6.3. Tabla de bases de datos existentes

La aplicación deberá crear reporte de las bases de datos existentes, permitirá mostrar todas las bases de datos que se hayan encontrado en el análisis. El reporte debe contener la siguiente información:

- Nombre de la base de datos.
- El número de tablas existentes.
- Línea en que se encontró la base de datos.

No.	Nombre	No. Tablas	Línea
1	Tienda	4	58
2	Control académico	2	1500
3	Proyecto 1	8	90

6.4. Tablas de base de datos

Este reporte deberá mostrar las tablas que existen en cada base de datos existentes en las que fueron encontrados después del análisis. El reporte debe contener la siguiente información:

- Nombre de la tabla.
- Nombre de la base de datos.
- Línea donde fue encontrada.

No.	Nombre tabla	Nombre base de datos	Línea
1	Venta	Tienda	58
2	Compra	Tienda	1500
3	Control académico	Control académico	90
4	Práctica	Proyecto 1	80

7. Manejo de errores semánticos

El intérprete deberá ser capaz de detectar todos los errores semánticos que se encuentren durante todo el análisis. Todos los errores se deberán de recolectar y se mostrará un reporte de errores mencionado en la sección 6.

En la siguiente tabla se muestra los errores que puede encontrar el estudiante en todo el análisis:

No.	Validación	Ejemplo	Error
1	Se debe de hacer las respectivas validaciones para no operar tipos incompatibles.	✘ “prueba” * 5	Tipos incompatibles
2	Validar que al hacer referencia a un ID en cualquier tipo de expresión, este si exista.	✘ funcion_suma(x + 2)	Variable x no encontrada

3	Validar que las condiciones en cualquier instrucción de control sea válido	✘ if x + 5 {
		}
		
			Condición invalida, debe devolver un bool

4	Validar que las instrucciones break y continue estén dentro de un bloque de repetición	✘ continue;
		loop {
		let x mut = 0;
		if x == 9 {
		break;
		}
		}
		✘ break;
			Instrucciones de transferencia en lugares incorrectos

5	Validar que la instrucción return estre dentro de una función y el retorno sea correspondiente al tipo que la función	fn calc() -> bool {
		✘ return 5
		}
		✘ return false
			- Valor de retorno no válido.
			
			- Return fuera del ámbito de función

6	Validar tipos de parámetros en función	fn calc(x : i64) -> i64 {
		return x * 5
		}
		
		✘ calc(“cadena”)
			Se esperaba i64 y se encontró string

7	Validar declaración existente de variables y funciones	let x : i64 = 0;	Variable x ya declarada

		✘ let x: f64 = 3.5;	

8	Validar asignaciones de diferentes tipos, en variables, vectores o arreglos.	let x:i64 = 0;
		let array = [1, 2, 3, 4, 5];
		
		x = “cadena” ✘
		array[0] = “cadena”
			No se puede asignar string a i64

9	Validar acceso a arreglos y vectores, fuera del rango	let array = [1, 2, 3, 4, 5]; println!(array[100]); ✘	desbordamiento de memoria, index 100 invalido
10	Validar errores aritméticos	println!(12/0) ✘	No es posible dividir un número entre 0

11	Validar acceso a propiedades no públicas de módulos	mod ejercicio {
		fn suma(a,b) {
		return a + b;
		}
		}
		
		println!(ejercicio::suma(1,1));
			No es posible acceder a una propiedad dentro de un módulo si no ha sido declara publica (pub)

12	Validar asignación a variables no mutables	let x = 5; x = 10;	no se puede cambiar el valor de una variable no mutable

13	Validar structs mutables	let emp1 = Employee {
		age:50
		};
		
		emp1.age = 51;
			No se puede modificar la propiedad de un struct no mutable

14	Validar existencia de métodos o funciones.	et mut v: Vec<i32> =
		Vec::new();
		v.push(1);
		v.push(2);
		
		v.apilar(3);
			Método apilar no encontrado en la definición

15	Validar que un rango sea proveniente de un objeto vector	for valor in 0..objeto.len() {
		println!(objeto[valor]);
		}
			Objeto no es un vector, error de rango

16	Validar expresión en función match	let persona = Persona { } let numCadena = match persona { 1 => “uno”,	persona no es un tipo válido para un match

		_ => “otra cosa”, };	
17	Validar que exista una o más definiciones de módulos	// sin módulos	No se encontró ninguna definición de módulo.

8. Entregables y calificación

Para el desarrollo del proyecto se deberá utilizar un repositorio de GitHub, este repositorio deberá ser privado y tener a los auxiliares como colaboradores.

8.1. Entregables

El código fuente del proyecto se maneja en GitHub por lo tanto, el estudiante es el único responsable de mantener actualizado dicho repositorio hasta la fecha de entrega, si se hacen más commits luego de la fecha y hora indicadas no se tendrá derecho a calificación.

- Código fuente y archivos de compilación publicados en un repositorio de GitHub cada uno en una carpeta independiente.
- Enlace al repositorio y permiso a los auxiliares para acceder. Para darle permiso a los auxiliares, agregar estos usuarios al repositorio:
    - pablorocad
    - Losajhonny
    - alexYovani53
- Interfaz gráfico para el manejo de la aplicación

8.2. Restricciones

- La herramienta para generar los analizadores del proyecto será ANTLR. La documentación se encuentra en el siguiente enlace https://www.antlr.org/.
- No está permitido compartir código con ningún estudiante. Las copias parciales o totales tendrán una nota de 0 puntos y los responsables serán reportados a la Escuela de Ingeniería en Ciencias y Sistemas.
- El desarrollo y entrega del proyecto es individual.

8.3. Consideraciones

- Es válido el uso de cualquier librería de **Go** para el desarrollo de la interfaz gráfica.
- El repositorio únicamente debe contener el código fuente empleado para el desarrollo, no deben existir archivos PDF o DOCX.
- El sistema operativo a utilizar es libre.
- Se van a publicar archivos de prueba y dudas sobre sintaxis del lenguaje en el siguiente repositorio: https://github.com/pablorocad/DB-Rust.
- El lenguaje está basado en Rust (https://www.rust-lang.org/) por la última versión 1.58.1, por lo que el estudiante es libre de realizar archivos de prueba en esta herramienta, el funcionamiento debería ser el mismo y limitado a lo descrito en este enunciado.

8.4. Calificación

- Durante la calificación se realizarán preguntas sobre el código y reportes generados para verificar la autoría de este, de no responder correctamente la mayoría de las preguntas se reportará como copia.
- Se tendrá un máximo de 45 minutos por estudiante para calificar el proyecto. La calificación será de manera virtual y se grabará para tener constancia o verificación posterior.
- La hoja de calificación describe cada aspecto a calificar, por lo tanto, si la funcionalidad a calificar falla en la sección indicada se tendrá 0 puntos en esa funcionalidad y esa nota no podrá cambiar si dicha funcionalidad funciona en otra sección.
- Si una función del programa ya ha sido calificada, esta no puede ser penalizada si en otra sección la función falla o es errónea.
- Los archivos de entrada permitidos en la calificación son únicamente los archivos de pruebas preparados por los tutores.
- Los archivos de entrada podrán ser modificados solamente antes de iniciar la calificación eliminando funcionalidades que el estudiante indique que no desarrolló.
- Los archivos de entrada podrán ser modificados si contienen errores semánticos no descritos en el enunciado o provocados para verificar el manejo y recuperación de errores.

8.5. Entrega de proyecto

- La entrega será mediante GitHub, y se va a tomar como entrega el código fuente publicado en el repositorio a la fecha y hora establecidos.
- Cualquier commit luego de la fecha y hora establecidas invalidará el proyecto, por lo que se calificará hasta el último commit dentro de la fecha válida.
- Fecha de entrega:

Sábado 20 de marzo de 8:00 A.M. a 8:00 P.M.