Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas
Organización de Lenguajes y Compiladores 2
Segundo Semestre 2022
Catedráticos: Ing. Edgar Sabán, Ing. Bayron López e Ing. Luis Espino
Tutores académicos: Alex Jerónimo, Erick Villatoro, Eduardo Ajsivinac

DB-RUST

Segundo proyecto de laboratorio

1. Competencias

1.1. Competencia general

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

1.2. Competencias específicas

- Que los estudiantes utilicen herramientas para la generación de analizadores léxicos y sintácticos.
- Que los estudiantes apliquen los conocimientos adquiridos durante la carrera y el curso para el desarrollo de la solución.
- Que los estudiantes realicen análisis semántico, la generación de código intermedio y optimización del código intermedio del lenguaje DB-Rust.
- Que los estudiantes generen una traducción de código de alto nivel a código de tres direcciones.

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.

El fin de este proyecto es que el estudiante pueda comprender y aplicar la teoría de código intermedio, para generar una salida en código de tres direcciones escrito en lenguaje C, la cual deberá ser equivalente a la salida de un intérprete, al ejecutar una entrada en alto nivel en lenguaje RUST.

El código resultante no debe contener instrucciones complejas, únicamente las descritas en las siguientes secciones y para validar ello, se utilizara un compilador de C para validar el correcto funcionamiento de la salida.

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. La aplicación tendrá la opción de traducir un código fuente o de optimizar un código en tres direcciones ya sea por mirilla o bloques.

Flujo para traducir a código tres direcciones:

1. El analizador deberá hacer el escaneo de la entrada en lenguaje RUST.
2. Durante el escaneo de la entrada, el analizador irá construyendo una salida usando únicamente 3 direcciones simuladas de memoria.
3. Antes de imprimir la salida, el código generado deberá de tener la estructura de un programa en C, incluyendo los encabezados necesarios para el manejo del Stack y el Heap. Esto se detalla en la sección 6.11.
4. Se imprime la salida en sintaxis de C, para proceder a su validación.

Flujo para optimizar un código en tres direcciones:

1. El analizador deberá recorrer el código en tres direcciones ingresadas y luego tendrá que aplicar cada una de las reglas de la optimización por bloques.
2. Este proceso de optimización podrá hacerse N cantidad de veces siempre y cuando se ingrese código de tres direcciones al analizador.
3. Por último, se deberá mostrar en el apartado de reportes la aplicación de las reglas de optimización con la información que se detalle en la sección de reportes.

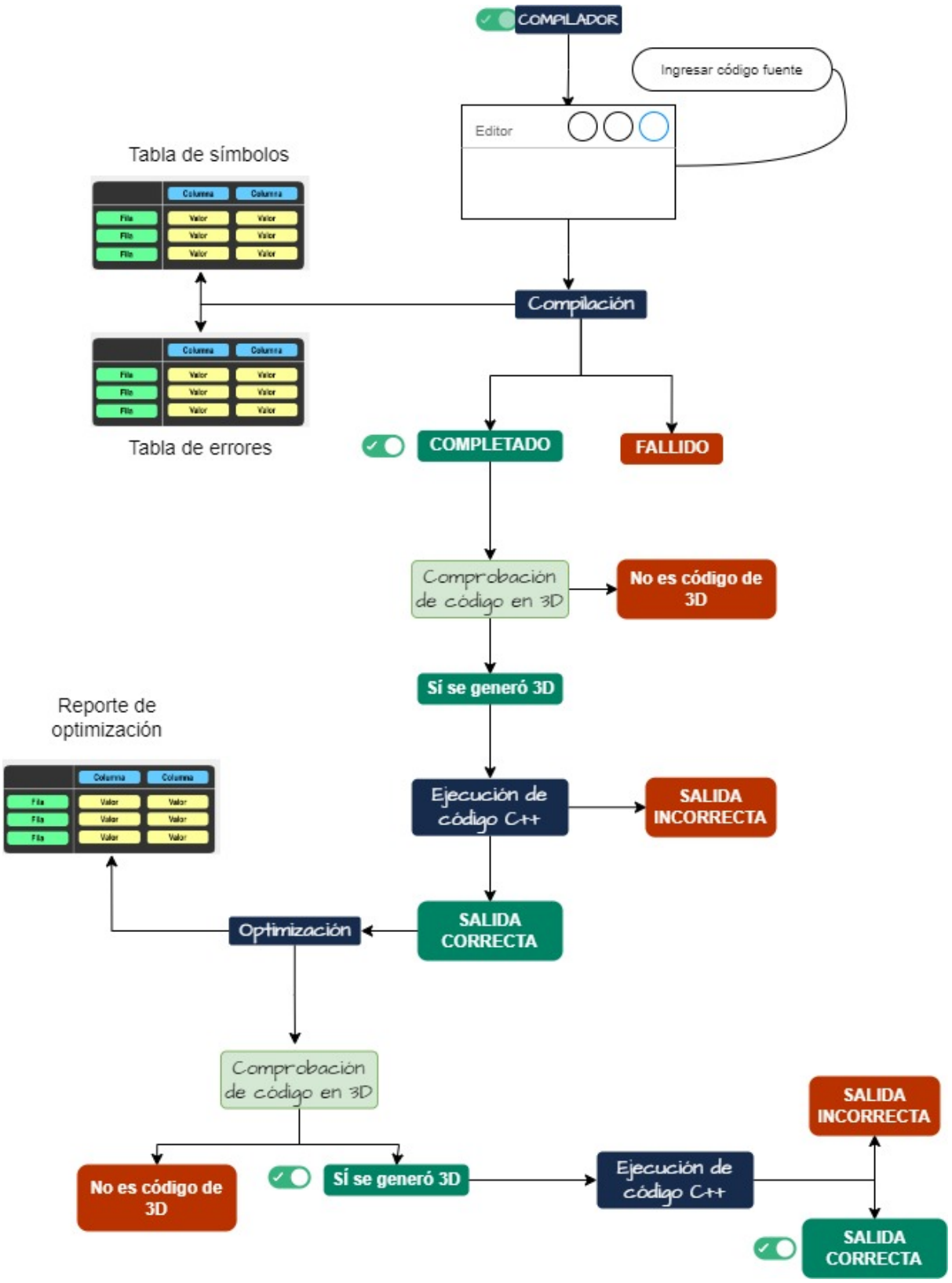

Ilustración 1: Flujo específico de la aplicación

La aplicación se compone de cuatro procesos importantes que son el proceso de compilación, optimización, comprobación y ejecución. Estos procesos se describen a continuación.

2.2.1. Proceso de compilación

El proceso de compilación recibe una entrada de código fuente en alto nivel generada por parte del usuario y así generar una salida que será una representación intermedia en formato de código de tres direcciones, este formato utilizará sentencias del lenguaje C para su posterior ejecución.

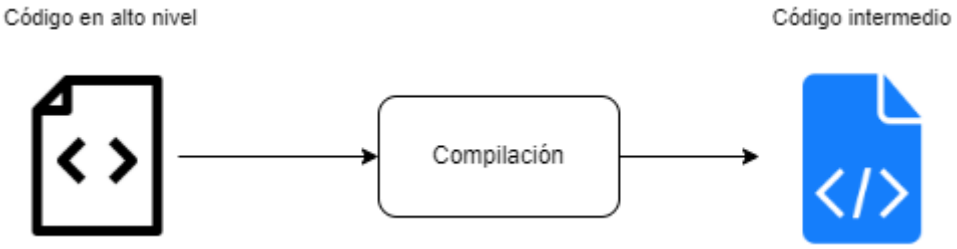

Ilustración 2. Proceso de Compilación.

2.2.2. Proceso de optimización

Un compilador puede introducir secuencias de código que contienen instrucciones innecesarias que consumen el tiempo de ejecución, por tal razón, se deberá aplicar transformaciones de optimización del código generado para producir código más eficiente. El proceso de optimización se debe de hacer por bloques.

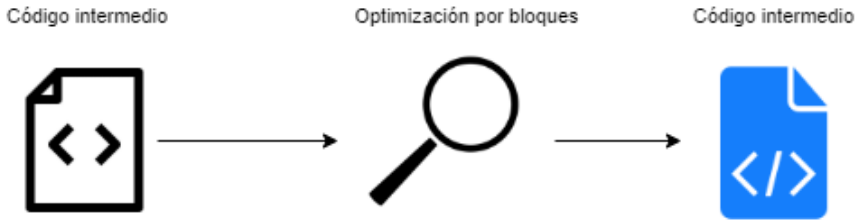

Ilustración 3. Proceso de Optimización.

2.2.3. Proceso de comprobación

Como el código de tres direcciones utiliza sentencias del lenguaje C, se debe asegurar que el compilador genere el formato correcto antes de su ejecución, por lo cual, se debe de realizar el proceso de comprobación después de generar el código de tres direcciones, después de la optimización por mirilla y después de la optimización por bloques. Así para cumplir con los objetivos del proyecto, por este motivo estará disponible un analizador de código de tres direcciones desarrollado por los tutores para constatar que el código generado y optimizado tenga la sintaxis correcta.

Ilustración 4. Proceso de Comprobación.

2.2.4. Proceso de ejecución

En el proceso de ejecución recibe como entrada el código de tres direcciones y será ejecutado en un compilador en línea del lenguaje de programación C, este proceso se puede realizar después del proceso de compilación y después de la optimización por bloques, se ejecutará el código únicamente si el código de tres direcciones tiene el formato correcto.

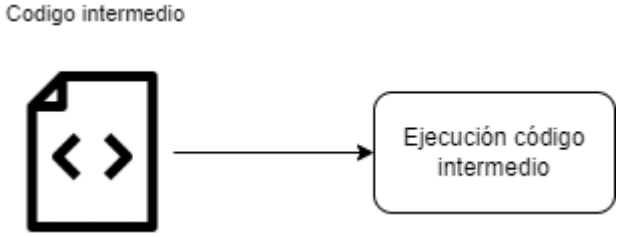

Ilustración 5. Proceso de Ejecución.

2.2.5. Visualización de reportes

Luego de haber finalizado el proceso de compilación el usuario podrá consultar el reporte de errores, tabla de símbolos y bases de datos. Asimismo, una vez finalizado el proceso de optimización, el usuario podrá consultar el reporte de optimización.

3. Componentes de la aplicación

3.1. Página de editor

DB Rust tendrá una página que contiene un editor de texto que recibirá como entrada el código fuente de alto nivel y así llevar a cabo el proceso de compilación mostrando como resultado el código de tres direcciones en la consola de salida. Y para los procesos de optimización, la entrada será el código de tres direcciones mostrando como resultado el código de tres direcciones optimizado en la consola de salida.

Para este editor no hace falta abrir archivos, basta con copiar y pegar la entrada.

Ilustración 6: Vista de editor

El editor contará con varias opciones que apoyen a la aplicación para realizar los procesos de compilación y optimización, estas opciones son:

- **Editor:** Redirecciona a la página de editor.
- **Compilador**
  - **Compilar**: Realizará el análisis del código fuente para generar el código de tres direcciones.
  - **Optimizar por mirilla**: Realizará la optimización por mirilla aplicando las nueve reglas descritas en la sección 7.1.

    - **Optimizar por bloques**: Realizará la optimización por bloques aplicando las cuatro reglas descritas en la sección 7.2.
- **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.

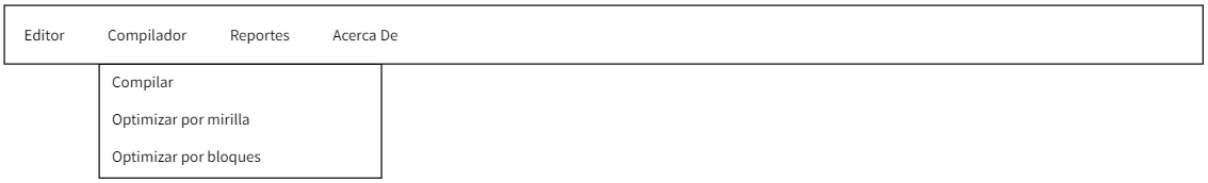

Ilustración 7: Opciones del compilador

3.2. Página de reportes

En este componente se podrán consultar los reportes de tabla de símbolos, tabla de errores y tabla de base de datos después de la compilación, además se puede consultar el reporte de optimización después de haber optimizado el código de tres direcciones.

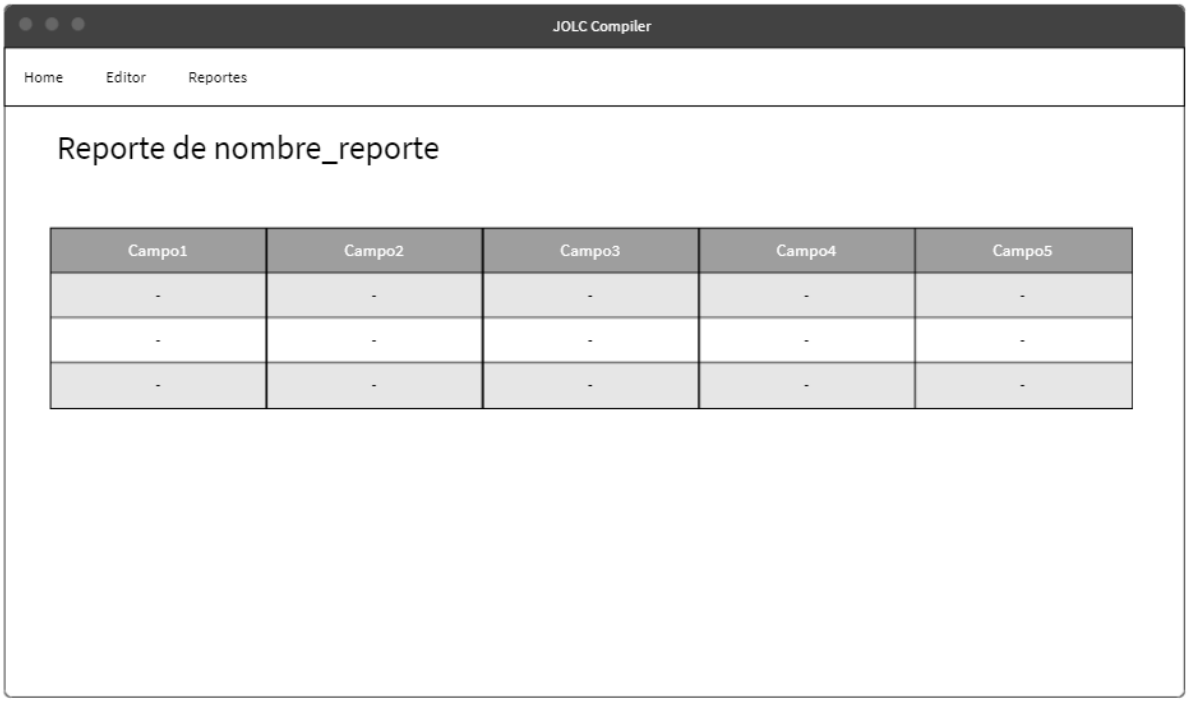

Ilustración 8: Página de consulta de reportes

Para mostrar un reporte en específico, la aplicación contará con tres opciones, se usará la vista de la ilustración 8 como plantilla para mostrar cada reporte:

- **Reporte de tabla de símbolos**: Este reporte mostrará la tabla de símbolos del código fuente generado en el proceso de compilación. Se espera que únicamente aparezcan los símbolos encontrados en el ámbito global y en los ámbitos locales.

- **Reporte de errores**: Este reporte mostrará los errores encontrados durante el proceso de compilación.
- **Reporte de base de datos existente**: Este mostrará todas las bases de datos (módulos padre) que haya encontrado dentro del código fuente.
- **Reporte de optimización**: Este reporte mostrará todas las optimizaciones que fueron posible realizar en el código de tres direcciones.

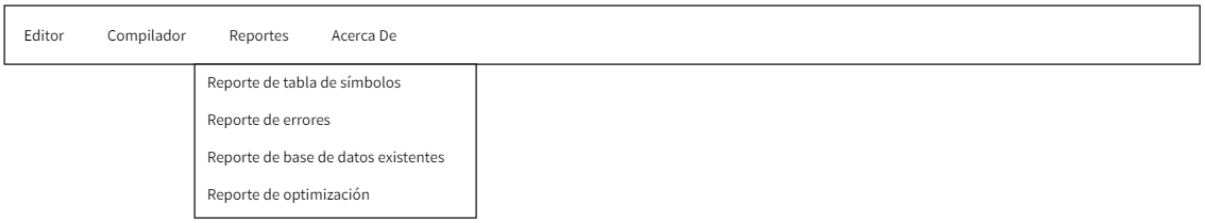

Ilustración 9: Opciones de reportes.

4. Sintaxis de DB-RUST

BD-RUST es la simulación de un pequeño sistema de base de datos, haciendo uso de la sintaxis de 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. Sistema de tipos

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

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

4.2.1. Tipos básicos

- **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.

4.2.2. Tipos compuestos

Los constructores de tipo utilizan los tipos básicos para crear nuevos tipos de datos. Los constructores de tipo que utilizan la mayoría de los lenguajes de programación son arreglos y estructuras.

Nombre	Descripción	Expresión de tipo
Arreglos	Conjunto de valores indexados entre 0 hasta n. Puede almacenar diferentes tipos. Para más información, consulte la sección 4.9.	Array
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 4.10.	Struct

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 explícito, 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	false	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 ID: TIPO = Expresión;
```

El sufijo **: TIPO** es obligatorio. Su función es asegurar que la expresión sea del tipo deseado.

```
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: i64 = 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: String = "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 igualando 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.

```
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.

```
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.

```
// 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.

```
// Ejemplo 1
// Contador del ciclo
let mut var1 = 0;
while var1 < 10 {
   println!("{}", 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;
    println!("{}", x);                // imprime 2-1-4-7-10
    if x % 5 == 0 {
        completado = true;
    }
}
```

4.8.3. For

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.

```
// 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
   println!("{ } ", 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
    println!("{ } -", letra);   // Imprime H-o-l-a- -M-u-n-d-o-!-
}
println!("");

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

// Ejemplo 5 - Es aplicable para arreglos o vectores
for letra in ["perro", "gato", "tortuga"] {
    println!("{} 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.

```
// 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;
   }
   println!("{ }", 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;
       }
       println!("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 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.

```
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.

```
// 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.

```
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.

```
// 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.

```
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.

```
// 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.

```
// 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.

```
// 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*.

```
// 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.

```
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.

```
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:

```
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.

```
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.

```
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.

```
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
}
```

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/EduardoAjsivinac/OLC2_2S2022_Generalidades , ahí estará publicado otro ejemplo con más detalles.

6. Generación de código Intermedio

El código intermedio es una representación intermedia del programa fuente que se ingresó en BD-RUST. Esta representación intermedia se realizará en código en tres direcciones, las cuales son secuencias de pasos de programa elementales.

En el proyecto se utilizarán las sentencias del lenguaje C para generar la correspondiente representación del código de alto nivel, en código intermedio. El código de salida ya no posee instrucciones y expresiones complejas, por lo que se debe implementar correctamente la traducción, para obtener el mismo resultado que obtenemos al escribir código en RUST.

No está permitido el uso de toda función o característica del lenguaje C, no descrita en este apartado. Se utilizará una herramienta de análisis para verificar que el código de tres direcciones generado tenga el formato correcto.

6.1. Tipos de dato

El lenguaje a compilar solo acepta tipos de datos numéricos, es decir, tipos int y float.
Consideraciones:

- No está permitido el uso de otros tipos de datos como cadenas o booleanos.
- El uso de arreglos no está permitido, únicamente para las estructuras heap y stack las cuales serán detalladas más adelante.
- Por facilidad, se recomienda trabajar todas las variables de tipo float, para no incurrir en problemas de casteo al momento de generar la correspondiente salida en 3D.

6.2. Temporales

Los temporales serán creados por el compilador en el proceso de generación de código de tres direcciones. Estas serán variables de tipo float. El identificador asociado a un temporal puede ser de la siguiente manera.

```
t[0-9]+
Ejejmplos:
t1
t145
```

6.3. Asignación a temporales

Los temporales son usados dentro del código intermedio para el manejo de las operaciones de cada instrucción, estos pueden recibir una operación aritmética, el valor de uno de los apuntadores o el valor dentro de la estructura Stack o Heap.

Las operaciones aritméticas para asignar a temporales contarán con:

- Resultado
- Argumento 1
- Operador
- Argumento 2

Operaciones aceptadas:

Operación	Símbolo	Ejemplo
Suma	+	t1=t0+1
Resta	-	t2=50-12
Multiplicación	*	t33=5*5
División	/	t67=4/1
Módulo	%	t44=4 % 2

6.4. Etiquetas

Las etiquetas son identificadores únicos que indican una posición en el código fuente, estas mismas serán creadas por el compilador en el proceso de generación de código en tres direcciones. El identificador asociado a una etiqueta puede ser de la siguiente manera.

```
L[0-9]+
Ejemplos:
L1
L21
```

Comentarios

Para llevar un mejor control de las instrucciones que se realizan dentro del código tres direcciones, se recomienda el uso de comentarios donde se podrá definir el flujo de cada bloque de código, los comentarios se definen de la siguiente manera:

- Comentarios de una línea. Inician con un conjunto de barras diagonales (//) y continúan hasta el final de la línea.
- Comentarios de múltiples líneas. Inician con los símbolos “/*” y finalizan con los símbolos “*/”

6.5. Saltos

Para definir el flujo que seguirá el programa se contará con bloques de código, estos bloques están definidos por etiquetas. La instrucción que indica que se realizará un salto hacia una etiqueta es la palabra reservada “goto”.

Para tener los mismos resultados en cuanto a las instrucciones en alto nivel, se deberá manejar correctamente los saltos entre los bloques generados. Los saltos se dividen en:

- **Condicionales**. Se realiza una evaluación para determinar si se realiza el salto.
- **Incondicionales**. Realiza el salto sin realizar una evaluación.

6.5.1. Saltos incondicionales

El formato de saltos incondicionales contará únicamente con una instrucción **goto** que indicara una etiqueta destino dentro del código, en la cual, se continúa con la ejecución del programa.

```
//Ejemplo de salto incondicional
     goto L1;
    printf("%c", 64); //código inalcanzable
L1:
     t2 = 100 + 5;
```

6.5.2. Saltos condicionales

El formato de los saltos condicionales utilizará la instrucción if del lenguaje C, donde se realizará un salto a una etiqueta donde se encuentre el código a ejecutar si la condición es verdadera, seguida de otro salto a una etiqueta donde están las instrucciones si la condición no se cumple.

Las instrucciones if tendrán como condición una expresión relacional, dichas expresiones se definen en la siguiente tabla:

Operación	Símbolo	Ejemplo
Menor que	<	t3<4
Mayor que	>	t6>44
Menor o igual que	<=	t55<=50
Mayor o igual que	>=	t99>=100
Igual que	==	t23==t44
Diferente que	!=	t34!=99

```
//Ejemplo de saltos condiciones
if (t1 > 10) goto L1;
goto L2;

L1:
    //código si la condición es verdadera
L2:
    //código si la condición es falsa
```

6.6. Asignación a temporales

La asignación nos va a permitir cambiar el valor de los temporales, para lograrlo se utiliza el operador igual, este permite una asignación directa o con una expresión.

```
//Entrado código alto nivel
println!("{}",1+2*5);

//Salida de código en tres direcciones en lenguaje C
t1 = 2 * 5;
t2 = 1 + t1;
printf("%d",(int)t2);
printf("%c",10);
```

6.7. Métodos

Estos son bloques de código a los cuales se accede únicamente con una llamada al método.

```
//Definición de métodos
void x() {
    goto L0;
    printf("%d", (int)100);
    L0:
    return;
}
```

Consideraciones:

- No está permitido el uso de parámetros en los métodos. Debe utilizar el stack para el paso de parámetros.
- Al final de cada método se debe incluir la instrucción “return”.

6.8. Llamada a métodos

Esta instrucción nos permite invocar a los métodos. Al finalizar su ejecución se retorna el control al punto donde fue llamada para continuar con las siguientes instrucciones.

```
void funcion1(){
    printf("%d", (int)100);
    return;
}

void main(){
    // INSTRUCCIONES DE MAIN
    funcion1();                    // Llamada a método function1
    // DESPUÉS DE EJECUTAR funcion1 REGRESA A MAIN
    return;
}
```

6.9. Impresión en consola

Su función principal es imprimir en consola un valor, el primer parámetro que recibe la función es el formato del valor a imprimir, y el segundo es el valor en sí.

La siguiente tabla lista los parámetros permitidos para el proyecto:

Parámetro	Acción
%c	Imprime el carácter del identificador, se basa según el código ASCII.
%d	Imprime valores enteros. El segundo parámetro debe ser una conversión explícita de int.
%f	Imprime valores con punto decimal.

```
printf("%d", (int)100);   // Imprime 100
printf("%c", 37);     // Imprime %
printf("%f", 32.2);  // Imprime 32.200000
```

6.10. Estructuras en tiempo de ejecución

En el código de tres direcciones no existen cadenas, operaciones complejas, llamadas a métodos con parámetros y otras características que sí están presentes en los lenguajes de alto nivel. Esto debido a que el código de tres direcciones busca acercarse al lenguaje máquina, siendo así el uso de las estructuras para el entorno de ejecución.

Las estructuras del entorno de ejecución son bloques de bytes contiguos que emplean mecanismos para emular la ejecución de instrucciones de alto nivel. Se utilizarán únicamente dos estructuras para este proyecto, la pila (**Stack**) y el montículo (**Heap**). Estas estructuras se utilizarán para almacenar los valores que sean necesarios durante la ejecución.

6.10.1. Stack

Es una estructura que soporta las llamadas de procedimientos o funciones, asignando espacio de memoria para variables locales, parámetros y retornos de un método en código de alto nivel.

Esta estructura utilizará un apuntador llamado “Stack Pointer”, que se identifica con el nombre **P**, este valor va cambiando conforme se ejecute el programa, y su manejo debe ser cuidadoso para no corromper espacios de memoria ajenos al método que se está ejecutando, su asignación se realizará de la misma manera que se realizan las asignaciones temporales.

```
float stack[10000];          // Stack
float P;                     // Stack pointer

int main() {
    P = P + 1;
    stack[(int)P] = 10;      // asignación
    t1 = stack[(int)P];      // acceso
    return 0;
}
```

6.10.2. Heap

Es una estructura del entorno de ejecución encargada de guardar las referencias a las cadenas, arreglos y estructuras. Esta estructura también cuenta con un apuntador que se identifica con el nombre **H**.

A diferencia del apuntador **P**, este apuntador no decrece, sino que sigue aumentando su valor, su función es apuntar a la primera posición de memoria libre dentro del heap.

```
float heap[10000];          // Heap
float H;                    // Heap pointer

int main() {
    H = H + 1;
    heap[(int)H] = 10;       // asignación
    t1 = heap[(int)H];       // acceso
    return 0;
}
```

Consideraciones:

- Al guardar cadenas, cada espacio debe ser ocupado por únicamente un carácter representado por su código ASCII.
- El heap solamente crece, nunca reutiliza espacios de memoria.

6.10.3. Acceso y asignación a estructuras en tiempo de ejecución

Para realizar las asignaciones y el acceso a estas estructuras, se debe respetar el formato de código de 3 direcciones:

- La asignación a las estructuras se debe realizar mediante su apuntador, temporales o valores constantes, no es permitido el uso de operaciones aritméticas o lógicas para la asignación a estas estructuras.
- No se permite la asignación a una estructura mediante el acceso a otra, por ejemplo “Stack[0] = Heap[100]”.
- Si el acceso a las estructuras se hace por medio del apuntador o temporales, se debe realizar un casteo, debido que los temporales pueden ser de tipo float.

```
//Asignación
heap[(int)H] = t1;
stack[(int)t2] = 150;
stack[10] = 250;

//Acceso
t10 = heap[(int)t10];
t20 = stack[(int)t1];
t30 = stack[100];
```

6.11. Encabezado

En esta sección se definirán todas las variables y estructuras a utilizar. Solo en esta sección se permite el uso de declaraciones, no se permite las declaraciones dentro de los métodos. El encabezado debe ser generado junto con el código de tres direcciones para hacer uso de los temporales y las estructuras. La estructura del encabezado es la siguiente:

```
#include <stdio.h>

float stack[10000];         // Stack
float heap[10000];          // Heap
float P;                    // Puntero Stack
float H;                    // Puntero Heap
float t1, t2, t3, t4;       // Temporales
```

Consideraciones:

- No es permitido el uso de otras librerías ajenas a “stdio”, en esta librería únicamente se usará la función de imprimir “printf”.
- Todas las declaraciones de temporales se deben encontrar en el encabezado
- El tamaño que se le asigne al stack y heap queda a discreción del estudiante. Tomar en cuenta que el heap únicamente aumenta, por lo que el tamaño de este debe ser grande.

6.12. Método main

Este es el método donde iniciará la ejecución del código traducido. Su estructura es la siguiente:

```
int main(){
     return 0;
}
```

6.13. Comprobación de código tres direcciones

Una vez generado el código tres direcciones este será ingresado a un analizador para corroborar que el código generado sea la correcta y no se encuentre código diferente al explicado anteriormente, una vez analizado se procederá a ejecutar el código en tres direcciones en un compilador de C para obtener el resultado esperado.

La herramienta que utilizarán los estudiantes para comprobar que estén generando el código de tres direcciones con la sintaxis correcta estará disponible a partir del 10 de octubre, los tutores serán los encargados de indicar el enlace donde estará publicado.

7. Optimización de código Intermedio

DB-Rust deberá aplicar transformaciones de optimización al código de tres direcciones como parte de su compilación para producir código más eficiente, estas optimizaciones se realizarán mediante optimización por bloques que se describe a continuación.

7.1. Optimización por bloque

Por lo regular, el código fuente tiene una serie de instrucciones que se ejecutan siempre en orden y están conformadas por bloques básicos del código.

Un bloque básico es una parte de código, en donde las instrucciones se ejecutan secuencialmente, por el cual, el flujo de control solo puede entrar en la primera instrucción y puede salir del bloque por una bifurcación o seguir con las siguientes instrucciones.

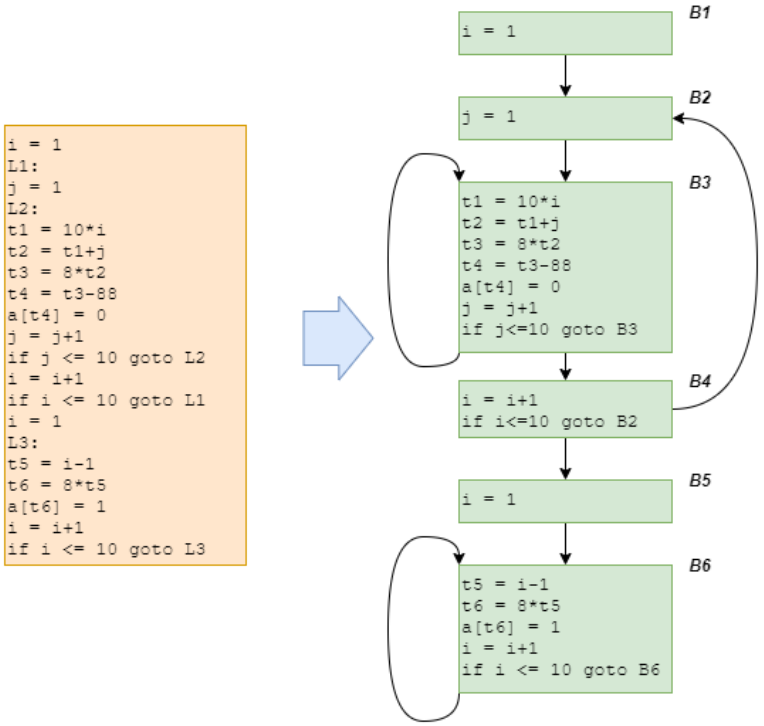

Ilustración 12. Grafo de bloques básicos.

Se debe de realizar la optimización sobre el código generado de las siguientes maneras:

- A nivel local: Consiste en optimizar el código de un bloque básico, se debe de realizar bloque por bloque.
- A nivel global: Consiste en optimizar el código de todos los bloques básicos, se debe de buscar candidatos posibles de optimización para aplicar las reglas.

Los tipos de transformación para realizar la optimización por bloque serán los siguientes:

- Subexpresiones comunes.
- Propagación de copias.
- Eliminación de código muerto.
- Propagación de constantes.

7.1.1. Subexpresiones comunes

7.1.1.1. Regla 1

Consiste en buscar expresiones que se repiten y analizar si vale la pena reemplazarlas o eliminarlas. Considerando el siguiente ejemplo, $t6$ tiene una subexpresión común con $t4$, por lo cual, se reemplaza $t6 = t2 * t3$ por la siguiente asignación $t6 = t4$, si los valores de la expresión de $t4$ no cambian.

Ejemplo	Optimización

t1 = 6 / 2;
t2 = 3 * t1;
t3 = 4 - 2;
t4 = t2 * t3;
t5 = 2 + 5;
t6 = t2 * t3;
	t1 = 6 / 2;
	t2 = 3 * t1;
	t3 = 4 - 2;
	t4 = t2 * t3;
	t5 = 2 + 5;
	t6 = t4;

7.1.2. Propagación de copias

7.1.2.1. Regla 2

Debido a que el algoritmo normal para eliminar subexpresiones comunes las introduce, es necesario aplicar está regla. La propagación de copias se relaciona con la asignación $u = v$, conocidas como instrucciones de copia o simplemente copias, la idea de la transformación por propagación de copias es utilizar v para u, siempre que sea posible después de la instrucción de copia $u = v$.

Ejemplo	Optimización

t1 = 6 / 2;
t2 = 3 * t1;
t3 = 4 - 2;
t4 = t2 * t3;
t5 = 2 + 5;
t6 = t4;
t7 = t6 + t5;
	t1 = 6 / 2;
	t2 = 3 * t1;
	t3 = 4 - 2;
	t4 = t2 * t3:
	t5 = 2 + 5;
	t6 = t4;
	t7 = t4 + t5;

7.1.3. Eliminación de código muerto

7.1.3.1. Regla 3

Una variable está viva en un punto en el programa, si su valor puede utilizarse más adelante; en caso contrario, está muerta en ese punto. La propagación de copias es que a menudo convierte la instrucción de copia en código muerto como es el caso de $t6$.

Ejemplo	Optimización

t1 = 6 / 2;
t2 = 3 * t1;
t3 = 4 - 2;
t4 = t2 * t3;
t5 = 2 + 5;
t6 = t4;
t7 = t6 + t5;
	t1 = 6 / 2;
	t2 = 3 * t1;
	t3 = 4 - 2;
	t4 = t2 * t3;
	t5 = 2 + 5;
	t7 = t4 + t5;

7.1.4. Propagación de constantes

7.1.4.1. Regla 4

Desde que se asigna a una variable un valor constante hasta la siguiente asignación, se considera a la variable equivalente a la constante.

Ejemplo	Optimización
t1 = 3.14; t2 = t1 / 180;	t2 = 3.14 / 180;

8. Reportes generales

8.1. Reporte de tabla de Símbolos

En este reporte se solicita mostrar la tabla de símbolos después de la compilación de la entrada. Se deberán mostrar todas las variables, funciones y struct reconocidas, junto con su tipo y toda la información que el estudiante considere necesaria. Este reporte al menos debe contener la fila y columna de la declaración del símbolo junto con su nombre, tipo y ámbito. En el caso de las funciones, deberá mostrar el nombre de sus parámetros, en caso tenga.

Nombre	Tipo	Ámbito	Fila	Columna
x		valores	2	18
valores	Función	Global	2	1
arr	arreglo	Global	6	1
x	arreglo	Global	7	1

8.2. Reporte de tabla de errores

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

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

8.3. Reporte 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.
- Línea y columna en que se encontró la base de datos.

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

8.4. Reporte de optimización

Este reporte mostrará las reglas de optimización que fueron aplicadas sobre el código intermedio. Se debe indicar el tipo de optimización utilizada y la sección. Como mínimo se solicita la siguiente información:

- Tipo de optimización (Mirilla o por bloques)
- Regla de optimización aplicada
- Expresión original
- Expresión optimizada
- Fila

9. Manejo de errores

9.1. Errores semánticos

El compilador deberá ser capaz de detectar todos los errores semánticos que se encuentren durante el proceso de compilación. Todos los errores se deberán de recolectar y se mostrará un reporte de errores antes mencionado.

Un error semántico es cuando la sintaxis es la correcta, pero la lógica no es la que se pretendía, por eso, la recuperación de errores semánticos será de ignorar y reportar la instrucción en donde se generó el error. En la siguiente tabla se muestra los errores que puede encontrar el estudiante en todo el análisis:

No.	Validación	Ejemplo	Error
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

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; ✘ let x: f64 = 3.5;	Variable x ya declarada

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”,
		_ => “otra cosa”,
		};
			persona no es un tipo válido para un match

17	Validar que exista una o más definiciones de módulos	// sin módulos	No se encontró ninguna definición de módulo.

9.2. Comprobación dinámica

Existen ciertos casos en los que no es posible comprobar la validez de operaciones en tiempo de compilación, solamente en tiempo de ejecución. En el proyecto se tomarán en cuenta los siguientes casos:

División entre cero

Se deberá de realizar la comprobación de división entre cero siempre y cuando se realicen expresiones aritméticas con el operador de división (/) o el operador de módulo (%). Se deberá realizar la verificación en código de tres direcciones mostrando como mensaje “MathError”. Por ejemplo:

Entrada	Salida

`let a: i64=(55+3)/(3-3);`	`T1 = 55 + 3;`
	`T2 = 3 - 3;`
	`if (T2 != 0) goto L1;`
	`Printf("%c", 77);  //M`
	`Printf("%c", 97);  //a`
	`Printf("%c", 116); //t`
	`Printf("%c", 104); //h`
	`Printf("%c", 69);  //E`
	`Printf("%c", 114); //r`
	`Printf("%c", 114); //r`
	`Printf("%c", 111); //o`
	`Printf("%c", 114); //r`
	`T3 = 0;  // resultado incorrecto`
	`goto L2;`
	`L1:`
	`T3 = T1 / T2;  // resultado correcto`
	`L2:`

Índice fuera de los límites

Se deberá realizar la comprobación de índice fuera de los límites, tanto superior como inferior, siempre que se realice un acceso a un arreglo. Se deberá realizar la verificación en código de tres direcciones mostrando como mensaje “BoundsError”. Por ejemplo:

Entrada	Salida

`let mut numeros: [i64; 3] = [1,2,3];`
`numeros[10]=44;`
	`T2 = 10;  // índice al que desea acceder`
	`if (T2 < 1) goto L1;`
	`// 1 es el límite inferior del arreglo`
	`if (T2 > 3) goto L1;`
	`// 3 es el límite superior del arreglo`
	`goto L2;`
	`L1:`
	`Printf("%c", 66);  //B`
	`Printf("%c", 111); //o`
	`Printf("%c", 117); //u`
	`Printf("%c", 110); //n`
	`Printf("%c", 100); //d`
	`Printf("%c", 115); //s`
	`Printf("%c", 69);  //E`
	`Printf("%c", 114); //r`
	`Printf("%c", 114); //r`
	`Printf("%c", 111); //o`
	`Printf("%c", 114); //r`
	`// No continúa con la instrucción`
	`goto L3;`
	`L2:`
	`// Continúa con la instrucción`
	`L3:`

Consideraciones:

- En caso se produzca un error al intentar ejecutar una instrucción, esta se debe omitir y continuar con la siguiente instrucción. En caso se produzca un error en una expresión, esta debe resultar con valor 0. Todo esto luego de imprimir en consola el texto solicitado según el caso.

10. 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.

10.1. 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:
    - alexYovani53
    - EduardoAjsivinac
- Interfaz gráfico para el manejo de la aplicación.

10.2. Restricciones

- La herramienta para generar los analizadores del proyecto será PLY de python. La documentación se encuentra en el siguiente enlace https://www.dabeaz.com/ply/
- 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.

10.3. Consideraciones

- Es válido el uso de cualquier librería de **Python** 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/EduardoAjsivinac/OLC2_2S2022_Generalidades .
- 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.

10.4. 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.
- En el proyecto se calificará la simulación de base de datos, es decir, todas las instrucciones relacionadas a la simulación.
- 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.
- Durante la calificación se realizará un análisis en una herramienta desarrollada por los tutores, para verificar que se utilicen únicamente las instrucciones definidas en este enunciado. La herramienta la pueden encontrar en: https://manuelmiranda99.github.io/Analizador_C3D/
- El uso de la herramienta es para garantizar que no se usen instrucciones, que no estén comprendidas en el enunciado. Por ejemplo.

<table>
<tr><td colspan="2">Entrada

println!(“Hello World !”);</td></tr>
<tr><td>NO ES CD3</td><td>printf(“Hello World”);</td></tr>
<tr><td>ESTO SI ES C3D</td><td>#include <stdio.h>
float stack[100000];
float heap[100000];
float P;
float H;
float t1, t2, t3, t4, t5, t6, t7, t8;

void imprimir () {
t1 = stack[(int)P];
t2 = heap[(int)t1];
t3 = - 1;
L1:
if (t2 == t3) goto L2;
printf("%c", (int)t2);
t1 = t1 + 1;
t2 = heap[(int)t1];
goto L1;
L2:
return;
}

int main()
{

t7 = H;
heap[(int)H] = 72;
H = H + 1;
heap[(int)H] = 101;
H = H + 1;</td></tr>
</table>

```
        heap[(int)H] = 108;
        H = H + 1;
        heap[(int)H] = 108;
        H = H + 1;
        heap[(int)H] = 111;
        H = H + 1;
        heap[(int)H] = 32;
        H = H + 1;
        heap[(int)H] = 87;
        H = H + 1;
        heap[(int)H] = 111;
        H = H + 1;
        heap[(int)H] = 114;
        H = H + 1;
        heap[(int)H] = 108;
        H = H + 1;
        heap[(int)H] = 100;
        H = H + 1;
        heap[(int)H] = 32;
        H = H + 1;
        heap[(int)H] = 33;
        H = H + 1;
        heap[(int)H] = - 1;
        H = H + 1;
        P = P + 0;
        stack[(int)P] = t7;
        imprimir();
        P = P - 0;
        printf("%c", (int)10);
        printf("%c", (int)13);

        return 0;
}
```

10.5. 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.
- La fecha de entrega será el miércoles 26 de octubre, tendrán un lapso de 12 horas para realizar la entrega del proyecto, el proceso de entrega inicia a las 8:00 y

termina a las 20:00 horas. Cualquier inconveniente notificar a su tutor para poder brindar apoyo para el proceso.

Miércoles 26 de octubre en horario de 8:00 a 20:00