Compiladores: Proyecto Final

Instrucciones

1. El objetivo del proyecto es desarrollar un compilador que genere código en lenguaje ensamblador x86-64 para un lenguaje de programación previamente asignado.

2. Los grupos deberán estar conformados por un máximo de tres integrantes.

3. El entregable final estará compuesto por los siguientes elementos:

   - Código fuente en C++, estructurado en los módulos: `scanner`, `parser`, `visitors` y casos de prueba (`test cases`).
   - Presentación oral del proyecto.
   - Reporte técnico que documente la arquitectura, las decisiones de diseño y las pruebas realizadas. En este reporte puede adjuntar el link del repositorio del proyecto.

4. Todo grupo que no respete la estructura exigida recibirá una calificación de cero.

5. Requisitos del código:

   - Cada grupo deberá seleccionar un lenguaje compilado (por ejemplo: C, Pascal, Rust, Kotlin u otro).
   - Deberán implementar un compilador base para dicho lenguaje que incluya, como mínimo, las siguientes funcionalidades:
     - Declaración de variables.
     - Operaciones aritméticas y de relación.
     - Sentencias de control `if` y `while`.
     - Definición y uso de funciones.

     Podrán revisar los **anexos**, donde se presentan ejemplos de implementación correspondientes a los lenguajes `Rust`, `Pascal`, `Kotlin` y `C`, que ilustran las funcionalidades requeridas en el compilador base.
   - Posteriormente, cada grupo deberá desarrollar la implementación adicional asignada según las indicaciones del proyecto.
   - Luego, deberán incorporar dos extensiones seleccionadas de la lista establecida.
   - Además, cada grupo deberá presentar dos herramientas de optimización aplicadas al proceso de compilación o a la generación de código ensamblador.
   - Solo se otorgará puntaje si el compilador genera código ensamblador completamente funcional que incluya la implementación adicional asignada.

6. Implementación adicional:

N°	Nombre de la propuesta	Características principales
1	Tipos numéricos	Float type, Unsigned int, Long int
2	Estructuras de datos compuestas	Struct, Arrays, Strings
3	Gestión de memoria y acceso indirecto	Memoria dinámica, Punteros, Arrays
4	Tipos primitivos y estructuras	Unsigned int, Float type, Struct
5	Cadenas, arreglos y memoria dinámica	Strings, Arrays, Memoria dinámica
6	Punteros, estructuras y arrays	Punteros, Struct, Arrays
7	Manejo de tipos y coerciones	Float type, Long int, Punteros
8	Modelado de estructuras dinámicas	Memoria dinámica, Struct, Punteros

7. Extensiones disponibles:
   - Arreglos multidimensionales
   - Conversión y promoción de tipos
   - Inferencia de tipos
   - Tipos genéricos y plantillas
   - Alias de tipo (`typedef`)
   - Expresiones condicionales (ternarias)
   - Sobrecarga de operadores
   - Parámetros por referencia
   - Retorno de estructuras o arreglos
   - Funciones `lambda`
   - Recolección de basura básica
8. Cada grupo deberá desarrollar los siguientes casos de prueba:
   - 3 casos de funciones.
   - 5 casos para la implementación base.
   - 5 casos para las extensiones.
   - 5 casos para optimización.
9. El reporte y la presentación deberán enfocarse exclusivamente en las extensiones implementadas e incluir:
   - Clases elaboradas
   - Operaciones implementadas
   - Estructura de los `visitors` y diseño del compilador

10. Rúbrica de evaluación (20 puntos):

Implementación base	6 pts
Extensiones	5 pts
Optimización	3 pts
Reporte	3 pts
Presentación oral	3 pts

11. La fecha de entrega es el miércoles 26 de noviembre, hasta las 11:59 p.m.

12. Las presentaciones tendrán una duración total de 30 minutos: 20 minutos de exposición y 10 minutos destinados a preguntas.

13. Si algún grupo desea implementar una app para su compilador, que incluya una herramienta interactiva para visualizar el *estado de los registros* y la *pila de ejecución* paso a paso, se otorgará un **bonus de hasta 3 puntos** a cada integrante en la calificación del **Examen 3**.

A. Ejemplos sencillos

A.1. Lenguaje Pascal

A.1.1. Ejemplo 1

```
program Example;
var
x: integer;
y: integer;
z: longint;
begin
x := 1;
y := 10;
z := 1000000;
x := 20;
writeln(x);
writeln(y);
writeln(z);
end.
```

A.1.2. Ejemplo 2

```
program ExampleProgram;
var
x, y: Integer;
begin
    x := 5;
    y := 10;
    if x > y then
    begin
        writeln(x);
    end
    else
    begin
        writeln(y);
    end;
end.
```

A.1.3. Ejemplo 3

```
program ExampleProgram;
var
    x: Integer;
    i: Integer;
begin
    x := 1;
    for i := 0 to 9 do
    begin
        x := x + i;
    end;
    writeln(x);
end.
```

A.1.4. Ejemplo 4

```
program ExampleProgram;

function suma(a, b: Integer): Integer;
begin
    suma := a + b;
end;
var
    x, y: Integer;
begin
    x := 1;
    y := 20;
    writeln(suma(x, y));
end.
```

A.2. Lenguaje Rust

A.2.1. Ejemplo 1

```
fn main() {
    let mut x: i32;
    let mut y: i32;
    let mut z: i64;
    x = 1;
    y = 10;
    z = 1000000;
    x = 20;
    println!("{}", x);
    println!("{}", y);
    println!("{}", z);
}
```

A.2.2. Ejemplo 2

```
fn main() {
    let mut x: i32;
    let mut y: i32;
    x = 5;
    y = 10;
    if x > y
    {
        println!("{}", x);
    }
    else
    {
        println!("{}", y);
    }
}
```

A.2.3. Ejemplo 3

```
fn main() {
    let mut x: i32;
    x = 1;
    for i in 0..10
    {
        x += i;
    }
    println!("{}", x);
}
```

A.2.4. Ejemplo 4

```
fn suma(a: i32, b: i32) -> i32 {
    a + b
}

fn main() {
    let mut x: i32;
    let mut y: i32;
    x = 1;
    y = 20;
    println!("{}", suma(x, y));
}
```

A.3. Lenguaje Kotlin

A.3.1. Ejemplo 1

```
fun main() {
    var x: Int
    val y: Int
    val z: Long
    x = 1;
    y = 10;
    z = 1000000;
    x = 20;
    println(x)
    println(y)
    println(z)
}
```

A.3.2. Ejemplo 2

```
fun main() {
    var x: Int
    val y: Int
    x = 5;
    y = 10;
    if (x > y) {
        println(x)
    } else {
        println(y)
    }
}
```

A.3.3. Ejemplo 3

```
fun main() {
    var x: Int
    x = 1
    for (i in 0..9) {
        x = x + i
    }
    println(x)
}
```

A.3.4. Ejemplo 4

```
fun suma(a: Int, b: Int): Int {
    return a + b
}
fun main() {
    var x: Int
    var y: Int
    x = 1
    y = 20
    println(suma(x, y))
}
```

A.4. Lenguaje C

A.4.1. Ejemplo 1

```
#include<stdio.h>
int main(){
    int x;
    int y;
    long z;
    x = 1;
    y = 10;
    z = 1000000;
    x = 20;
    printf("%d\n", x);
    printf("%d\n", y);
    printf("%ld\n", z);
    return 0;
}
```

A.4.2. Ejemplo 2

```
#include<stdio.h>
int main(){
    int x;
    int y;
    x = 5;
    y = 10;
    if (x > y){
        printf("%d\n", x);
    } else {
        printf("%d\n", y);
    }
    return 0;
}
```

A.4.3. Ejemplo 3

```c
#include<stdio.h>
int main(){
    int x;
    x = 1;
    for (int i = 0; i < 10; i++){
        x = x + i;
    }
    printf("%d\n", x);
    return 0;
}
```

A.4.4. Ejemplo 4

```c
#include<stdio.h>
int suma(int a, int b){
    return a + b;
}
int main(){
    int x;
    int y;
    x = 1;
    y = 20;
    printf("%d\n", suma(x, y));
    return 0;
}
```

B. Extensiones

- **Arreglos multidimensionales**

```c
int matriz[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
printf("%d", matriz[1][2]); // Imprime 6
```

Listing 1: Arreglos multidimensionales en C

- **Conversión y promoción de tipos**

```rust
let x: i32 = 5;
let y: f64 = x as f64 + 2.5; // Conversin explcita
println!("{}", y);
```

Listing 2: Conversión y promoción de tipos en Rust

- **Inferencia de tipos**

```kotlin
val x = 10      // Tipo inferido como Int
val y = 3.14    // Tipo inferido como Double
println(x + y)
```

Listing 3: Inferencia de tipos en Kotlin

- **Tipos genéricos y plantillas**

```cpp
template<typename T>
T suma(T a, T b) {
    return a + b;
}
int main() {
    printf("%d\n", suma(3, 4));   // Enteros
    printf("%.2f\n", suma(2.5, 3.5)); // Flotantes
}
```

Listing 4: Plantillas en C++

- **Alias de tipo (`typedef`)**

```c
typedef unsigned long ulong;
ulong n = 1000000;
printf("%lu", n);
```

Listing 5: Alias de tipo en C

- **Expresiones condicionales (ternarias)**

```c
int x = 5, y = 10;
int max = (x > y) ? x : y;
printf("Max: %d", max);
```

Listing 6: Expresión condicional en C

- **Sobrecarga de operadores**

```rust
use std::ops::Add;

struct Punto { x: i32, y: i32 }

impl Add for Punto {
    type Output = Punto;
    fn add(self, other: Punto) -> Punto {
        Punto { x: self.x + other.x, y: self.y + other.y }
    }
}

fn main() {
    let p1 = Punto { x: 1, y: 2 };
    let p2 = Punto { x: 3, y: 4 };
    let p3 = p1 + p2;
    println!("({}, {})", p3.x, p3.y);
}
```

Listing 7: Sobrecarga de operadores en Rust

- **Parámetros por referencia**

```pascal
program Referencia;
procedure Incrementar(var x: integer);
begin
    x := x + 1;
end;
var a: integer;
begin
    a := 5;
    Incrementar(a);
    writeln(a); // Imprime 6
end.
```

Listing 8: Parámetros por referencia en Pascal

- **Retorno de estructuras o arreglos**

```c
typedef struct {
    int x, y;
} Punto;

```

```
Punto crearPunto(int a, int b) {
    Punto p = {a, b};
    return p;
}

int main() {
    Punto q = crearPunto(3, 4);
    printf("(%d, %d)", q.x, q.y);
}
```

Listing 9: Retorno de estructuras en C

- **Funciones** `lambda`

```
val cuadrado = { x: Int -> x * x }
println(cuadrado(5)) // Imprime 25
```

Listing 10: Funciones lambda en Kotlin

- **Recolección de basura básica**

```
fn main() {
    let s = String::from("Hola");
    let t = s; // s se mueve, Rust libera automticamente cuando t sale
        ↪ del alcance
    println!("{}", t);
}
```

Listing 11: Recolección de basura (ownership) en Rust