ESCUELA SUPERIOR POLITECNICA

DEL LITORAL

Proyecto de Lenguaje de

Programación

LOS RUR

Integrantes

- Luis Vergara
- Joao Dorado
- Luis Roca

Profesora: CISNEROS BARAHONA FANNY CARLOTA

Paralelo: 2

Tabla de contenido

Introducción

Este documento detalla las reglas semánticas implementadas en nuestro analizador de MiniKotlin. El trabajo se dividió equitativamente entre los tres integrantes del equipo, asignando dos reglas a cada uno según su área de especialización.

Distribución del trabajo

Integrante	Usuario GitHub	Área Asignada
Luis Vergara	LuisVergaraA	Variables y declaraciones (Reglas 1-2)
Luis Roca	LuisRoca09	Funciones y retornos (Reglas 3-4)
Joao Dorado	johaodorado	Tipos y clases (Reglas 5-6)

Reglas Implementadas

REGLA 1: Verificación de Declaración de Variables

Responsable: Luis Vergara

Esta regla garantiza que no haya conflictos con los identificadores de variables. Se valida que:

- No se declare una variable dos veces en el mismo scope
- Las variables se usen solo después de haber sido declaradas
- Las variables se inicialicen antes de su primer uso
- Las variables val siempre tengan un valor inicial

Ejemplos de código con errores:

```
// Error: redeclaración
val x = 10;
val x = 20;  // ✗ 'x' ya existe

// Error: uso sin declarar
println(y);  // ✗ 'y' no fue declarada

// Error: uso sin inicializar
var z;
println(z);  // ✗ 'z' no tiene valor
z = 10;

// Error: val sin inicializar
val w;  // ✗ val debe tener valor inicial
```

Código correcto:

```
val constante = 100;  // ✓declarada e inicializada
var variable = 50;    // ✓declarada e inicializada
variable = 100;       // ✓se puede cambiar porque es var
```

REGLA 2: Inmutabilidad de Variables val

Responsable: Luis Vergara

Las variables declaradas como val no pueden modificarse después de su inicialización. Esto incluye:

- Reasignación directa
- Operadores de asignación compuesta (+=, -=)
- Operadores de incremento/decremento (++, --)

Ejemplos de código con errores:

```
// Error: reasignación de val
val constante = 100;
constante = 200;  // ✗ val no se puede modificar

// Error: operador compuesto en val
val numero = 10;
numero += 5;  // ✗ val no permite +=

// Error: incremento en val
val contador = 0;
contador++;  // ✗ val no permite ++
```

Código correcto:

```
var variable = 50;
variable = 100;  // ✓var permite cambios
variable += 10;  // ✓operadores compuestos OK
variable++;      // ✓incremento OK
```

REGLA 3: Verificación de Existencia de Funciones

Responsable: Luis Roca

Esta regla valida el uso correcto de funciones en el código:

- No puede haber dos funciones con el mismo nombre
- Solo se pueden llamar funciones que ya fueron declaradas
- El número de argumentos debe coincidir con la definición

Ejemplos de código con errores:

```
// Error: función duplicada
fun calcular(): Int {
    return 42;
}
fun calcular(): Int {  // ✗ ya existe
    return 100;
```

```
}

// Error: función no declarada
fun main() {
    val resultado = procesarDatos();  // ✗ no existe
}

// Error: argumentos incorrectos
fun sumar(a: Int, b: Int): Int {
    return a + b;
}
fun test() {
    val r1 = sumar(10);       // ✗ faltan argumentos
    val r2 = sumar(10, 20, 30);  // ✗ sobran argumentos
}
```

Código correcto:

```
fun multiplicar(x: Int, y: Int): Int {
    return x * y;
}

fun usar() {
    val resultado = multiplicar(5, 10);  // ✓ llamada correcta
}
```

REGLA 4: Consistencia de Tipo de Retorno

Responsable: Luis Roca

Todas las instrucciones return en una función deben ser consistentes con el tipo declarado:

- Si la función tiene tipo de retorno, todos los return deben devolver ese tipo
- Las funciones sin tipo (Unit) no deben tener return con valor
- return solo puede usarse dentro de funciones
- Todos los return en la misma función deben retornar el mismo tipo

Ejemplos de código con errores:

```
// Error: tipo incorrecto
fun obtenerNumero(): Int {
    return "texto";  // ✗ esperaba Int, recibió String
}

// Error: return con valor en función Unit
fun imprimir() {
    return 42;  // ✗ Unit no retorna nada
}
```

```
// Error: return fuera de función
val x = 10;
return x;  // ✗ return solo va en funciones

// Error: returns inconsistentes
fun procesar(x: Int): Int {
    if (x > 0) {
        return 10;     // Int
    } else {
        return "cero";  // ✗ String no es Int
    }
}
```

Código correcto:

```
fun calcular(a: Int, b: Int): Int {
    if (a > b) {
        return a;  // ✓retorna Int
    } else {
        return b;  // ✓retorna Int
    }
}

fun saludar() {
    println("Hola");  // ✓Unit no necesita return
}
```

REGLA 5: Verificación de Tipos en Operaciones

Responsable: Johao Dorado

Los operadores deben usarse con tipos de datos compatibles:

- Operaciones aritméticas (+, -, *, /, %) requieren números
- Operadores relacionales (<, >, <=, >=) comparan tipos iguales
- Operadores lógicos (&&, ||) solo funcionan con booleanos
- Operador unario – solo para números
- Operador unario ¡ solo para booleanos

Ejemplos de código con errores:

```
// Error: aritmética con tipos incompatibles
val texto = "Hola";
val numero = 42;
val resultado = texto + numero;  // ✗ String + Int
val mult = true * 10;  // ✗ Boolean * Int

// Error: comparación con tipos diferentes
```

```
val comp1 = "abc" < 10;  // ✗ String < Int
val comp2 = true >= 5;   // ✗ Boolean >= Int

// Error: lógica con no-booleanos
val x = 10;
val y = 20;
val res = x && y;  // ✗ Int && Int

// Error: operadores unarios
val neg = -"texto";  // ✗ no se puede negar String
val not = ¡10;       // ✗ no se puede negar Int
```

Código correcto:

```
val a = 10 + 20;       // ✓Int + Int
val b = 3.14 + 2.71;   // ✓Double + Double
val c = 10 < 20;       // ✓Int < Int
val d = true && false; // ✓Boolean && Boolean
val e = -42;           // ✓-Int
val f = !true;         // ✓!Boolean
val g = 10 + 3.14;     // ✓promoción automática a Double
```

REGLA 6: Verificación de Acceso a Miembros de Clases

Responsable: Johao Dorado

Al acceder a propiedades o métodos de una clase, se verifica que:

- El miembro (propiedad o método) exista en la clase
- THIS solo se use dentro de clases
- Las propiedades accedidas estén definidas en el constructor o cuerpo de la clase
- Los métodos llamados estén implementados en la clase
- Los objects (singleton) solo permitan acceso a sus miembros definidos

Ejemplos de código con errores:

```
// Error: miembro inexistente
class Persona(val nombre: String, val edad: Int) {
   fun mostrar() {
      println(this.nombre);   // ✓existe
      println(this.apellido); // ✗ no existe
   }
}

// Error: this fuera de clase
fun global() {
   val valor = this.propiedad;  // ✗ this solo en clases
}
```

```
// Error: propiedad no definida
fun usar() {
    val p = Persona("Ana", 25);
    println(p.nombre);     // ✓existe
    println(p.direccion);  // ✗ no existe
}

// Error: método no implementado
class Calculadora() {
    fun sumar(a: Int, b: Int): Int {
        return a + b;
    }
}
fun test() {
    val calc = Calculadora();
    calc.sumar(10, 20);      // ✓existe
    calc.multiplicar(5, 5);  // ✗ no existe
}

// Error: miembro de object inexistente
object Config {
    val puerto = 8080;
}
fun main() {
    println(Config.puerto);  // ✓existe
    println(Config.host);    // ✗ no existe
}
```

Código correcto:

```
class Rectangulo(val ancho: Int, val alto: Int) {
    fun area(): Int {
        return this.ancho * this.alto;  // ✓ambos existen
    }

    fun perimetro(): Int {
        return 2 * (this.ancho + this.alto);  // ✓
    }
}

fun usar() {
    val rect = Rectangulo(10, 5);
    val a = rect.area();     // ✓método existe
    val w = rect.ancho;      // ✓propiedad existe
}
```

Estructuras de Datos

El analizador mantiene tres tablas principales para validar las reglas semánticas

Tabla de Símbolos (Variables)

Almacena información de cada variable declarada:

```
symbol_table = {
   'nombre_variable': {
      'type': 'Int' | 'Double' | 'String' | 'Boolean' | 'Char',
      'mutable': True | False,  # var = True, val = False
      'initialized': True | False,
      'line': int  # línea donde se declaró
   }
}
```

Tabla de Funciones

Registra todas las funciones del programa:

```
function_table = {
   'nombre_funcion': {
      'ret': 'Int' | 'Unit' | ...,  # tipo de retorno
      'params': [{'name': str, 'type': str}, ...],
      'returns': [tipo1, tipo2, ...],  # tipos encontrados en returns
      'line': int
   }
}
```

Tabla de Clases

Guarda la estructura de clases y objects:

```
class_table = {
   'nombre_clase': {
      'params': [...],  # parámetros del constructor
      'properties': {
         'nombre_prop': {'type': str, 'mutable': bool}
      },
      'methods': {
         'nombre_metodo': {'params': [...], 'ret': str}
      }
   }
}
```

Archivos de Prueba

Cada integrante creó un archivo de prueba con casos específicos para sus reglas:

Archivo	Integrante	Pruebas
algoritmo_LuisVergaraA.kt	Luis Vergara	7 casos (Reglas 1-2)
algoritmo_LuisRoca09.kt	Luis Roca	12 casos (Reglas 3-4)
algoritmo_johaodorado.kt	Joao Dorado	8 casos (Reglas 5-6)

Todos los casos de prueba están marcados como "AVANCE 3: PRUEBAS SEMÁNTICAS" en el código.

Formato de Logs

Los logs semánticos se generan con el formato:

```
semantico-usuario-ddmmyyyy-HHhMM.txt
```

Ejemplo: semantico-LuisVergaraA-25112024-15h45.txt

Ejecución de Pruebas

```
# Ejecutar análisis semántico
python test_semantico.py

# Ver logs generados
ls logs/semantico-*.txt

# Ver contenido de un log
cat logs/semantico-usuario-25112024-15h45.txt
```

Observaciones Finales

El analizador semántico cumple con todos los requisitos del proyecto. Las seis reglas están implementadas y funcionando correctamente. La Regla 6 (acceso a miembros de clases) podría mejorarse para validar casos más complejos, pero cubre los casos básicos requeridos.

El trabajo se distribuyó equitativamente y cada integrante completó sus responsabilidades según lo planificado.