Reglas de Tipado y Tabla de Símbolos para el Analizador Semantico de C-

Este documento resume **las reglas de inferencia de tipos** que se usaron para construir el **analizador semántico** del lenguaje C-, y explica cómo está organizada la **tabla de símbolos**.

Reglas de Tipado:

El analizador semántico se encarga de revisar que el uso de variables, operaciones y funciones tenga **sentido en cuanto a tipos**. Estas son las reglas que fueron utilizadas:

1. Constantes

Cualquier número como `10` o `3` es reconocido como tipo `int`.
Si ves un número entero, es un entero.

---

2. Variables y Parámetros

Con las variables el compilador va a la **tabla de símbolos** a ver qué tipo tiene. Si fue declarada como `int`, pues es `int`. Si no fue declarada, se marca como error.

---

3. Operaciones Matemáticas (`+`, `-`, `*`, `/`)

Para que una suma, resta, multiplicación o división sea válida, **los dos operandos deben ser enteros**. El resultado también es un entero.

```
x + y   // ok si x e y son enteros
```

Si intentas sumar algo que no es entero (como una función `void`), marca error.

---

4. Comparaciones (`==`, `<`, `>`, `!=`, etc.)

Estas comparaciones también requieren **dos enteros**, pero su resultado es tipo **booleano** (aunque en C- lo representamos con `int`, como 0 = falso, no 0 = verdadero).

---

5. Asignaciones

Solo puedes asignar a una variable `int` algo que también sea `int`.

```
int x;
x = 5;        // válido
```

```
x = "hola";   // error
```

---

6. Condicionales `if` / `while`

Las condiciones deben ser algo que pueda evaluarse como **verdadero o falso**, así que aceptamos tanto enteros como booleanos.

```
if (x < 5) { ... }  // ok
while (x) { ... }   // ok
```

---

7. return en funciones

Depende del tipo de la función:

- Si es `int`, debe hacer `return` con un valor entero.
- Si es `void`, no debe devolver nada.

```
int suma() {
  return 5;    // válido
}

void saludar() {
  return;      // válido
}
```

---

8. Llamadas a funciones

Cuando llamas una función:

- El número de argumentos debe coincidir con el número de parámetros.
- Cada argumento debe ser del tipo esperado.

```
int suma(int a, int b);

suma(5, 6);      // válido
suma(5);         // falta uno
suma("hola", 2); // tipo incorrecto
```

Tabla de símbolos

La usamos para verificar que todo lo que usamos fue declarado correctamente, con su tipo, y en el lugar correcto.

Funciona de la siguiente manera

Cada entrada es un objeto que contiene datos como estos:

```
SymbolAttributes(
  name="x",              # El nombre
  kind="variable",       # Puede ser 'variable', 'function', o 'param'
  type=TokenType.INT,    # INT o VOID
  lineno=[5, 10],        # En qué líneas aparece
  scope="main",          # En qué función o ámbito está
  is_array=False,        # Si es un arreglo
  params=[],             # Lista de parámetros si es una función
  location=2             # Posición simulada en memoria
)
```

Ejemplo de tabla de símbolos

Nombre	Tipo	Rol	Ámbito	Líneas	Extra
`gcd`	INT	función	global	4	`params: 2`
`u`	INT	parámetro	gcd	4, 5	
`x`	INT	variable	main	9, 10	

Scopes

Por ahora, el compilador solo maneja los scopes `"global"` y el nombre de la función.
Pero si se quisiera mejorar en un futuro (por ejemplo, variables dentro de un `if {}` o un `while {}`), podríamos usar una **pila (stack) de scopes** como esta:

```
scope_stack = ["global"]

# Al entrar a una función o bloque:
scope_stack.append("main")

# Al salir:
scope_stack.pop()
```

Esto ayudaría a saber exactamente dónde está cada variable y evitar confusiones entre funciones distintas.

Conclusión del Analizador Semántico

Con todo esto, el analizador semántico hace varias cosas:

- Verifica que todo lo que uses haya sido declarado antes.
- Revisa que las operaciones y funciones tengan sentido con sus tipos.
- Detecta errores como llamar funciones mal o hacer operaciones con tipos incorrectos.
- Se asegura de que el `return` de las funciones coincida con su tipo.
