🧪 Fase 2 – Generación de Código Intermedio (Explicación Teórica)

Este documento resume los temas de la **Fase 2 del proyecto Compiscript**, siguiendo la secuencia de trabajo desde la entrada hasta la salida de la fase, basado en el libro *Compilers: Principles, Techniques, and Tools (Dragon Book)*.

---

0) ¿Qué fase es?

- Es la **Fase 2 del proyecto**, que sigue a la **fase de análisis semántico**.
- En el flujo de un compilador:

```
Fuente → Análisis léxico → Análisis sintáctico → Análisis semántico
       → Generación de Código Intermedio → Optimización → Código final
```

📖 *Dragon Book, Cap. 6: Intermediate Code Generation*.

---

1) ¿Qué tema es?

- El tema es **Generación de Código Intermedio (CI)**.
- Consiste en **traducir el AST** (árbol sintáctico con información semántica) en una forma intermedia, normalmente más **cercana al código máquina** pero todavía independiente de la arquitectura.
- En el proyecto, se usará **Código de Tres Direcciones (Three Address Code, TAC)** como representación.

---

2) ¿Qué recibe de entrada?

La fase recibe como entrada:

1. **Árbol de Sintaxis Anotado (AST)** generado en fases anteriores.
    - Contiene nodos para expresiones, sentencias, funciones, etc.
    - Incluye información semántica validada (tipos, ámbitos).
2. **Tabla de Símbolos**:
    - Nombres de variables, funciones, clases.
    - Tipos de datos.
    - Información de alcance (scope).
    - Posibles offsets de memoria o direcciones relativas.

📖 *Dragon Book, Sección 6.1: Intermediate Languages*.

---

3) ¿Qué proceso hace?

La fase de Generación de CI realiza varias transformaciones:

a) Definir el lenguaje intermedio

- Se elige el **formato TAC**:
    - **Cuádruplas**: `(op, arg1, arg2, resultado)`
    - **Tríplices**: `(op, arg1, arg2)` con resultado implícito
    - **Código lineal con temporales**

Ejemplo:

```
x = a + b * c
```

Se convierte a:

```
t1 = b * c
t2 = a + t1
x = t2
```

---

b) Recorrido del AST

- Se usa un **Visitor** o **Listener ANTLR** que recorre los nodos del AST.
- En cada nodo:
  - Expresiones → se transforman en TAC con temporales.
  - Condicionales → generan etiquetas y saltos.
  - Bucles → generan etiquetas de inicio/fin y `goto`.
  - Funciones → generan prólogos/epílogos (activación/desactivación de RA).

📖 *Dragon Book, Sección 6.6: Translation of Expressions*.

---

c) Gestión de temporales y etiquetas

- **Temporales (`t1, t2, …`)**:
  - Se crean para almacenar resultados intermedios.
  - Se reciclan cuando ya no son necesarios.
- **Etiquetas (`L1, L2, …`)**:
  - Se crean para control de flujo (saltos, condiciones).

📖 *Dragon Book, Sección 6.4: Three-Address Code*.

---

d) Extender la tabla de símbolos

- Guardar información adicional:
  - Dirección de memoria de cada variable.
  - Relación de variables locales y globales.
  - Posiciones de parámetros en registros de activación.

---

e) Construcción de Registros de Activación (RA)

- Cada función activa un RA en la pila.
- El RA incluye:
  - Parámetros formales.
  - Variables locales.
  - Dirección de retorno.
  - Enlaces estáticos/dinámicos (para closures).

Ejemplo en TAC:

```
param a
param b
call suma, 2
return t1
```

📖 *Dragon Book, Sección 7.2: Activation Records*.

---

f) Generación del TAC de sentencias y expresiones

- **Asignaciones**: `x = y + z`
- **Expresiones booleanas** con cortocircuito (`&&`, `||`).
- **Condicionales**:

```
if x < y goto L1
goto L2
L1: …
L2: …
```

- **Bucles** (`while`, `for`, `do-while`) usando etiquetas.
- **Funciones y llamadas**: manejo de parámetros, retorno.
- **Clases/Objetos**: acceso a atributos y métodos traducido a TAC.

📖 *Dragon Book, Sección 6.6: Boolean Expressions and Control Flow*.

---

4) ¿Qué salida tiene?

La salida es:

1. **Archivo de Código Intermedio (TAC)**, por ejemplo `program.tac`, con instrucciones en formato de tres direcciones.
    - Independiente de la máquina.
    - Útil para la fase de **optimización** y luego para traducir a ensamblador.

Ejemplo:

Entrada Compiscript:

```
let x: integer = 10;
let y: integer = 20;
let z = x + y * 2;
```

Salida TAC:

```
t1 = y * 2
t2 = x + t1
z = t2
```

2. **Tabla de símbolos extendida** (`symbol_table.json`) con offsets, tipos, temporales.
3. **Registros de activación modelados** en TAC para llamadas a funciones.

📌 **En resumen**:

- **Entrada:** AST + Tabla de símbolos.

- **Proceso:** recorrido del AST, creación de temporales, etiquetas, RA, traducción a TAC.
- **Salida:** TAC + tabla de símbolos extendida + documentación del CI.