Generación de Código Intermedio (TAC)

Equipo: *Andy Fuentes 22944, Davis Roldán 22672, Diederich Solis 22952* — Fecha: 29 de septiembre de 2025

1 Resumen

Implementamos un compilador de **Compiscript** (subset de TypeScript) hasta **código intermedio TAC**. El flujo es: ANTLR4 → AST → **Chequeo semántico + Tabla de símbolos** → **Generación TAC** (con *peephole* básico). Incluimos un **IDE en Streamlit** y una **batería de pruebas** automatizadas. Resultado: `18 passed` (sin fallos).

2 Cómo ejecutar

CLI:

```
PYTHONPATH=src python -m src.cli program/program.cps
# Genera program.cps.tac y muestra tabla de s mbolos + errores si los hay
```

IDE:

```
streamlit run src/ide/app.py
# Editor, consola, AST (Graphviz), s mbolos y pesta a TAC (descargable)
```

3 Arquitectura (alto nivel) (15/25 CI)

Módulos clave:

- `parsing/antlr`: gramática `Compiscript.g4`, Lexer/Parser/Visitor de ANTLR4 y utilidades de construcción.
- `semantic/`: `checker.py` (*visitor* semántico), `symbol_table.py`, `types.py`, `symbols.py`, `diagnostics.py`.
- `ir/tac/`: `instructions.py` (ISA TAC), `emitter.py` (builder), `program.py` (funciones/programa).
- `ir/backend/tac_generator.py`: *visitor* que traduce AST → TAC.
- `src/cli.py`: orquestación (parse, semántica, TAC, guardado).
- `src/ide/app.py`: IDE (editor, resultados, TAC).

Decisiones: Visitor único para semántica y otro para TAC; TAC textual simple, legible y estable; *peephole* seguro y local.

4 Tabla de símbolos (10/10)

Soporte: ámbitos anidados (GLOBAL, FUNCTION, CLASS, BLOCK), funciones (firma/retorno), clases (campos/métodos e *lookup* por herencia), `const` (no reasignable), `foreach` (item tipado desde `Array<T>`). **Errores** (códigos E101–E500): tipo en condicionales/operaciones, símbolo no definido, retorno inconsistente, `break/continue` fuera de bucles, casos `switch` incompatibles, etc.

Listing 1: Ejemplo de volcado (IDE: pestaña "Tabla de símbolos").

```
[
  {"scope":"GLOBAL __global__","entries":[
    {"name":"N","kind":"const","type":"IntType"},
    {"name":"max","kind":"func","type":"(int,int)->int"}
  ]},
  {"scope":"FUNCTION main","entries":[
    {"name":"acc","kind":"var","type":"IntType"}
  ]}
]
```

5 Diseño del TAC (ISA) (10/25 CI + 25/65 TAC)

Operandos: temporales `tN`, locales `%x`, globales `@g`, literales `#5`, `#"str"`.

Estructura de función

```
.func <name>(<params>) : <ret>
  .locals <N>
<labels / instrucciones...>
.endfunc
```

Instrucciones principales

```
x = y          (move)          t = a op b    (Binary: + - * / % == != < <= > >=)
t = op a       (Unary: - !)    label L       goto L          if t goto L
   ifFalse t goto L
param v        call f, n -> t                ret [v]
t = new C      getf obj,"f" -> t             setf obj,"f",v
t = newarr T, size              t = aload arr, i       astore arr, i, v
print v
```

Convenciones: `call` empuja `param` en orden de aparición; retorno en temporal; `.locals` cuenta declaradas en función (no temporales).

6 Generación TAC (visitor) (40/65 TAC)

Expresiones: cascadas izquierda→derecha (`+,-,*,/,%,==,!=,<,<=,>,>=`), unarios (`-,!`), literales (`#`, `#"..."`, `#1/#0` para `true/false`, `#null`). **Corto-circuito:**

```
# a && b
dst = a; ifFalse dst goto Lend; dst = b; ifFalse dst goto Lend; Lend:
# a || b
dst = a; if dst goto Lend; dst = b; if dst goto Lend; Lend:
```

Ternario:

```
ifFalse cond goto Lfalse; dst = then; goto Lend; Lfalse: dst = else; Lend:
```

LHS encadenado (call/index/prop):

```
f(...): param args; call f, n -> t
arr[i]: t = aload arr, i
obj.f : t = getf obj, "f"
```

Asignación: `%id = expr`; `obj.f = v` → `setf`. (Asignación a `arr[i]` lista como trabajo futuro si se requiere). **Control de flujo:** `if/else`, `while`, `do-while`, `for(init;cond;step)`, `break/continue` con pila de bucles. **Funciones/retorno:** epílogo único `Lret`; return mueve al `ret_temp` y `goto Lret`; funciones void retornan `ret`.

7 Optimizaciones (peephole seguro, 5/65 TAC)

Regla implementada: `goto L; L:` ⇒ eliminar `goto`. Aplicada al final de cada función:

```
def _peephole(self, code):
    out = []; i = 0
    from ir.tac.instructions import Goto, Label
    while i < len(code):
        cur = code[i]
        if isinstance(cur, Goto) and i+1 < len(code) \
           and isinstance(code[i+1], Label) and code[i+1].name == cur.
               label:
```

```
            i += 1; continue
        out.append(cur); i += 1
    return out
```

Efecto: elimina "ruido" tras `continue`/`break` y ramas que saltan directo a `Lret`.

8 IDE (Streamlit) **(Completa)**

Pestañas: *Diagnósticos*, *Árbol*, *Tokens*, *Intermedio (TAC)*. Se **bloquea** la generación de TAC si hay errores sintácticos/semánticos; al compilar OK se habilita descarga del `.tac`. Editor con tema oscuro y ejemplos cargables.

9 Pruebas **(Validación)**

Cobertura: expresiones, if/else, while, do-while, for, and/or con corto-circuito, ternario, llamadas y retorno, arrays (newarr/aload), `print`, `break/continue`, y semántica (tipos, const, foreach, this/new, switch). **Resultado:**

```
$ pytest -q
18 passed in 0.25s
```

10 Ejemplo final (smoke test)

```
.func max(a, b) : integer
  t1 = %a > %b
  ifFalse t1 goto Lelse
  t0 = %a
  goto Lret
Lelse:
  t0 = %b
Lret:
  ret t0
.endfunc
```