UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

[M2] Fase de Generación de Código Intermedioa

Sofía Velasquez, 22049
José Rodrigo Marchena, 22398
Irving Fabricio Acosta, 22781
Diego García, 22404

Gabriel Brolo Tobar
Construcción de Compiladores
Guatemala, 2025

Link repositorio: https://github.com/wwIrvingww/The_Compiler.git

1. Arquitectura del Compilador

1.1. Estructura General

El compilador de **Compiscript** está diseñado bajo un enfoque modular, donde cada fase del proceso de compilación se encuentra desacoplada y con responsabilidades bien definidas. El proyecto se organiza dentro de un entorno Dockerizado, lo que garantiza portabilidad y consistencia entre equipos de desarrollo.

Estructura principal del repositorio:

- src/: Contiene las fases principales del compilador.
    - parser/: Archivos generados por ANTLR que definen la gramática del lenguaje.
    - semantic/: Implementación del análisis semántico y verificación de tipos.
    - tac_generator.py: Módulo encargado de generar el **código intermedio (TAC)**.
    - tac_nodes.py: Define las estructuras de datos utilizadas por el TAC.
    - temps.py: Contiene la lógica de gestión de variables temporales.
    - runtime_integration.py y cfg.py: Implementan el manejo de registros de activación y control de flujo.
- tests/: Batería de pruebas unitarias y de integración.

1.2. Componentes Principales

a) Visitor de Generación TAC

El archivo tac_generator.py implementa un visitor que recorre el árbol sintáctico abstracto (AST) producido por ANTLR y genera las instrucciones TAC correspondientes. Cada nodo del AST se traduce en una o más instrucciones en forma de **cuádruplas**.

b) Emisión de Instrucciones

El generador TAC ofrece funciones para emitir instrucciones como:

- Operaciones aritméticas y lógicas.
- Asignaciones (=).
- Saltos (goto, if-goto).
- Manejo de arreglos (CREATE_ARRAY, PUSH_TO_ARRAY, len, getidx).

Esto permite mantener una representación intermedia flexible, legible y fácilmente traducible a lenguaje ensamblador (por ejemplo, MIPS).

c) Gestor de Temporales

El módulo temps.py implementa la clase TempAllocator, que administra la creación, reserva y liberación de variables temporales (t0, t1, …).
Esto evita colisiones de nombres y prepara la base para un futuro asignador de registros optimizado.

d) Manejo de Registros de Activación

runtime_integration.py y cfg.py implementan la gestión de **frames** para cada función, asignando offsets a parámetros y variables locales:

- **Parámetros**: offsets positivos (0, 4, 8, …)
- **Locales**: offsets negativos (−4, −8, …)

Esto permite la integración directa con la etapa de generación de código assembler.

2. Ejecución del Compilador

2.1. Configuración del Entorno

El proyecto se ejecuta en un entorno Dockerizado para garantizar la portabilidad.
Los comandos principales son:

```
docker compose build dev
docker compose run --service-ports dev bash
```

Una vez dentro del contenedor (root@container:/app), se pueden utilizar los siguientes scripts:

- Generar parser (ANTLR):
  scripts/gen_parser.sh
- Ejecutar el parser completo:
  scripts/run_parser.sh
- Probar la generación TAC:
  scripts/run_tac_gen.sh

3. Lenguaje Intermedio: TAC (Three Address Code)

3.1. Formato General

Cada instrucción del TAC se representa como una **cuádrupla**:

(op, arg1, arg2, result)

- **op**: Operación u opcode (por ejemplo, +, =, goto).
- **arg1**, **arg2**: Operandos.
- **result**: Resultado o destino.

Ejemplo:

```
+, a, b, t0 ; t0 = a + b
=, t0, -, x ; x = t0
```

3.2. Operaciones Soportadas

Tipo	Operaciones
Aritméticas	+, -, *, /, %, uminus
Lógicas	&&, `
Comparaciones	==, !=, <, <=, >, >=
Asignación y movimiento	=
Control de flujo	label, goto, if-goto
Arreglos	CREATE_ARRAY, PUSH_TO_ARRAY, len, getidx
Funciones	return (y próximamente param, call)

3.3. Temporales y Etiquetas

- **Temporales:** t0, t1, t2, … generados dinámicamente.
- **Etiquetas:** L0, L1, L2, … utilizadas para marcar posiciones de salto.

4. Ejemplo de Traducción

Código fuente:

```
let x = 2 * 2;
```

TAC generado:

```
t0 = 2
t1 = 2
t2 = t0 * t1
x  = t2
```

Representado como cuádruplas:

```
(*, 2, 2, t2)
(=, t2, -, x)
```

5. Control Flow Graph (CFG)

5.1 Propósito:

El CFG representa cómo fluye la ejecución dentro de un programa. Cada nodo del grafo corresponde a una secuencia de instrucciones que se ejecutan de forma lineal, sin saltos internos, llamados bloques. Los edges conectan dichos bloques y muestran las posibles transiciones de control provocadas por instrucciones de salto, condicionales o retornos.

5.2 Construcción del CFG

El módulo *cfg.py* se encarga de recorrer la lista de instrucciones TAC y genera el grafo de flujo. El proceso de divide en cuatro etapas:

- Identificación de labels:
  Crea un mapa entre las etiquetas de salto y la posición que ocupa dentro del código TAC. Esto permite identificar destinos de saltos.
- Detección de líderes:
  Se busca las instrucciones que empiezan un bloque.
- Construcción de bloques:
  Utilizando al lider, el CFG agrupa las instrucciones contiguas entre dos líderes consecutivos. Cada bloque guarda su rango de instrucciones y las etiquetas asociadas.
- Conexión de los bloques:
  Según lo que sea la última instrucción de cada bloque (goto, if-goto, return, etc) se crean edges que conectan a los bloques entre ellos.

5.3 Ejemplo

Si se tiene un código así:

```
let x: integer = 0;
if (x < 10) {
   x = x + 1;
}
```

El TAC genera algo así:

```
t0 = x < 10
if t0 goto L1
goto L2
label L1
t1 = x + 1
x = t1
label L2
```

Y el CFG que se genera daría algo así:

Bloque	Instrucciones	Sucesores

0	Evaluación de condición y saltos	1, 2
1	Cuerpo del if	2
2	Continuación del programa	N/A