Universidad del Valle de Guatemala
Facultad de Ingeniería
Departamento de Ciencias de la Computación

Fase de Compilación: Generación de Código Intermedio

Lucía Alejandra Guzmán Domínguez 20262
Bianca Calderón 22272
Daniel Eduardo Dubón Ortiz 22233

1. Arquitectura General

La arquitectura de Compiscript se organiza en módulos independientes que representan cada fase del proceso de compilación:

• CompiscriptLexer.py: responsable del análisis léxico.
• CompiscriptParser.py y CompiscriptVisitor.py: construyen el árbol sintáctico.
• SemanticListener.py: valida tipos, ámbitos y declaraciones.
• symbol_table.py y symbols.py: gestionan la tabla de símbolos y entornos.
• tac_generator.py y tac.py: generan el código intermedio en forma de instrucciones TAC.
• server.py: implementa un servidor Flask que permite la interacción y prueba del compilador vía API REST.
• Driver.py: ejecuta el flujo principal del compilador.

2. Módulo: symbol_table.py

El módulo symbol_table.py administra la tabla de símbolos durante el análisis semántico. Cada símbolo (variable, función o clase) se almacena con metadatos como tipo, ámbito, dirección de memoria, nivel léxico y desplazamiento dentro del marco de activación. Incluye soporte para entornos anidados y generación de etiquetas de función, direcciones de memoria y frames de activación (FrameLayout).

Ejemplo de estructura de datos usada:

```
class SymbolTable:
    def declare_var(self, name: str, varinfo: VarInfo): ...
    def enter_function(self, name: str): ...
    def assign_memory_addresses(self): ...
```

Esta clase controla la creación de nuevos ámbitos, asignación de memoria y offsets de parámetros.

3. Módulo: tac_generator.py

El generador TAC (Three Address Code) traduce el Árbol de Sintaxis Abstracta (AST) en un conjunto de instrucciones intermedias que representan operaciones básicas (asignaciones, saltos, llamadas, etc.). Utiliza un conjunto de clases de apoyo definidas en tac.py como BinaryOp, Jump, CondJump, Param y Return.

Cada nodo del AST tiene un método visitX correspondiente (por ejemplo, visitVarDecl, visitBinary, visitCall) que traduce la operación al formato TAC. El resultado es una lista ordenada de instrucciones que puede interpretarse o traducirse posteriormente a ensamblador.

4. Módulo: SemanticListener.py

El listener semántico coordina la verificación de tipos, la coherencia entre declaraciones y usos de variables, y la correcta vinculación de las funciones y sus parámetros. Implementa el modelo de ambientes estáticos y dinámicos mediante la interacción con la tabla de símbolos.

5. Módulo: server.py

Este módulo implementa un servidor Flask que expone una API REST. El endpoint /compile recibe código fuente de Compiscript y devuelve como salida el árbol sintáctico, el AST, la tabla de símbolos y el código TAC generado. Facilita la depuración y visualización del proceso de compilación en entorno web.

6. Ejemplo de Ejecución

Entrada del usuario:

```
let a = 10;
let b = 20;
function add(x:int, y:int):int {
  let sum = x + y;
  return sum;
}
add(a, b);
```

Salida esperada:

```
--- Symbol Table ---
a -> integer@mem_0
b -> integer@mem_1
add(x:int@-16, y:int@-20)

--- TAC ---
a = 10
b = 20
add:
BeginFunc
t0 = x + y
sum = t0
Return sum
EndFunc
```

7. Conclusiones

El compilador Compiscript constituye una implementación modular y extensible que demuestra los principios fundamentales del diseño de compiladores: análisis, síntesis y generación de código. Su arquitectura permite añadir optimizaciones futuras y extender el lenguaje base con nuevas estructuras y tipos.