

Arquitectura del Compilador Python a x86_64

Visión General

El compilador está diseñado con una arquitectura de múltiples fases que transforma código Python en ensamblador x86_64 siguiendo el patrón clásico de compiladores:

```
Código Python → Lexer → Parser → AST → Generador de Código → ASM x86_64
```

Componentes Principales

1. Frontend - Análisis Léxico y Sintáctico

Lexer (PythonSubsetLexer + IndentationLexer)

- **Ubicación:** `src/main/java/parser/`
- **Responsabilidad:** Convierte el texto fuente en tokens
- **Características especiales:**
 - Manejo de indentación Python con tokens `INDENT` y `DEDENT`
 - Procesamiento de strings, números, identificadores
 - Soporte para palabras clave (`for`, `while`, `and`, `or`, etc.)

Parser (PythonSubsetParser)

- **Generado por:** ANTLR4 desde `grammar/PythonSubset.g4`
- **Responsabilidad:** Análisis sintáctico y construcción del parse tree
- **Soporte para:**
 - Expresiones aritméticas con precedencia correcta
 - Estructuras de control (for, while)
 - Asignaciones y llamadas a funciones

2. Representación Intermedia - AST

Nodos AST

- **Ubicación:** `src/main/java/parser/ast/`
- **Patrón:** Visitor Pattern para traversal
- **Tipos de nodos:**

Nodo	Propósito	Ejemplo
ProgNode	Programa completo	Raíz del AST
AssignNode	Asignaciones	<code>x = 10</code>
BinaryOpNode	Operaciones binarias	<code>x + y, a < b</code>

Nodo	Propósito	Ejemplo
ForNode	Ciclos for	<code>for i in range(10):</code>
WhileNode	Ciclos while	<code>while x < 5:</code>
FuncCallNode	Llamadas a función	<code>print(x)</code>

ASTBuilder

- **Responsabilidad:** Convierte parse tree en AST
- **Patrón:** Visitor de ANTLR4
- **Simplificaciones:** Elimina tokens innecesarios, estructura jerárquica

3. Backend - Generación de Código

CodeGenerator

- **Ubicación:** `src/main/java/codegen/CodeGenerator.java`
- **Responsabilidad:** Genera código ensamblador x86_64
- **Algoritmo en dos fases:**
 1. **Recolección:** Identifica strings literales
 2. **Generación:** Produce código ASM

Manejo de Variables

- **Stack-based:** Variables en memoria relativa a `rbp`
- **Offsets negativos:** `[rbp-8]`, `[rbp-16]`, etc.
- **Gestión automática:** El compilador asigna offsets automáticamente

Flujo de Compilación

Fase 1: Preprocesamiento

```
// Main.java - preprocessPythonIndentation()
"for i in range(3):\n    print(i)"
→
"for i in range(3):\nINDENT print(i)\nDEDENT"
```

Fase 2: Análisis Léxico

```
"for i in range(3):" → [FOR, IDENTIFIER(i), IN, IDENTIFIER(range), ...]
```

Fase 3: Análisis Sintáctico

```
Tokens → Parse Tree (estructura ANTLR)
```

Fase 4: Construcción AST

```
// ASTBuilder.visitFor_stmt()  
Parse Tree → ForNode(variable="i", iterable=RangeNode([3]), body=[...])
```

Fase 5: Generación de Código

```
// CodeGenerator.visit(ForNode)  
ForNode →  
"""  
loop_start_0:  
    mov rax, [rbp-8]  
    cmp rax, 3  
    jge loop_end_1  
    ; cuerpo del ciclo  
    inc rax  
    jmp loop_start_0  
loop_end_1:  
"""
```

Patrones de Diseño Utilizados

Visitor Pattern

- **Usado en:** AST traversal, generación de código
- **Ventaja:** Separación de algoritmos y estructura de datos
- **Implementación:** `ASTVisitor<T>` interface

Strategy Pattern

- **Usado en:** Diferentes tipos de nodos AST
- **Implementación:** Cada nodo implementa `ASTNode.accept()`

Builder Pattern

- **Usado en:** Construcción del AST
- **Implementación:** `ASTBuilder` construye nodos gradualmente

Gestión de Memoria

Variables

```
; Stack frame layout  
; rbp-8: primera variable
```

```
; rbp-16: segunda variable  
; rbp-24: tercera variable
```

Strings Literales

```
section .data  
str12345: db "Hello World", 0x0A, 0
```

Expresiones

- **Stack-based evaluation:** Usa `push/pop` para evaluación
- **Registros temporales:** `rax, rbx` para operaciones

Extensibilidad

Agregar Nuevos Nodos AST

1. Crear clase `XxxNode implements ASTNode`
2. Agregar `visit(XxxNode)` a `ASTVisitor`
3. Implementar en `ASTPrinter` y `CodeGenerator`
4. Agregar caso en `ASTBuilder.visitXxx()`

Agregar Nuevas Construcciones Sintácticas

1. Modificar `grammar/PythonSubset.g4`
2. Regenerar parser con ANTLR4
3. Implementar visitor en `ASTBuilder`
4. Agregar generación de código

Optimizaciones Futuras

- Análisis de flujo de datos
- Eliminación de código muerto
- Optimización de registros
- Plegado de constantes