Guía de Desarrollo - Compilador Python to x86_64

Arquitectura de Desarrollo

Estructura del Código Fuente

```
src/
├── main/java/
│   ├── codegen/
│   │   └── CodeGenerator.java      # Generación de código ASM
│   ├── parser/
│   │   ├── ASTBuilder.java         # Construcción del AST
│   │   ├── Main.java               # Punto de entrada
│   │   └── ast/
│   │       ├── ASTNode.java        # Interfaz base
│   │       ├── ASTVisitor.java     # Patrón Visitor
│   │       ├── ASTPrinter.java     # Depuración AST
│   │       └── *Node.java          # Nodos específicos
│   └── util/
│       ├── PythonIndentPreprocessor.java  # Manejo de indentación
│       ├── TreeViewer.java         # Visualizador del parse tree
│       └── ASTViewer.java          # Visualizador del AST
├── main/antlr4/parser/             # Clases generadas por ANTLR
└── test/                           # Archivos de prueba Python
```

Stack Tecnológico

- **Lenguaje**: Java 8+
- **Parser Generator**: ANTLR 4.13.2
- **Target**: x86_64 Assembly (Linux ABI)
- **Build System**: Manual compilation
- **Testing**: Archivos de prueba en `src/test/`

Configuración del Entorno de Desarrollo

Prerrequisitos

1. **JDK 8 o superior**

   ```
   java -version  # Verificar instalación
   ```

2. **ANTLR 4.13.2**
   - Descargar `antlr-4.13.2-complete.jar`
   - Colocar en directorio `lib/`

3. **Editor recomendado**: VS Code con extensión Java y extension ANTLR4 gramar syntax support

Setup del Workspace

1. **Compilar ANTLR Grammar**

```
cd grammar/
java -jar ../lib/antlr-4.13.2-complete.jar PythonSubset.g4 -visitor -o
../src/main/antlr4/parser/
```

2. **Compilar Proyecto Completo**

```
# Desde directorio raíz
javac -cp "lib/*" -d build/ src/main/java/**/*.java
src/main/antlr4/parser/*.java
```

3. **Verificar Setup**

```
java -cp "build:lib/*" parser.Main src/test/ejemplo.py
```

Flujo de Desarrollo

Agregar Nuevas Construcciones Sintácticas

Ejemplo: Agregar Funciones Definidas por Usuario

1. Modificar Gramática ANTLR

Editar `grammar/PythonSubset.g4`:

```
// Ejemplo: Agregar definición de funciones
func_def
    : 'def' IDENTIFIER '(' param_list? ')' ':' NEWLINE INDENT stmt+ DEDENT
    ;

param_list
    : IDENTIFIER (',' IDENTIFIER)*
    ;

compound_stmt
    : for_stmt
    | while_stmt
    | if_stmt
    | func_def  // Agregar nueva regla
    ;
```

2. Regenerar Parser

```
cd grammar/
java -jar ../lib/antlr-4.13.2-complete.jar PythonSubset.g4 -visitor -o
../src/main/antlr4/parser/
```

3. Crear Nodo AST

src/main/java/parser/ast/FuncDefNode.java:

```
public class FuncDefNode implements ASTNode {
    private String name;
    private List<String> params;
    private List<ASTNode> body;

    public FuncDefNode(String name, List<String> params, List<ASTNode> body) {
        this.name = name;
        this.params = params;
        this.body = body;
    }

    public String getName() { return name; }
    public List<String> getParams() { return params; }
    public List<ASTNode> getBody() { return body; }

    @Override
    public <T> T accept(ASTVisitor<T> visitor) {
        return visitor.visit(this);
    }
}
```

4. Actualizar ASTVisitor Interface

src/main/java/parser/ast/ASTVisitor.java:

```
public interface ASTVisitor<T> {
    // ... métodos existentes ...
    T visit(FuncDefNode node);  // Agregar nuevo método
}
```

5. Implementar en ASTBuilder

src/main/java/parser/ASTBuilder.java:

```
@Override
public ASTNode visitFunc_def(PythonSubsetParser.Func_defContext ctx) {
    String name = ctx.IDENTIFIER(0).getText();

    List<String> params = new ArrayList<>();
    if (ctx.param_list() != null) {
        for (TerminalNode id : ctx.param_list().IDENTIFIER()) {
            params.add(id.getText());
        }
    }

    List<ASTNode> body = new ArrayList<>();
    for (PythonSubsetParser.StmtContext stmt : ctx.stmt()) {
        body.add(visit(stmt));
    }

    return new FuncDefNode(name, params, body);
}
```

6. Implementar Generación de Código

src/main/java/codegen/CodeGenerator.java:

```
@Override
public Void visit(FuncDefNode node) {
    // Generar etiqueta de función
    sb.append("func_").append(node.getName()).append(":\n");
    sb.append("    ; Prólogo de función\n");
    sb.append("    push rbp\n");
    sb.append("    mov rbp, rsp\n");

    // Procesar parámetros (en rdi, rsi, rdx, rcx, r8, r9 según System V ABI)
    int paramOffset = 16;
    for (int i = 0; i < node.getParams().size() && i < 6; i++) {
        String param = node.getParams().get(i);
        varOffsets.put(param, -paramOffset);
        // Copiar desde registros a stack
        String[] regs = {"rdi", "rsi", "rdx", "rcx", "r8", "r9"};
        sb.append("    mov [rbp-").append(paramOffset).append("],
").append(regs[i]).append("\n");
        paramOffset += 8;
    }

    // Cuerpo de la función
    for (ASTNode stmt : node.getBody()) {
        stmt.accept(this);
    }

    // Epílogo
    sb.append("    ; Epílogo de función\n");
```

```
        sb.append("    mov rsp, rbp\n");
        sb.append("    pop rbp\n");
        sb.append("    ret\n\n");

        return null;
    }
```

7. Actualizar ASTPrinter para Depuración

src/main/java/parser/ast/ASTPrinter.java:

```
    @Override
    public String visit(FuncDefNode node) {
        StringBuilder sb = new StringBuilder();
        sb.append("FuncDefNode: ").append(node.getName()).append("\n");
        sb.append("  Params: [");
        for (int i = 0; i < node.getParams().size(); i++) {
            if (i > 0) sb.append(", ");
            sb.append(node.getParams().get(i));
        }
        sb.append("]\n");
        sb.append("  Body: [\n");
        for (ASTNode stmt : node.getBody()) {
            sb.append("    ").append(stmt.accept(this)).append("\n");
        }
        sb.append("  ]\n");
        return sb.toString();
    }
```

Testing

Crear Archivo de Prueba

src/test/test_func.py:

```
def suma(a, b):
    resultado = a + b
    return resultado

x = suma(5, 3)
print(x)  # Debería imprimir 8
```

Ejecutar y Verificar

```
java -cp "build;lib/*" parser.Main src/test/test_func.py
cat build/ejemplo.asm  # Verificar código generado
```

```
# Ensamblar y ejecutar
nasm -f elf64 build/ejemplo.asm -o build/ejemplo.o
gcc build/ejemplo.o -o build/programa
./build/programa
```

Convenciones de Código

Estilo Java

- **Nomenclatura**:
  - Clases: `PascalCase`
  - Métodos/variables: `camelCase`
  - Constantes: `UPPER_CASE`
- **Estructura de clases**:

```
public class ExampleNode implements ASTNode {
    // Fields privados
    private Type field;

    // Constructor
    public ExampleNode(Type field) {
        this.field = field;
    }

    // Getters
    public Type getField() { return field; }

    // Accept method (requerido por ASTNode)
    @Override
    public <T> T accept(ASTVisitor<T> visitor) {
        return visitor.visit(this);
    }
}
```

Estilo Assembly

- **Indentación**: 4 espacios para instrucciones
- **Labels**: Sin indentación, terminados en `:`
- **Comentarios**: `; Descripción`

```
section .text
global _start

_start:
    ; Inicialización
    mov rax, 1
```

```
    mov rbx, 42

loop_start:
    ; Cuerpo del ciclo
    cmp rax, 10
    jge loop_end
    inc rax
    jmp loop_start

loop_end:
    ; Salida
    mov rax, 60
    syscall
```

Depuración y Testing

Habilitar Modo Debug

Modifica `Main.java`:

```
public class Main {
    private static final boolean DEBUG = true;

    public static void main(String[] args) {
        // ... código existente ...

        if (DEBUG) {
            System.out.println("=== TOKENS ===");
            // Imprimir tokens

            System.out.println("=== AST ===");
            ASTPrinter printer = new ASTPrinter();
            System.out.println(program.accept(printer));

            System.out.println("=== ASSEMBLY ===");
        }

        // ... generación de código ...
    }
}
```

Casos de Prueba

Estructura recomendada para archivos de test:

```
# test_feature.py
# Caso simple
simple_case = 42
```

```
# Caso complejo
for i in range(5):
    if i % 2 == 0:
        print("Par")
    else:
        print("Impar")
```

Verificación de Output

```
# Compilar
java -cp "build:lib/*" parser.Main src/test/test_feature.py

# Ensamblar y ejecutar
nasm -f elf64 build/ejemplo.asm -o build/ejemplo.o
gcc build/ejemplo.o -o build/programa
./build/programa
```

Optimizaciones y Mejoras

Performance

1. **Reutilización de registros**
    - Actual: Stack-based evaluation
    - Mejora: Register allocation
2. **Eliminación de código muerto**
    - Detectar variables no usadas
    - Optimizar expresiones constantes
3. **Optimización de ciclos**
    - Loop unrolling para rangos conocidos
    - Strength reduction

Características Pendientes

1. **Funciones definidas por usuario** (parcialmente diseñado arriba)
2. **Arrays y listas** con indexación y slicing
3. **Operadores de asignación compuesta** (`+=`, `-=`, `*=`, `/=`)
4. **Range con múltiples argumentos** (`range(start, stop, step)`)
5. **Import system** y módulos
6. **Diccionarios** y otras estructuras de datos
7. **Excepciones** (try/except)
8. **Clases y objetos** (OOP básico)
9. **Operaciones con strings** (concatenación, slicing, format)
10. **Tipos float/decimal** con operaciones de punto flotante

Arquitectura de Extensión

Plugin System (Future)

```
public interface CodegenPlugin {
    boolean canHandle(ASTNode node);
    String generate(ASTNode node, CodegenContext context);
}
```

Target Backends

El diseño actual permite agregar backends alternativos:

- **x86_32**: 32-bit assembly
- **ARM64**: ARM assembly
- **LLVM IR**: Para optimizaciones avanzadas
- **JVM Bytecode**: Java virtual machine

Análisis Semántico

Futuras mejoras pueden incluir:

```
public class SemanticAnalyzer implements ASTVisitor<Void> {
    private SymbolTable symbolTable;
    private List<SemanticError> errors;

    public void analyze(ProgNode program) {
        // Type checking
        // Variable declaration checking
        // Function call validation
    }
}
```

Contribución al Proyecto

Pull Request Workflow

1. **Fork y clone** del repositorio
2. **Crear branch** para feature: `git checkout -b feature/if-statements`
3. **Implementar** siguiendo las convenciones
4. **Testing exhaustivo** con casos edge
5. **Documentar** cambios en este archivo
6. **Submit PR** con descripción detallada

Coding Standards

- **Unit tests** para cada nueva característica

- **Documentación** inline en código complejo
- **Backwards compatibility** cuando sea posible
- **Performance testing** para cambios críticos

Bug Reports

Template para reportar bugs:

```
## Descripción
[Descripción concisa del problema]

## Reproducir
1. Archivo de prueba: [adjuntar archivo.py]
2. Comando ejecutado: [comando completo]
3. Output actual: [resultado obtenido]
4. Output esperado: [resultado esperado]

## Entorno
- OS: [Windows/Linux/macOS]
- Java version: [version]
- ANTLR version: [version]
```