

Documentación del Compilador Python to x86_64

Índice de Documentación

Este archivo contiene la documentación completa del proyecto **Compilador Python a x86_64**. Los documentos están organizados para proporcionar información desde conceptos básicos hasta detalles técnicos avanzados.

Documentos Disponibles

| Documento | Descripción | Audiencia |
|--|---|---------------------------------|
| architecture.md | Arquitectura general del sistema, componentes principales y patrones de diseño | Desarrolladores, Arquitectos |
| user-guide.md | Manual de instalación, uso y sintaxis soportada | Usuarios finales, Estudiantes |
| grammar-specification.md | Especificación completa de la gramática ANTLR, tokens y reglas de producción | Desarrolladores de compiladores |
| development.md | Guía para contribuir al proyecto, extender funcionalidad y convenciones de código | Contribuidores, Desarrolladores |
| examples.md | Ejemplos de código, casos de uso y limitaciones actuales | Usuarios, Testers |

Guía de Lectura Recomendada

Para Usuarios Nuevos:

1. Comenzar con [user-guide.md](#) para instalación y uso básico
2. Revisar [examples.md](#) para ver casos de uso prácticos
3. Consultar [architecture.md](#) si desea entender el funcionamiento interno

Para Desarrolladores:

1. Leer [architecture.md](#) para comprender el diseño
2. Estudiar [grammar-specification.md](#) para entender la gramática
3. Seguir [development.md](#) para contribuir al proyecto
4. Usar [examples.md](#) como referencia para testing

Para Estudiantes de Compiladores:

1. Iniciar con [grammar-specification.md](#) para comprender ANTLR
2. Continuar con [architecture.md](#) para el patrón AST/Visitor
3. Experimentar con [examples.md](#) para casos prácticos
4. Explorar [development.md](#) para extensiones

Resumen del Proyecto

El **Compilador Python to x86_64** es una implementación educativa que transforma un subconjunto del lenguaje Python en código ensamblador x86_64. El proyecto demuestra los conceptos fundamentales de construcción de compiladores utilizando tecnologías modernas.

Características Principales:

- **Sintaxis Python:** Soporte completo para indentación, variables, expresiones
- **Estructuras de Control:** Ciclos `for` y `while`, condicionales `if/elif/else`
- **Tipos de Datos:** Enteros, strings literales y booleanos
- **Expresiones:** Aritméticas (+, -, *, /, %, **), lógicas (and, or, not) y comparaciones (==, !=, <, >, <=, >=)
- **Operadores Unarios:** Negación aritmética y lógica (-, not, +)
- **Generación ASM:** Código x86_64 optimizado para Linux con gestión de strings

Stack Tecnológico:

- **ANTLR 4.13.2:** Generación de parsers
- **Java 8+:** Lenguaje de implementación
- **x86_64 Assembly:** Target de compilación
- **Linux ABI:** Convenciones de llamadas del sistema

📁 Estructura de Documentación

```
documentation/
├── README.md                  # Este archivo - índice general
├── architecture.md            # Diseño y componentes del sistema
├── user-guide.md              # Manual de usuario e instalación
├── grammar-specification.md   # Especificación completa de gramática ANTLR
├── development.md              # Guía de desarrollo y contribución
└── examples.md                 # Ejemplos de código y casos de uso
```

🔧 Quick Start

Si desea empezar rápidamente:

1. **Instalación Básica:** Ver [user-guide.md#instalación](#)
2. **Primer Ejemplo:** Consultar [examples.md#básicos](#)
3. **Compilar Código:** Seguir [user-guide.md#uso](#)

🔧 Para Contribuidores

Si desea contribuir al proyecto:

1. **Setup de Desarrollo:** [development.md#setup](#)
2. **Agregar Features:** [development.md#extensiones](#)
3. **Testing:** [development.md#testing](#)
4. **Code Style:** [development.md#convenciones](#)

📋 Referencias Adicionales

Documentación Externa:

- [ANTLR 4 Documentation](#)
- [x86_64 Assembly Reference](#)
- [System V ABI](#)

Recursos Académicos:

- **Compiladores**: "Engineering a Compiler" por Cooper & Torczon
- **ANTLR**: "The Definitive ANTLR 4 Reference" por Terence Parr
- **Assembly**: "Programming from the Ground Up" por Jonathan Bartlett

⚡ Reporte de Issues

Para reportar problemas o sugerir mejoras:

1. **Bugs**: Seguir template en [development.md#bug-reports](#)
2. **Feature Requests**: Crear issue con etiqueta "enhancement"
3. **Preguntas**: Usar etiqueta "question" en issues

📄 Licencia y Uso

Este proyecto está desarrollado con fines educativos. Consulte el archivo [LICENSE](#) en el directorio raíz para información sobre términos de uso.

📊 Estado del Proyecto

| Componente | Estado | Notas |
|---------------|---|-------------------------|
| Lexer/Parser | <input checked="" type="checkbox"/> Completo | ANTLR 4.13.2 |
| AST | <input checked="" type="checkbox"/> Completo | Patrón Visitor |
| CodeGen | <input checked="" type="checkbox"/> Completo | x86_64 básico |
| Testing | <input checked="" type="checkbox"/> Funcional | Casos de prueba |
| Documentation | <input checked="" type="checkbox"/> Completo | Esta documentación |
| If/Elif/Else | <input checked="" type="checkbox"/> Completo | Condicionales completos |
| While Loops | <input checked="" type="checkbox"/> Completo | Ciclos while |
| For Loops | <input checked="" type="checkbox"/> Completo | Ciclos for con range() |
| Logical Ops | <input checked="" type="checkbox"/> Completo | and, or, not |

Features Pendientes

| | | |
|--------------|------------------------------------|---------------------------------|
| Functions | <input type="checkbox"/> Pendiente | Funciones definidas por usuario |
| Arrays/Lists | <input type="checkbox"/> Pendiente | Estructuras de datos |

| Componente | Estado | Notas |
|------------|-----------|-----------------------------------|
| Range Args | Pendiente | range(start, stop, step) completo |

Última Actualización

Versión: 2.0 **Fecha:** Noviembre 2025 **Autores:**

- **Melo Reséndiz Jorge**
- **Guerrero Serrano Jafeth Oswaldo**
- **Paniagua Rico Juan Julián**

Estado: Funcional y documentado con características completas de control de flujo

Para información sobre versiones y changelog, ver el historial de commits en el repositorio principal.

Tip: Mantenga esta documentación actualizada conforme evolucione el proyecto. Cada nueva característica debe incluir ejemplos en [examples.md](#) y consideraciones de desarrollo en [development.md](#).

Manual de Usuario - Compilador Python to x86_64

Guía de Instalación

Requisitos del Sistema

- **Java:** JDK 8 o superior
- **ANTLR 4.13.2:** Para regeneración de parsers (opcional)
- **Sistema Operativo:** Windows/Linux/macOS
- **Ensamblador:** NASM (para ensamblar el código generado)
- **Enlazador:** GCC o similar (para crear ejecutables)

Instalación Paso a Paso

1. Clonar o descargar el proyecto

```
git clone <repository-url>
cd python2asm_ProgSistBase1
```

2. Compilar el proyecto

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

3. Verificar instalación

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

Uso del Compilador

Sintaxis del Comando

```
java -cp "build:lib/*" parser.Main <archivo_python>
```

Ejemplos de Uso

Ejemplo 1: Programa Simple

```
# archivo: test_simple.py
x = 10
```

```
y = 20
print(x + y)
```

Compilación:

```
java -cp "build:lib/*" parser.Main test_simple.py
```

Resultado: Genera `ejemplo.asm` en el directorio `build/`

Ejemplo 2: Ciclo For

```
# archivo: test_for.py
for i in range(3):
    print(i)
print("Fin del ciclo for")
```

Código generado:

```
section .data
; ... strings literales ...

section .text
; ... código del ciclo ...
```

Ejemplo 3: Ciclo While

```
# archivo: test_while.py
contador = 0
while contador < 5:
    print(contador)
    contador = contador + 1
```

Ejecución del Código Generado

1. Ensamblar

```
nasm -f elf64 build/ejemplo.asm -o build/ejemplo.o
```

2. Enlazar

```
gcc build/ejemplo.o -o build/programa -lc
```

3. Ejecutar

```
./build/programa
```

Sintaxis Soportada

Variables y Asignaciones

```
# Tipos soportados
x = 42          # Enteros
mensaje = "Hola" # Strings
activo = True    # Booleanos
inactivo = False

# Asignaciones
variable = expresion
```

Expresiones Aritméticas

```
# Operadores soportados
suma = a + b
resta = a - b
multiplicacion = a * b
division = a / b

# Precedencia respetada
resultado = 2 + 3 * 4 # = 14
```

Expresiones Lógicas

```
# Operadores de comparación
mayor = a > b
menor = a < b
mayor_igual = a >= b
menor_igual = a <= b
igual = a == b
diferente = a != b

# Operadores lógicos (completamente implementados)
y_logico = True and False # False
o_logico = True or False # True
```

```
negacion = not True          # False

# Expresiones complejas
resultado = (x > 5) and (y < 10) or (z == 0)
complejo = not (a == b) and (c != d)
```

Estructuras de Control

Ciclo For

```
# Sintaxis básica
for variable in range(numero):
    # cuerpo del ciclo
    instrucciones...

# Ejemplo
for i in range(10):
    print(i)
    x = i * 2
```

Ciclo While

```
# Sintaxis básica
while condicion:
    # cuerpo del ciclo
    instrucciones...

# Ejemplo
contador = 0
while contador < 5:
    print(contador)
    contador = contador + 1
```

Condicionales If/Elif/Else

```
# Sintaxis básica
if condicion:
    # cuerpo del if
    instrucciones...
elif otra_condicion:
    # cuerpo del elif
    instrucciones...
else:
    # cuerpo del else
    instrucciones...
```

```
# Ejemplo simple
x = 10
if x > 15:
    print("Mayor que 15")
elif x > 5:
    print("Mayor que 5")
else:
    print("5 o menor")

# Condicionales anidados
if x > 0:
    if x < 10:
        print("Entre 0 y 10")
    else:
        print("Mayor o igual a 10")
else:
    print("Menor o igual a 0")
```

Llamadas a Funciones

```
# Función print soportada
print(variable)
print("string literal")
print(expresion_aritmetica)

# Función range para ciclos for
range(5)      # 0 a 4
range(10)     # 0 a 9
```

Limitaciones Actuales

Características No Soportadas

- **Funciones definidas por usuario:** Solo `print()` y `range()` están implementados
- **Listas y estructuras de datos:** Solo variables escalares (`int`, `string`, `bool`)
- **Import statements:** Sin soporte para módulos
- **Range con argumentos:** Solo soporta `range(stop)`, no `range(start, stop, step)`
- **Tipos float/decimal:** Solo enteros, strings y booleanos
- **Operaciones sobre strings:** No se pueden concatenar o manipular strings
- **Asignación compuesta:** No soporta `+=`, `-=`, `*=`, etc.
- **Slicing:** No se pueden hacer operaciones de slicing

Restricciones de Sintaxis

- **Indentación:** Debe ser consistente (4 espacios o tabs, no mezclar)
- **Nombres de variables:** Solo letras, números y underscore (no pueden empezar con número)
- **Strings:** Comillas dobles o simples ("`texto`" o '`texto`')
- **Comentarios:** Soportados con `#` pero son ignorados por el compilador

- **Print:** Solo acepta un argumento a la vez
- **Range:** Solo acepta la forma `range(stop)` con un único argumento

Solución de Problemas

Errores Comunes

Error de Indentación

```
Error: Token recognition error at: '
```

Solución: Verificar que la indentación sea consistente

Error de Sintaxis

```
Error: mismatched input 'palabra' expecting {...}
```

Solución: Revisar que la sintaxis coincida con la gramática soportada

Error de Compilación Java

```
Exception in thread "main" java.lang.ClassNotFoundException
```

Solución: Verificar el classpath incluya `lib/*` y `build/`

Archivos de Prueba Incluidos

| Archivo | Propósito | Descripción |
|---------------------------------------|--------------------|-----------------------------|
| <code>src/test/ejemplo.py</code> | Prueba básica | Ciclo for simple |
| <code>src/test/test_while.py</code> | Ciclos while | Múltiples ejemplos de while |
| <code>src/test/test_for.py</code> | Ciclos for | Variaciones de for |
| <code>src/test/test_logical.py</code> | Operadores lógicos | And, or, not |

Depuración

Habilitar Modo Verbose

Modifica `Main.java` para agregar:

```
System.out.println("AST: " + program.toString());
System.out.println("Generated ASM saved to: " + outputPath);
```

Inspeccionar AST Generado

Usa **ASTPrinter** para visualizar el árbol:

```
ASTPrinter printer = new ASTPrinter();
program.accept(printer);
```

Ejemplos Completos

Programa de Suma Acumulativa

```
# suma_acumulativa.py
total = 0
for i in range(10):
    total = total + i
print(total)
```

Contador con While

```
# contador.py
numero = 1
while numero <= 5:
    print("Contador:")
    print(numero)
    numero = numero + 1
print("Terminado")
```

Operaciones Lógicas

```
# logica.py
a = True
b = False
resultado1 = a and b
resultado2 = a or b
print(resultado1) # False
print(resultado2) # True
```

Próximos Pasos

Después de usar exitosamente el compilador:

1. **Examinar código ASM:** Revisar `build/ejemplo.asm` generado

2. **Ejecutar programa:** Seguir pasos de ensamblado y enlazado
3. **Experimentar:** Probar diferentes construcciones sintácticas
4. **Contribuir:** Ver [development.md](#) para extender funcionalidad

Especificación de la Gramática PythonSubset

Visión General

La gramática [PythonSubset.g4](#) define un subconjunto simplificado del lenguaje Python que incluye las características esenciales para programación estructurada básica. Está diseñada para ser procesada por ANTLR4 y generar código ensamblador x86_64.

Estructura de la Gramática

Tokens Léxicos

Palabras Clave

```
FOR      : 'for' ;
IN       : 'in' ;
WHILE    : 'while' ;
IF       : 'if' ;
ELIF     : 'elif' ;
ELSE     : 'else' ;
AND      : 'and' ;
OR       : 'or' ;
NOT      : 'not' ;
TRUE     : 'True' ;
FALSE    : 'False' ;
```

Operadores

```
PLUS     : '+' ;
MINUS    : '-' ;
MULT     : '*' ;
DIV      : '/' ;
EQ       : '==' ;
NEQ      : '!=';
LT       : '<' ;
GT       : '>' ;
LE       : '<=' ;
GE       : '>=' ;
ASSIGN   : '=' ;
```

Delimitadores

```
LPAREN  : '(' ;
RPAREN  : ')' ;
```

```
COLON    : ':' ;
COMMA    : ',' ;
```

Tokens Especiales para Indentación

```
INDENT   : 'INDENT' ;    // Generado por preprocesador
DEDENT   : 'DEDENT' ;    // Generado por preprocesador
NEWLINE  : '\n' ;
```

Literales e Identificadores

```
INT      : [0-9]+ ;
STRING   : '"' (~["\r\n"])* '"';
ID       : [a-zA-Z_][a-zA-Z_0-9]* ;
WS       : [ \t\r]+ -> skip ;
```

Reglas de Producción

Estructura Principal

```
program
: statement+ EOF
;
```

Propósito: Define un programa como secuencia de una o más declaraciones.

Declaraciones (Statements)

```
statement
: simple_stmt NEWLINE
| compound_stmt
;
```

Tipos soportados:

- **simple_stmt**: Asignaciones, expresiones, llamadas a función
- **compound_stmt**: Estructuras de control con bloques

Declaraciones Simples

```

simple_stmt
: expr_stmt
| assign_stmt
;

assign_stmt
: ID ASSIGN expression
;

expr_stmt
: expression
;

```

Declaraciones Compuestas

```

compound_stmt
: for_stmt
| while_stmt
| if_stmt
;

```

Estructura de Control For

```

for_stmt
: FOR ID IN range_call COLON NEWLINE suite
;

range_call
: RANGE LPAREN expression RPAREN
;

```

Características:

- Variable de iteración: `ID`
- Función range obligatoria: `range(n)`
- Cuerpo con indentación: `suite`

Estructura de Control While

```

while_stmt
: WHILE expression COLON NEWLINE suite
;

```

Características:

- Condición: **expression** (debe evaluar a booleano)
- Cuerpo con indentación: **suite**

Estructura de Control If/Elif/Else

```

if_stmt
    : IF expr ':' NEWLINE INDENT stmt+ DEDENT elif_clause* else_clause?
    ;

elif_clause
    : ELIF expr ':' NEWLINE INDENT stmt+ DEDENT
    ;

else_clause
    : ELSE ':' NEWLINE INDENT stmt+ DEDENT
    ;

```

Características:

- Condición obligatoria en **if: expression**
- Cláusulas **elif** opcionales: Múltiples permitidas
- Cláusula **else** opcional: Máximo una
- Cuerpos con indentación: Cada bloque tiene su propio **suite**
- Evaluación secuencial: Se detiene en la primera condición verdadera

Bloques de Código (Suite)

```

suite
    : INDENT statement+ DEDENT
    ;

```

Manejo de Indentación:

- **INDENT**: Incremento de nivel de indentación
- **DEDENT**: Decremento de nivel de indentación
- Generados por preprocesador en **Main.java**

Expresiones

Jerarquía de Precedencia

```

expression
    : logic_or_expr
    ;

logic_or_expr
    : logic_and_expr (OR logic_and_expr)*

```

```

;

logic_and_expr
: equality_expr (AND equality_expr)*
;

equality_expr
: relational_expr ((EQ | NEQ) relational_expr)*
;

relational_expr
: additive_expr ((LT | GT | LE | GE) additive_expr)*
;

additive_expr
: multiplicative_expr ((PLUS | MINUS) multiplicative_expr)*
;

multiplicative_expr
: unary_expr ((MULT | DIV) unary_expr)*
;

unary_expr
: (MINUS | NOT) unary_expr
| primary_expr
;

```

Precedencia (mayor a menor):

1. Unarios: `-`, `not`
2. Multiplicativos: `*`, `/`
3. Aditivos: `+`, `-`
4. Relacionales: `<`, `>`, `<=`, `>=`
5. Igualdad: `==`, `!=`
6. AND lógico: `and`
7. OR lógico: `or`

Expresiones Primarias

```

primary_expr
: INT
| STRING
| TRUE
| FALSE
| ID
| func_call
| LPAREN expression RPAREN
;

func_call

```

```
: ID LPAREN (expression (COMMA expression)*)? RPAREN
;
```

Mapeo AST

Correspondencia Reglas → Nodos

| Regla Gramática | Nodo AST | Responsabilidad |
|---------------------|--------------|--|
| program | ProgNode | Programa completo |
| assign_stmt | AssignNode | Asignaciones |
| for_stmt | ForNode | Ciclos for |
| while_stmt | WhileNode | Ciclos while |
| if_stmt | IfNode | Condicionales if/elif/else |
| func_call | FuncCallNode | Llamadas a función |
| operadores binarios | BinaryOpNode | Operaciones +, -, *, /, <, >, <=, >=, ==, !=, and, or, % |
| operadores unarios | UnaryOpNode | Operaciones -, +, not |
| INT | IntNode | Números enteros |
| STRING | StringNode | Cadenas de texto |
| TRUE/FALSE | BoolNode | Valores booleanos |
| ID | VarRefNode | Referencias a variables |
| range_call | RangeNode | Función range() |

Construcción del AST

El **ASTBuilder** implementa el patrón Visitor de ANTLR para transformar el parse tree en AST:

```
// Ejemplo: visitFor_stmt()
@Override
public ASTNode visitFor_stmt(PythonSubsetParser.For_stmtContext ctx) {
    String variable = ctx.ID().getText();
    RangeNode iterable = (RangeNode) visit(ctx.range_call());
    List<ASTNode> body = new ArrayList<>();

    for (PythonSubsetParser.StatementContext stmt : ctx.suite().statement()) {
        body.add(visit(stmt));
    }

    return new ForNode(variable, iterable, body);
}
```

Extensiones de la Gramática

Proceso de Extensión

1. Modificar gramática

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

param_list
    : ID (',' ID)*
    ;
```

2. Regenerar parser

```
antlr4 grammar/PythonSubset.g4 -o src/main/antlr4/parser/
```

3. Crear nodo AST

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

4. Implementar visitor

```
@Override
public ASTNode visitFunc_def(PythonSubsetParser.Func_defContext ctx) {
    String name = ctx.ID(0).getText();
    List<String> params = extractParams(ctx.param_list());
    List<ASTNode> body = visitStatements(ctx.stmt());
    return new FuncDefNode(name, params, body);
}
```

Limitaciones Actuales

No Soportado en la Gramática

- **Funciones def:** Sin definición de funciones de usuario
- **Listas y tuplas:** Solo escalares (int, string, bool)
- **Dictionaries:** No soportados

- **Import statements:** Sin módulos
- **Clases:** Programación solo procedural
- **Exception handling:** try/except
- **Decoradores:** @decorator
- **Comprehensions:** List/dict/set comprehensions
- **Generators:** yield y funciones generadoras
- **Lambda:** Funciones lambda anónimas

Restricciones Sintácticas

- **Indentación fija:** Debe ser consistente (4 espacios o tabs)
- **Un statement por línea:** Sin ; para múltiples
- **Strings:** Comillas dobles "" o simples ' ' soportadas
- **Range limitado:** Solo range(stop) con un argumento
- **Print con un argumento:** print(x), no print(x, y)
- **Sin asignación compuesta:** No +=, -=, etc.
- **Comentarios ignorados:** Permitidos con # pero no afectan el AST

Ejemplos de Parsing

Programa Simple

```
x = 10
y = 20
print(x + y)
```

Parse Tree Resultante:

```
program
└── statement (assign_stmt: x = 10)
└── statement (assign_stmt: y = 20)
└── statement (expr_stmt: print(x + y))
```

Ciclo For

```
for i in range(3):
    print(i)
    x = i * 2
```

Parse Tree:

```
program
└── statement
```

```

└── compound_stmt
    └── for_stmt
        ├── ID: i
        ├── range_call: range(3)
        └── suite
            ├── INDENT
            ├── statement (print(i))
            ├── statement (x = i * 2)
            └── DEDENT

```

Condicional If/Elif/Else

```

x = 15

if x > 20:
    print("Mayor que 20")
elif x > 10:
    print("Mayor que 10")
elif x > 5:
    print("Mayor que 5")
else:
    print("5 o menor")

```

Parse Tree:

```

program
└── statement (assign_stmt: x = 15)
└── statement
    └── compound_stmt
        └── if_stmt
            ├── IF
            ├── expr (x > 20)
            └── suite (print("Mayor que 20"))
            └── elif_clause
                ├── expr (x > 10)
                └── suite (print("Mayor que 10"))
            └── elif_clause
                ├── expr (x > 5)
                └── suite (print("Mayor que 5"))
            └── else_clause
                └── suite (print("5 o menor"))

```

Expresión Compleja

```
resultado = (a + b) * 2 > 10 and not activo
```

Árbol de Expresión:

```
logic_and_expr
├── relational_expr
│   ├── multiplicative_expr
│   │   ├── additive_expr (a + b)
│   │   └── INT: 2
│   └── GT
└── INT: 10
└── AND
└── unary_expr
    ├── NOT
    └── ID: activo
```

Herramientas de Depuración

Visualizar Parse Tree

```
# Generar diagrama del árbol
antlr4 PythonSubset.g4 -gui
```

Probar Lexer

```
# Ver tokens generados
antlr4 PythonSubset.g4 -tokens input.py
```

Validar Gramática

```
# Verificar sintaxis de gramática
antlr4 PythonSubset.g4 -Werror
```

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, a and b</code> |

| Nodo | Propósito | Ejemplo |
|--------------|---------------------|--|
| UnaryOpNode | Operaciones unarias | -x, not y, +z |
| ForNode | Ciclos for | for i in range(10): |
| WhileNode | Ciclos while | while x < 5: |
| IfNode | Condicionales | if x > 5: ... elif x == 5: ... else: ... |
| FuncCallNode | Llamadas a función | print(x) |
| StringNode | Literales de cadena | "Hola mundo" |
| RangeNode | Iterables range | range(10), range(0, 10, 2) |

ASTBuilder

- **Responsabilidad:** Convierte parse tree en AST
- **Patrón:** Visor 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):\n    INDENT print(i)\n    DEDENT"
```

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
```

Sistema de dos fases:

1. **Primera pasada (collectStrings):** Recorre el AST completo identificando todos los `StringNode` y asignándoles etiquetas únicas basadas en hash
2. **Segunda pasada (generate):** Genera la sección `.data` con todos los strings y luego el código que los referencia

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 (actualmente stack-based)

- Plegado de constantes
- Short-circuit evaluation para operadores lógicos (actualmente evalúa ambos operandos)
- Loop unrolling para rangos conocidos en compile-time
- Inline de funciones print cuando sea posible

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("],");
        sb.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]
```

Ejemplos de Código - Python to x86_64 Compiler

Casos de Uso Básicos

1. Variables y Asignaciones

Ejemplo Simple

```
# Archivo: basic_variables.py
x = 42
mensaje = "Hola Mundo"
activo = True

print(x)
print(mensaje)
print(activo)
```

ASM Generado:

```
section .data
str12345: db "Hola Mundo", 0x0A, 0

section .text
global _start

_start:
; x = 42
mov rax, 42
mov [rbp-8], rax

; mensaje = "Hola Mundo"
lea rax, [str12345]
mov [rbp-16], rax

; activo = True
mov rax, 1
mov [rbp-24], rax

; print(x)
mov rax, [rbp-8]
call print_int

; print(mensaje)
mov rax, [rbp-16]
call print_string

; print(activo)
```

```
mov rax, [rbp-24]
call print_bool
```

2. Expresiones Aritméticas

Operaciones Básicas

```
# Archivo: arithmetic.py
a = 10
b = 5

suma = a + b
resta = a - b
multiplicacion = a * b
division = a / b

print(suma)
print(resta)
print(multiplicacion)
print(division)
```

Expresiones Complejas

```
# Archivo: complex_expressions.py
x = 2
y = 3
z = 4

# Precedencia de operadores
resultado1 = x + y * z      # = 2 + (3 * 4) = 14
resultado2 = (x + y) * z    # = (2 + 3) * 4 = 20
resultado3 = x * y + z * 2  # = (2 * 3) + (4 * 2) = 14

print(resultado1)
print(resultado2)
print(resultado3)
```

3. Expresiones Lógicas y Comparaciones

```
# Archivo: logical_operations.py
a = 10
b = 5
c = 10

# Comparaciones
mayor = a > b      # True
```

```
menor = a < b      # False
igual = a == c     # True
diferente = a != b # True

# Operadores lógicos
y_logico = mayor and igual      # True and True = True
o_logico = menor or diferente   # False or True = True
negacion = not menor             # not False = True

print(mayor)
print(menor)
print(igual)
print(diferente)
print(y_logico)
print(o_logico)
print(negacion)
```

Estructuras de Control

4. Ciclos For

For Simple

```
# Archivo: simple_for.py
print("Contando del 0 al 4:")
for i in range(5):
    print(i)
print("Fin del conteo")
```

Output Esperado:

```
Contando del 0 al 4:
0
1
2
3
4
Fin del conteo
```

For con Cálculos

```
# Archivo: for_calculations.py
suma_total = 0

print("Calculando suma de 0 a 9:")
for numero in range(10):
```

```
suma_total = suma_total + numero
print("Número:")
print(numero)
print("Suma parcial:")
print(suma_total)

print("Suma final:")
print(suma_total)
```

For con Expresiones

```
# Archivo: for_expressions.py
print("Tabla del 2:")
for i in range(1, 6): # No soportado aún - usar range(5)
    resultado = i * 2
    print(i)
    print(" * 2 = ")
    print(resultado)
```

5. Ciclos While

While Básico

```
# Archivo: simple_while.py
contador = 0

print("Contando con while:")
while contador < 5:
    print("Contador:")
    print(contador)
    contador = contador + 1

print("Fin del while")
```

While con Condiciones Complejas

```
# Archivo: complex_while.py
x = 1
limite = 100

print("Potencias de 2 menores a 100:")
while x < limite:
    print(x)
    x = x * 2
```

```
print("Última potencia:")
print(x)
```

While con Booleanos

```
# Archivo: while_boolean.py
continuar = True
contador = 0

while continuar:
    print("Iteración:")
    print(contador)
    contador = contador + 1

    # Condición de parada con if
    if contador == 3:
        continuar = False
```

6. Condicionales If/Elif/Else

If Simple

```
# Archivo: simple_if.py
x = 10

if x > 5:
    print("x es mayor que 5")

print("Fin del programa")
```

Output Esperado:

```
x es mayor que 5
Fin del programa
```

If con Else

```
# Archivo: if_else.py
edad = 18

if edad >= 18:
    print("Mayor de edad")
else:
    print("Menor de edad")
```

If/Elif/Else Completo

```
# Archivo: if_elif_else.py
calificacion = 85

if calificacion >= 90:
    print("Excelente")
elif calificacion >= 80:
    print("Muy bien")
elif calificacion >= 70:
    print("Bien")
elif calificacion >= 60:
    print("Suficiente")
else:
    print("Insuficiente")
```

Condicionales Anidados

```
# Archivo: nested_if.py
x = 15
y = 20

if x > 10:
    print("x es mayor que 10")
    if y > 15:
        print("y también es mayor que 15")
        if x + y > 30:
            print("La suma es mayor que 30")
    else:
        print("y no es mayor que 15")
else:
    print("x no es mayor que 10")
```

If con Expresiones Complejas

```
# Archivo: complex_if.py
x = 10
y = 5
z = 15

# Múltiples condiciones con and
if x > 5 and y < 10:
    print("Ambas condiciones son verdaderas")

# Múltiples condiciones con or
```

```
if x < 5 or z > 10:  
    print("Al menos una condición es verdadera")  
  
# Negación con not  
if not (x == y):  
    print("x no es igual a y")  
  
# Combinación compleja  
if (x > y) and (z > x) or (y == 0):  
    print("Expresión compleja evaluada como verdadera")
```

7. Combinando Estructuras

For Anidado (Conceptual)

```
# Archivo: nested_loops.py  
# NOTA: Anidamiento no completamente probado  
  
print("Tabla de multiplicar (conceptual):")  
for i in range(3):  
    print("Tabla del:")  
    print(i)  
    for j in range(3):  
        resultado = i * j  
        print(resultado)  
    print("---")
```

While y For Combinados

```
# Archivo: mixed_loops.py  
print("Ciclos combinados:")  
  
# For primero  
suma = 0  
for i in range(5):  
    suma = suma + i  
  
print("Suma del for:")  
print(suma)  
  
# Luego while  
contador = suma  
print("Contando hacia abajo:")  
while contador > 0:  
    print(contador)  
    contador = contador - 1  
  
print("Terminado")
```

Casos de Uso Avanzados

7. Simulación de Algoritmos

Factorial (con While)

```
# Archivo: factorial.py
numero = 5
factorial = 1
contador = 1

print("Calculando factorial de:")
print(numero)

while contador <= numero:
    factorial = factorial * contador
    print("Paso:")
    print(contador)
    print("Factorial parcial:")
    print(factorial)
    contador = contador + 1

print("Factorial final:")
print(factorial)
```

Fibonacci

```
# Archivo: fibonacci.py
n = 10
a = 0
b = 1
contador = 0

print("Serie Fibonacci:")
print(a)
print(b)

while contador < n:
    siguiente = a + b
    print(siguiente)
    a = b
    b = siguiente
    contador = contador + 1
```

Búsqueda Secuencial (Simulada)

```
# Archivo: search_simulation.py
# Simular búsqueda en "array" de 10 elementos
objetivo = 7
encontrado = False
posicion = 0

print("Buscando el número:")
print(objetivo)

while posicion < 10 and not encontrado:
    # Simular valor en posición (posición + 1)
    valor_actual = posicion + 1

    print("Revisando posición:")
    print(posicion)
    print("Valor:")
    print(valor_actual)

    if valor_actual == objetivo:
        encontrado = True
        print("¡Encontrado en posición:")
        print(posicion)
    else:
        posicion = posicion + 1

if not encontrado:
    print("No encontrado")
```

9. Validaciones y Testing

Testing de Operadores

```
# Archivo: operator_test.py
print("== Test de Operadores ==")

# Aritméticos
a = 15
b = 4

print("Suma:")
print(a + b)    # 19

print("Resta:")
print(a - b)    # 11

print("Multiplicación:")
print(a * b)    # 60

print("División:")
print(a / b)    # 3 (división entera)
```

```
# Comparaciones
print("Mayor que:")
print(a > b)      # True

print("Menor que:")
print(a < b)      # False

print("Igual:")
print(a == b)     # False

print("Diferente:")
print(a != b)     # True
```

Testing de Precedencia

```
# Archivo: precedence_test.py
print("== Test de Precedencia ==")

# Aritmética
resultado1 = 2 + 3 * 4
print("2 + 3 * 4 =")
print(resultado1)  # Debería ser 14

# Con paréntesis
resultado2 = (2 + 3) * 4
print("(2 + 3) * 4 =")
print(resultado2)  # Debería ser 20

# Lógica
a = True
b = False
c = True

resultado3 = a and b or c
print("True and False or True =")
print(resultado3)  # Debería ser True

# Comparación y lógica
x = 5
y = 10
resultado4 = x < y and y > 0
print("5 < 10 and 10 > 0 =")
print(resultado4)  # Debería ser True
```

Casos Edge y Limitaciones

10. Casos Límite

Números Grandes

```
# Archivo: large_numbers.py
grande = 1000000
print("Número grande:")
print(grande)

# Operaciones con números grandes
resultado = grande * 2
print("Doble:")
print(resultado)
```

Strings con Espacios

```
# Archivo: string_spaces.py
mensaje1 = "Hola mundo con espacios"
mensaje2 = "123 números en string"
mensaje3 = "" # String vacío

print(mensaje1)
print(mensaje2)
print(mensaje3)
```

Booleanos en Expresiones

```
# Archivo: boolean_expressions.py
verdadero = True
falso = False

# Booleanos como números (0 y 1)
suma_bool = verdadero + falso
print("True + False =")
print(suma_bool) # Debería ser 1

# En comparaciones
es_verdadero = verdadero == True
print("True == True =")
print(es_verdadero) # Debería ser True
```

11. Características No Soportadas

Ejemplos que NO Funcionan

```
# ESTOS EJEMPLOS NO FUNCIONAN AÚN

# 1. Funciones definidas por usuario
    return parametro * 2

# 2. Listas
mi_lista = [1, 2, 3, 4, 5]

# 3. Range con múltiples parámetros
for i in range(1, 10, 2): # Solo range(stop) funciona
    print(i)

# 4. Asignación compuesta
x += 5
y *= 2

# 5. Múltiples asignaciones
a, b = 1, 2

# 6. Operaciones con strings
mensaje = "Hola" + " " + "Mundo" # No soportado
substring = mensaje[0:5]           # No soportado

# 7. Print con múltiples argumentos
print("Valor:", x) # Solo print(x) funciona

# Nota: Los comentarios SÍ funcionan y son ignorados correctamente
```

Comandos de Testing

Ejecutar Ejemplos

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

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

# Testing batch (todos los ejemplos)
for file in src/test/*.py; do
    echo "Testing $file"
    java -cp "build:lib/*" parser.Main "$file"
    if [ $? -eq 0 ]; then
        echo "✓ Compilación exitosa"
    else
        echo "✗ Error en compilación"
    fi
done
```

Output Esperado

Cada ejemplo debería:

1. **Compilar sin errores:** No excepciones Java
2. **Generar ASM válido:** Archivo `build/ejemplo.asm` creado
3. **Ensamblar correctamente:** NASM sin errores
4. **Ejecutar y producir output:** Resultado coherente con código Python

Verificación de Resultados

```
# Ejemplo para verificar arithmetic.py
echo "Expected: 15, 5, 50, 2"
./build/programa
# Verificar que el output coincida
```

La documentación de ejemplos proporciona una guía completa para usuarios y desarrolladores sobre qué es posible hacer con el compilador actual y cómo estructurar código Python compatible.