

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
```