UNIVERSIDAD CATÓLICA DE SALTA

Facultad de Ingeniería - Ingeniería en Informática

CÁTEDRA: Compiladores - Año 2025

PROYECTO INTEGRADOR - INFORME FINAL

TEMA: Compilador para un subconjunto de Python

Grupo:	N° 26
Integrantes:	_____ (Legajo/DNI: _____)
	_____ (Legajo/DNI: _____)
Docentes:	_____
Fecha de entrega:	____ / ____ / 2026

Nota: Este informe sigue la estructura mínima solicitada en el enunciado oficial del proyecto integrador.

Índice

1. Introducción

El presente informe documenta el desarrollo de un analizador léxico y sintáctico para un subconjunto del lenguaje Python, construido con la librería **PLY (Python Lex-Yacc)**. El objetivo del proyecto es implementar las primeras fases de un compilador: identificar componentes léxicos (tokens), aplicar una gramática para reconocer estructuras válidas, e informar errores léxicos y/o sintácticos indicando el número de línea correspondiente.

Correcciones incorporadas (según devoluciones previas de la cátedra):

- No incluir saltos de línea como símbolos gramaticales del lenguaje (se tratan como separadores/ayuda de implementación).
- En la gramática, utilizar los nombres de tokens definidos por el analizador léxico.
- Evitar renombrar no terminales: se conserva la nomenclatura utilizada en la implementación para facilitar trazabilidad.

2. Alcance del lenguaje fuente

El lenguaje fuente implementado corresponde al subconjunto de Python asignado al **Grupo 26**. Se incluyen: Sentencias **DEF** con exactamente **2 parámetros**. Asignaciones: **=**, **+=**, **-=**. Tipos literales: **INT**, **FLOAT**, **STRING**. Funciones: **print**, **len**, **round**. Operadores aritméticos: **+**, **-**, *****, **/**. Operadores lógicos: **and**, **or**, **not**. Operadores de comparación: **==**, **!=**, **<**, **>**. Comentarios con # (ignorados por el lexer). Además, para poder reconocer bloques indentados (cuerpo de funciones), la implementación utiliza tokens auxiliares de control de indentación (**INDENT**/**DEDENT**).

3. Tokens del lenguaje

A continuación se detalla el listado de tokens definidos en el analizador léxico, indicando su significado y la expresión regular (o regla) asociada.

Token	Descripción	Expresión regular / Regla
DEF	Palabra reservada: definición de función (def)	'def' (detectado en t_NAME)
IF	Palabra reservada: if (no utilizado en la gramática actual)	'if' (detectado en t_NAME)
PRINT	Función incorporada: print	'print' (detectado en t_NAME)
LEN	Función incorporada: len	'len' (detectado en t_NAME)
ROUND	Función incorporada: round	'round' (detectado en t_NAME)
AND	Operador lógico: and	'and' (detectado en t_NAME)
OR	Operador lógico: or	'or' (detectado en t_NAME)
NOT	Operador lógico: not	'not' (detectado en t_NAME)
NAME	Identificador (variable/nombre de función)	[a-zA-Z_][a-zA-Z0-9_]*
INT	Entero	\d+
FLOAT	Número en punto flotante	\d+\.\d+([eE][+-]?\d+)?
STRING	Cadena entre comillas simples o dobles	'(\"([^\\\n]|(\\.))*\")|(\'([^\\\n]|(\\.))*\')'
PLUS	Operador +	\+
MINUS	Operador -	-

Token	Descripción	Expresión regular / Regla
TIMES	Operador *	*
DIVIDE	Operador /	/
EQUAL	Asignación =	=
PLUSEQ	Asignación acumulativa +=	\+=
MINUSEQ	Asignación acumulativa -=	-=
EQEQ	Comparación ==	==
NEQ	Comparación !=	!=
LT	Comparación <	<
GT	Comparación >	>
LPAREN	Paréntesis izquierdo (	\(
RPAREN	Paréntesis derecho)	\)
COMMA	Separador de argumentos ,	,
COLON	Dos puntos :	:
NEWLINE	Separador de líneas (uso interno para conteo y control de indentación)	\n+
INDENT	Aumento de indentación (inicio de bloque)	(ver implementación)
DEDENT	Disminución de indentación (fin de bloque)	(ver implementación)

Observación: Los tokens **NEWLINE**, **INDENT** y **DEDENT** se emplean como soporte de implementación para bloques indentados y para reportar líneas; no constituyen construcciones “semánticas” del lenguaje fuente, por lo que en la sección de gramática se evita tratarlos como elementos de la definición conceptual del lenguaje.

4. Gramática del lenguaje

Se presenta la gramática en forma de producciones, utilizando los mismos **no terminales** que en la implementación (parser.py) para evitar renombramientos. Asimismo, se evita introducir “saltos de línea” como símbolos del lenguaje; la separación por líneas se considera un detalle de implementación.

Producciones:

```
program : stmt_list
stmt_list : stmt_list stmt_line | stmt_line
stmt_line : simple_stmt | funcdef | empty_line
funcdef : DEF NAME LPAREN NAME COMMA NAME RPAREN COLON stmt_block
stmt_block : stmt_block stmt_line | stmt_line
simple_stmt : assign_stmt | expr_stmt
assign_stmt : NAME assign_op expr
assign_op : EQUAL | PLUSEQ | MINUSEQ
expr_stmt : expr
expr : or_expr
or_expr : and_expr | or_expr OR and_expr
and_expr : not_expr | and_expr AND not_expr
not_expr : NOT not_expr | comparison
comparison : arith_expr | arith_expr comp_op arith_expr
comp_op : EQEQ | NEQ | LT | GT
arith_expr : term | arith_expr PLUS term | arith_expr MINUS term
term : factor | term TIMES factor | term DIVIDE factor
factor : MINUS factor | atom
atom : NAME | literal | LPAREN expr RPAREN | call
call : NAME LPAREN arglist_opt RPAREN
     | PRINT LPAREN arglist_opt RPAREN
     | LEN LPAREN arglist_opt RPAREN
     | ROUND LPAREN arglist_opt RPAREN
arglist_opt : empty | arglist
arglist : expr | arglist COMMA expr
literal : INT | FLOAT | STRING
empty : ε
empty_line : ε
```

Nota de implementación de bloques: En el código (parser.py) la producción *funcdef* incluye tokens auxiliares **NEWLINE INDENT ... DEDENT** para representar el inicio/fin de un bloque indentado, ya que en Python la indentación es significativa. En el reporte conceptual se agrupan como parte de la estructura del bloque, sin considerarlos parte del lenguaje a nivel de gramática “abstracta”.

5. Implementación (Lex y Yacc con PLY)

5.1 Analizador léxico (lexer.py)

Se implementó con **ply.lex**. El lexer reconoce operadores, delimitadores, literales e identificadores. Las palabras reservadas se resuelven a partir del token **NAME** mediante un diccionario (*reserved*), mapeando lexemas como *def*, *print*, *and*, etc. Los comentarios iniciados con # se ignoran.

Manejo de indentación: Para simular el comportamiento de Python, se incluye un filtro (*IndentLexer*) que, luego de cada **NEWLINE**, calcula la cantidad de espacios/tabs al inicio de la siguiente línea y genera tokens **INDENT** o **DEDENT** cuando corresponde.

5.2 Analizador sintáctico (parser.py)

Se implementó con **ply.yacc** a partir de las producciones definidas en funciones *p_**. Para reflejar precedencia y asociatividad de operadores se utilizaron no terminales por niveles (*or_expr*, *and_expr*, *not_expr*, *comparison*, *arith_expr*, *term*, *factor*, *atom*), evitando ambigüedades. La definición de función (*funcdef*) restringe el encabezado a exactamente dos parámetros, según el enunciado.

5.3 Aplicación principal (main.py)

La aplicación recibe por línea de comandos un archivo fuente. Ejecuta dos fases: Tokenización completa del archivo (fase léxica), imprimiendo los tokens generados. Análisis sintáctico invocando el parser con el lexer utilizado. Al finalizar, informa si el programa es válido o lista los errores detectados.

6. Manejo de errores

Errores léxicos: Se reportan en *t_error* indicando el carácter ilegal y el número de línea.
Errores sintácticos: Se manejan en *p_error*. Cuando se encuentra un token inesperado se informa el valor y el tipo de token, junto con **p.lineno**. Si el error ocurre al final del archivo se informa como “fin de archivo inesperado”. Los mensajes se almacenan en una lista global (*errors*) para que *main.py* pueda decidir si el programa es válido.

7. Archivos de prueba

Se incluyen tres archivos de ejemplo, cumpliendo el requisito de presentar casos con error léxico, error sintáctico y un programa válido. En cada caso se comentan los errores en el propio archivo.

7.1 Programa con errores léxicos (lexico_error.py)

```
# Este archivo contiene caracteres ilegales

def funcion_mala(a, b):
    res = a + b @ 2  # @ no es un operador válido
    print(res)

valor = 10 $ 5      # $ tampoco es válido
```

7.2 Programa con errores sintácticos (sintactico_error.py)

```
# Este archivo contiene errores de sintaxis

def sin_dos_puntos(a, b)  # Falta :
    print(a)

def indentacion_mal(a, b):
print(a) # Falta indentación

x = 10 +  # Expresión incompleta
```

7.3 Programa válido (valido.py)

```
def suma(a, b):
    total = a + b
    print(total)

x = 10
y = 20
suma(x, y)

def promedio(a, b):
    res = (a + b) / 2
    print(res)

val1 = 50
val2 = 100
promedio(val1, val2)
```

8. Ejecución de la aplicación (con capturas)

Requisitos: Python 3.x y la librería *ply* instalada (*pip install ply*).
Ejecución: desde una terminal, ubicados en la carpeta del proyecto:

```
python main.py <archivo_fuente>

# Ejemplos:
python main.py valido.py
python main.py lexico_error.py
python main.py sintactico_error.py
```

A continuación se dejan espacios para incorporar capturas de pantalla reales de la ejecución y su salida.

ESPACIO PARA CAPTURA 1: Ejecución con archivo valido.py (tokens + análisis sintáctico exitoso)

ESPACIO PARA CAPTURA 2: Ejecución con archivo lexico_error.py (mensaje(s) de error léxico)

ESPACIO PARA CAPTURA 3: Ejecución con archivo sintactico_error.py (mensaje(s) de error sintáctico)

9. Conclusiones

Se logró implementar un analizador léxico y sintáctico funcional para el subconjunto de Python asignado, utilizando PLY como herramienta de construcción. Entre los puntos fuertes se destaca la separación clara entre fases (lexer/parser) y el reporte de errores con número de línea. Como aspecto a mejorar, se puede enriquecer el manejo de recuperación de errores sintácticos (para continuar luego de un error) y ampliar el conjunto de sentencias soportadas si el alcance del lenguaje fuente se extendiera.

10. Bibliografía (normas APA)

- Beazley, D. (s. f.). *PLY (Python Lex-Yacc)*. Recuperado de la documentación oficial del proyecto.
- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.
- Universidad Católica de Salta. (2025). *Proyecto Integrador - Cátedra de Compiladores (Grupo 26) - Tema 26*. Material provisto por la cátedra.

11. Anexos: códigos fuente

En este anexo se adjuntan los archivos principales del proyecto.

11.1 lexer.py

```
import ply.lex as lex

# --- Definición de Tokens ---

# Palabras reservadas
reserved = {
    'def': 'DEF',
    'if': 'IF',       # Aunque no estén explícitamente en el ejemplo mínimo, son comunes.
                      # El enunciado dice "Sentencias DEF", pero la gramática no muestra IF.
                      # Me ceñiré a lo que pide el enunciado estrictamente: DEF, PRINT, LEN, ROUND
    'print': 'PRINT',
    'len': 'LEN',
    'round': 'ROUND',
    'and': 'AND',
    'or': 'OR',
    'not': 'NOT'
}

tokens = [
    'NAME', 'INT', 'FLOAT', 'STRING',
    'PLUS', 'MINUS', 'TIMES', 'DIVIDE',
    'EQUAL', 'PLUSEQ', 'MINUSEQ',
    'EQEQ', 'NEQ', 'LT', 'GT',
    'LPAREN', 'RPAREN', 'COMMA', 'COLON',
    'NEWLINE', 'INDENT', 'DEDENT'
] + list(reserved.values())

# --- Expresiones Regulares Simples ---

t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_EQUAL   = r'='
t_PLUSEQ  = r'\+='
t_MINUSEQ = r'-='
t_EQEQ    = r'=='
t_NEQ     = r'!='
t_LT      = r'<'
t_GT      = r'>'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'
t_COMMA   = r','
t_COLON   = r':'

# --- Expresiones Regulares con Acción ---

def t_FLOAT(t):
    r'\d+\.\d+([eE][+-]?\d+)?'
    t.value = float(t.value)
    return t

def t_INT(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_STRING(t):
    r'(\"([^\\\n]|(\\.))*\")|(\'([^\\\n]|(\\.))*\')'
    t.value = t.value[1:-1] # Eliminar comillas
```

```
    return t

def t_NAME(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    t.type = reserved.get(t.value, 'NAME') # Verificar palabras reservadas
    return t

def t_COMMENT(t):
    r'\#.*'
    pass # Ignorar comentarios

# Definir NEWLINE para rastrear números de línea
def t_NEWLINE(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
    t.type = 'NEWLINE'
    # Para el manejo de indentación, necesitamos devolver este token
    return t

# Ignorar espacios y tabs (el manejo de indentación se hace aparte)
# OJO: PLY ignora t_ignore characters automáticamente, pero para Python
# necesitamos ver los espacios al inicio de la línea.
# Estrategia: Ignorar espacios dentro de la línea, pero NO el salto de línea.
# La indentación se calcula midiendo espacios después de un NEWLINE.
t_ignore = ' \t'

def t_error(t):
    print(f"Error léxico: Carácter ilegal '{t.value[0]}' en línea {t.lineno}")
    t.lexer.skip(1)

# --- Filtro de Indentación ---

class IndentLexer(object):
    def __init__(self, lexer):
        self.lexer = lexer
        self.token_stream = None

    def input(self, data):
        self.lexer.input(data)
        self.token_stream = self.filter_tokens(self.lexer)

    def token(self):
        try:
            return next(self.token_stream)
        except StopIteration:
            return None

    def filter_tokens(self, lexer):
        indent_stack = [0]
        tokens = iter(lexer.token, None)

        for token in tokens:
            yield token
            if token.type == 'NEWLINE':
                # Mirar el siguiente token para ver la indentación
                # Necesitamos "espiar" el input del lexer para ver los espacios
                # Pero PLY ya se comió los espacios por t_ignore.
                # ESTRATEGIA ALTERNATIVA TÍPICA PARA PLY+PYTHON:
                # Usar un estado o revisar lexer.lexdata en la posición actual.

                # Vamos a calcular la indentación manualmente leyendo lexdata
                # desde lexpos hasta que encontremos algo que no sea espacio.

                current_indent = 0
                pos = lexer.lexpos
                data = lexer.lexdata
                while pos < len(data):
```

```
            char = data[pos]
            if char == ' ':
                current_indent += 1
                pos += 1
            elif char == '\t':
                current_indent += 4 # Asumimos tab = 4 espacios o ajuste según pref
                pos += 1
            elif char == '\n':
                # Línea vacía con espacios, reiniciamos conteo para la sig línea
                current_indent = 0
                pos += 1
                # Opcional: emitir otro NEWLINE si queremos preservar líneas vacías
                # pero la gramática suele ignorarlas o colapsarlas.
                # Ajustar lineno manual si saltamos \n extra
                lexer.lineno += 1
            elif char == '#':
                # Comentario al inicio de linea o indentado, lo saltamos hasta el final de
                while pos < len(data) and data[pos] != '\n':
                    pos += 1
                # Si encontramos \n, se procesará en la siguiente iteración del while exte
                # Pero aquí estamos dentro del cálculo de indent.
                # Simplemente continuamos el bucle de indentación desde el \n
                if pos < len(data) and data[pos] == '\n':
                    # current_indent = 0 # Reset para la próxima línea
                    # pos += 1
                    # lexer.lineno += 1
                    # En realidad, si hay comentario, esa linea no cuenta para indent char
                    # mejor dejar que el lexer normal lo procese?
                    # El problema es que t_ignore se come los espacios antes del #.
                    pass

                # Simplificación: Si es comentario, ignorar esta línea para propósitos de
                # y tratar como si tuviera la misma indentación que la anterior o simpleme
                # no generar tokens INDENT/DEDENT.

                # Lo más fácil: dejar que el lexer siga.
                break
            else:
                break

        # Si llegamos al final del archivo con espacios, ignorar
        if pos >= len(data):
            break

        # Comprobar niveles
        if current_indent > indent_stack[-1]:
            indent_stack.append(current_indent)
            t = lex.LexToken()
            t.type = 'INDENT'
            t.value = current_indent
            t.lineno = token.lineno
            t.lexpos = pos
            yield t
        elif current_indent < indent_stack[-1]:
            while current_indent < indent_stack[-1]:
                indent_stack.pop()
                t = lex.LexToken()
                t.type = 'DEDENT'
                t.value = indent_stack[-1]
                t.lineno = token.lineno
                t.lexpos = pos
                yield t
            if current_indent != indent_stack[-1]:
                print(f"Error de Indentación en línea {token.lineno}")

        # NO actualizamos lexer.lexpos manualmente aquí porque el lexer
        # normal de PLY se saltará los espacios gracias a t_ignore.
```

```
                # Solo inyectamos los tokens.

        # Al final del archivo, vaciar la pila de indentación
        while len(indent_stack) > 1:
            indent_stack.pop()
            t = lex.LexToken()
            t.type = 'DEDENT'
            t.value = 0
            t.lineno = lexer.lineno
            t.lexpos = lexer.lexpos
            yield t

# Construir el lexer básico
lexer_base = lex.lex()

# Envolverlo
lexer = IndentLexer(lexer_base)
```

11.2 parser.py

```
import ply.yacc as yacc
from lexer import tokens

# Lista para acumular errores
errors = []

# --- Reglas Gramaticales ---

# Regla inicial
def p_program(p):
    '''program : stmt_list'''
    print("Análisis Sintáctico Exitoso: El programa es correcto.")
    p[0] = p[1]

# Lista de sentencias
def p_stmt_list(p):
    '''stmt_list : stmt_list stmt_line
                 | stmt_line'''
    if len(p) == 3:
        p[0] = p[1] + [p[2]]
    else:
        p[0] = [p[1]]

# Línea de sentencia
def p_stmt_line(p):
    '''stmt_line : simple_stmt NEWLINE
                 | funcdef
                 | NEWLINE'''
    if len(p) == 3: # simple_stmt NEWLINE
        p[0] = p[1]
    elif len(p) == 2: # funcdef or NEWLINE
        if p.slice[1].type == 'NEWLINE':
            p[0] = None
        else:
            p[0] = p[1]

# Definición de función
def p_funcdef(p):
    '''funcdef : DEF NAME LPAREN NAME COMMA NAME RPAREN COLON NEWLINE INDENT stmt_block DEDENT'''
    # Se especifica que DEF toma exactamente 2 parámetros según el enunciado
    p[0] = ('func_def', p[2], [p[4], p[6]], p[11])

# Bloque de sentencias (dentro de función)
def p_stmt_block(p):
    '''stmt_block : stmt_block stmt_line
                  | stmt_line'''
```

```
    if len(p) == 3:
        p[0] = p[1] + [p[2]]
    else:
        p[0] = [p[1]]

# Sentencias simples
def p_simple_stmt(p):
    '''simple_stmt : assign_stmt
                   | expr_stmt'''
    p[0] = p[1]

# Asignación
def p_assign_stmt(p):
    '''assign_stmt : NAME assign_op expr'''
    p[0] = ('assign', p[1], p[2], p[3])

def p_assign_op(p):
    '''assign_op : EQUAL
                 | PLUSEQ
                 | MINUSEQ'''
    p[0] = p[1]

# Expresión como sentencia (ej. llamada a función)
def p_expr_stmt(p):
    '''expr_stmt : expr'''
    p[0] = p[1]

# --- Expresiones ---

def p_expr(p):
    '''expr : or_expr'''
    p[0] = p[1]

def p_or_expr(p):
    '''or_expr : and_expr
               | or_expr OR and_expr'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = ('or', p[1], p[3])

def p_and_expr(p):
    '''and_expr : not_expr
                | and_expr AND not_expr'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = ('and', p[1], p[3])

def p_not_expr(p):
    '''not_expr : NOT not_expr
                | comparison'''
    if len(p) == 3:
        p[0] = ('not', p[2])
    else:
        p[0] = p[1]

def p_comparison(p):
    '''comparison : arith_expr
                  | arith_expr comp_op arith_expr'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = ('comparison', p[2], p[1], p[3])

def p_comp_op(p):
    '''comp_op : EQEQ
```

```
                | NEQ
                | LT
                | GT'''
    p[0] = p[1]

def p_arith_expr(p):
    '''arith_expr : term
                  | arith_expr PLUS term
                  | arith_expr MINUS term'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = ('arith', p[2], p[1], p[3])

def p_term(p):
    '''term : factor
            | term TIMES factor
            | term DIVIDE factor'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = ('term', p[2], p[1], p[3])

def p_factor(p):
    '''factor : MINUS factor
              | atom'''
    if len(p) == 3:
        p[0] = ('unary_minus', p[2])
    else:
        p[0] = p[1]

def p_atom(p):
    '''atom : NAME
            | literal
            | LPAREN expr RPAREN
            | call'''
    if len(p) == 2:
        p[0] = p[1] # NAME, literal, call
    else:
        p[0] = p[2] # (expr)

def p_call(p):
    '''call : NAME LPAREN arglist_opt RPAREN
            | PRINT LPAREN arglist_opt RPAREN
            | LEN LPAREN arglist_opt RPAREN
            | ROUND LPAREN arglist_opt RPAREN'''
    p[0] = ('call', p[1], p[3])

def p_arglist_opt(p):
    '''arglist_opt : empty
                   | arglist'''
    p[0] = p[1] if p[1] is not None else []

def p_arglist(p):
    '''arglist : expr
               | arglist COMMA expr'''
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[3]]

def p_literal(p):
    '''literal : INT
               | FLOAT
               | STRING'''
    p[0] = ('literal', p[1])
def p_empty(p):
```

```
    '''empty :'''
    pass

# --- Manejo de Errores ---

def p_error(p):
    if p:
        msg = f"Error sintáctico en línea {p.lineno}: Se encontró token inesperado '{p.value}' ({p
        print(msg)
        errors.append(msg)
    else:
        msg = "Error sintáctico: Fin de archivo inesperado (posiblemente falta cerrar paréntesis o
        print(msg)
        errors.append(msg)

# Construir el parser
parser = yacc.yacc()
```

11.3 main.py

```
import sys
from lexer import lexer
from parser import parser

def main():
    if len(sys.argv) < 2:
        print("Uso: python main.py <archivo_fuente>")
        return

    filename = sys.argv[1]
    try:
        with open(filename, 'r') as f:
            data = f.read()
    except FileNotFoundError:
        print(f"Error: No se encontró el archivo '{filename}'")
        return

    print(f"--- Iniciando Análisis de: {filename} ---")

    # 1. Análisis Léxico (opcional: mostrar tokens)
    print("\n[Fase 1: Análisis Léxico]")
    lexer.input(data)

    # Iterar sobre tokens para mostrar errores léxicos si los hay
    # Nota: El lexer ya imprime "Error léxico" en caso de problemas.
    token_list = []
    while True:
        tok = lexer.token()
        if not tok:
            break
        token_list.append(tok)
        print(tok)

    # Reiniciar lexer para el parser
    lexer.input(data)

    # Limpiar errores previos del módulo parser (si los hubiera)
    import parser as parser_mod
    parser_mod.errors.clear()

    # 2. Análisis Sintáctico
    print("\n[Fase 2: Análisis Sintáctico]")
    result = parser.parse(data, lexer=lexer)

    if not parser_mod.errors:
        print("\nResultado del análisis (Estructura interna):")
```

```
            print(result)
            print("\n>>> El programa es léxica y sintácticamente CORRECTO. <<<")
        else:
            print(f"\n>>> Se encontraron {len(parser_mod.errors)} error(es) sintáctico(s). <<<")
            print(">>> El programa NO es válido. <<<")

if __name__ == '__main__':
    main()
```

Nota: El archivo *parsetab.py* es generado automáticamente por PLY durante la construcción del parser y puede variar entre ejecuciones; se incluye en la entrega del proyecto, pero no es necesario modificarlo manualmente.