1. Teoria Fondamentale

1.1 Concetti Base

Il processo di interpretazione di un linguaggio si suddivide concettualmente in fasi sequenziali:

```
Testo Sorgente → [LEXER] → Token → [PARSER] → AST → [INTERPRETE] → Risultato
```

Lexer (Analizzatore Lessicale)

- Input: Sequenza di caratteri
- Output: Sequenza di token
- Funzione: Riconoscimento di pattern lessicali tramite espressioni regolari
- Elimina: Whitespace, commenti, caratteri irrilevanti

Parser (Analizzatore Sintattico)

- Input: Sequenza di token
- Output: Abstract Syntax Tree (AST)
- Funzione: Verifica conformità grammaticale e costruzione struttura gerarchica
- Gestisce: Precedenza operatori, associatività, annidamento

1.2 Elementi Tecnici

Token

```
class Token:
    def __init__(self, type, value, position):
        self.type = type      # Categoria lessicale (NUMBER, IDENTIFIER,
etc.)
        self.value = value    # Valore specifico ("42", "variabile", etc.)
        self.position = position  # Posizione nel testo sorgente per debug
```

Grammatiche

- **Context-Free Grammar (CFG)**: Definisce sintassi del linguaggio
- **BNF (Backus-Naur Form)**: Notazione standard per grammatiche
- **Precedenza**: Ordine di valutazione degli operatori
- **Associatività**: Direzione di valutazione (sinistra/destra)

Algoritmi di Parsing

1. **Recursive Descent**: Top-down, intuitivo, limitato a grammatiche LL
2. **LR/LALR**: Bottom-up, potente, gestisce più grammatiche
3. **PEG**: Parsing Expression Grammars, alternative moderne

---

2. Implementazione da Zero - Esempio Completo

2.1 Lexer Manuale

```python
#!/usr/bin/env python3
"""
Lexer implementato manualmente per espressioni aritmetiche
Supporta: numeri, operatori (+, -, *, /), parentesi, variabili
"""

import re
from enum import Enum, auto
from dataclasses import dataclass
from typing import List, Optional

class TokenType(Enum):
    # Literals
    NUMBER = auto()
    IDENTIFIER = auto()

    # Operators
    PLUS = auto()
    MINUS = auto()
    MULTIPLY = auto()
    DIVIDE = auto()
    ASSIGN = auto()

    # Delimiters
    LPAREN = auto()
    RPAREN = auto()
    SEMICOLON = auto()

    # Special
    EOF = auto()

@dataclass
class Token:
    type: TokenType
    value: str
    line: int
    column: int
```

```
class Lexer:
    def __init__(self, text: str):
        self.text = text
        self.position = 0
        self.line = 1
        self.column = 1

        # Pattern lessicali ordinati per priorità
        self.patterns = [
            # Numeri (interi e decimali)
            (r'\d+\.\d+', TokenType.NUMBER),
            (r'\d+', TokenType.NUMBER),

            # Identificatori e parole chiave
            (r'[a-zA-Z_][a-zA-Z0-9_]*', TokenType.IDENTIFIER),

            # Operatori multi-carattere (prima dei singoli)
            (r'==', TokenType.ASSIGN),  # Esempio per estensioni future

            # Operatori singoli
            (r'\+', TokenType.PLUS),
            (r'-', TokenType.MINUS),
            (r'\*', TokenType.MULTIPLY),
            (r'/', TokenType.DIVIDE),
            (r'=', TokenType.ASSIGN),

            # Delimitatori
            (r'\(', TokenType.LPAREN),
            (r'\)', TokenType.RPAREN),
            (r';', TokenType.SEMICOLON),

            # Whitespace (pattern speciale, None indica "ignora")
            (r'[ \t]+', None),
            (r'\n', 'NEWLINE'),  # Gestione speciale per conteggio righe
        ]

        # Compila regex per performance
        self.compiled_patterns = [
            (re.compile(pattern), token_type)
            for pattern, token_type in self.patterns
        ]

    def current_char(self) -> Optional[str]:
        """Carattere alla posizione corrente"""
        return self.text[self.position] if self.position < len(self.text)
else None

    def peek_char(self, offset: int = 1) -> Optional[str]:
        """Guarda avanti senza consumare"""
        pos = self.position + offset
```

```
        return self.text[pos] if pos < len(self.text) else None

    def advance(self):
        """Avanza di un carattere aggiornando posizione"""
        if self.position < len(self.text):
            if self.text[self.position] == '\n':
                self.line += 1
                self.column = 1
            else:
                self.column += 1
            self.position += 1

    def make_token(self, token_type: TokenType, value: str, start_col: int)
-> Token:
        """Crea token con informazioni di posizione"""
        return Token(token_type, value, self.line, start_col)

    def skip_whitespace(self):
        """Salta spazi e tab"""
        while self.current_char() and self.current_char() in ' \t':
            self.advance()

    def tokenize(self) -> List[Token]:
        """Converte testo in lista di token"""
        tokens = []

        while self.position < len(self.text):
            start_position = self.position
            start_column = self.column
            match_found = False

            # Prova ogni pattern in ordine
            for regex, token_type in self.compiled_patterns:
                match = regex.match(self.text, self.position)

                if match:
                    value = match.group(0)
                    end_position = match.end()

                    # Aggiorna posizione
                    while self.position < end_position:
                        self.advance()

                    # Gestisci token speciali
                    if token_type == 'NEWLINE':
                        # Non aggiungere token, solo aggiorna contatori
                        pass
                    elif token_type is not None:
                        # Converti valore se necessario
                        if token_type == TokenType.NUMBER:
```

```
                            # Mantieni come stringa, conversione nel parser
                            pass

                        tokens.append(self.make_token(token_type, value,
start_column))

                    match_found = True
                    break

            if not match_found:
                char = self.current_char()
                raise SyntaxError(
                    f"Carattere non riconosciuto: '{char}' "
                    f"alla riga {self.line}, colonna {self.column}"
                )

        # Aggiungi EOF
        tokens.append(self.make_token(TokenType.EOF, '', self.column))
        return tokens

# Test del lexer
def test_lexer():
    code = """
    x = 10 + 5;
    y = (x * 2) / 3.14;
    result = x + y;
    """

    lexer = Lexer(code)
    tokens = lexer.tokenize()

    print("=== TOKEN GENERATI ===")
    for token in tokens:
        if token.type != TokenType.EOF:
            print(f"{token.type.name:12} | {token.value:8} |
L{token.line}:C{token.column}")
```

2.2 Parser Manuale con AST

```
"""
Parser Recursive Descent per grammatica:

program     → statement*
statement   → assignment | expression ';'
assignment  → IDENTIFIER '=' expression ';'
expression  → term (('+'|'-') term)*
term        → factor (('*'|'/') factor)*
factor      → NUMBER | IDENTIFIER | '(' expression ')'
```

```
"""

from abc import ABC, abstractmethod
from typing import Union, Dict

# === NODI AST ===

class ASTNode(ABC):
    """Classe base per tutti i nodi dell'AST"""
    pass

class NumberNode(ASTNode):
    def __init__(self, value: float):
        self.value = value

    def __repr__(self):
        return f"Num({self.value})"

class IdentifierNode(ASTNode):
    def __init__(self, name: str):
        self.name = name

    def __repr__(self):
        return f"Id({self.name})"

class BinaryOpNode(ASTNode):
    def __init__(self, left: ASTNode, operator: Token, right: ASTNode):
        self.left = left
        self.operator = operator
        self.right = right

    def __repr__(self):
        return f"BinOp({self.left} {self.operator.value} {self.right})"

class AssignmentNode(ASTNode):
    def __init__(self, identifier: str, expression: ASTNode):
        self.identifier = identifier
        self.expression = expression

    def __repr__(self):
        return f"Assign({self.identifier} = {self.expression})"

class ProgramNode(ASTNode):
    def __init__(self, statements: List[ASTNode]):
        self.statements = statements

    def __repr__(self):
        return f"Program({self.statements})"

# === PARSER ===
```

```
class Parser:
    def __init__(self, tokens: List[Token]):
        self.tokens = tokens
        self.position = 0
        self.current_token = self.tokens[0] if tokens else None

    def error(self, message: str):
        """Solleva errore di parsing con informazioni di contesto"""
        if self.current_token:
            raise SyntaxError(
                f"{message} alla riga {self.current_token.line}, "
                f"colonna {self.current_token.column}. "
                f"Token trovato: {self.current_token.type.name}
'{self.current_token.value}'"
            )
        else:
            raise SyntaxError(f"{message} (EOF)")

    def advance(self):
        """Passa al token successivo"""
        self.position += 1
        if self.position < len(self.tokens):
            self.current_token = self.tokens[self.position]
        else:
            self.current_token = None

    def expect(self, token_type: TokenType) -> Token:
        """Consuma token del tipo specificato o solleva errore"""
        if self.current_token and self.current_token.type == token_type:
            token = self.current_token
            self.advance()
            return token
        else:
            expected = token_type.name
            found = self.current_token.type.name if self.current_token else
"EOF"
            self.error(f"Atteso {expected}, trovato {found}")

    def match(self, *token_types: TokenType) -> bool:
        """Verifica se il token corrente è di uno dei tipi specificati"""
        if self.current_token:
            return self.current_token.type in token_types
        return False

    def parse(self) -> ProgramNode:
        """Punto di ingresso del parser"""
        statements = []

        while self.current_token and self.current_token.type !=
```

```
TokenType.EOF:
            stmt = self.statement()
            statements.append(stmt)

        return ProgramNode(statements)

    def statement(self) -> ASTNode:
        """statement → assignment | expression ';'"""
        # Lookahead per distinguere assignment da expression
        if (self.current_token and
            self.current_token.type == TokenType.IDENTIFIER and
            self.position + 1 < len(self.tokens) and
            self.tokens[self.position + 1].type == TokenType.ASSIGN):
            return self.assignment()
        else:
            expr = self.expression()
            self.expect(TokenType.SEMICOLON)
            return expr

    def assignment(self) -> AssignmentNode:
        """assignment → IDENTIFIER '=' expression ';'"""
        identifier_token = self.expect(TokenType.IDENTIFIER)
        self.expect(TokenType.ASSIGN)
        expr = self.expression()
        self.expect(TokenType.SEMICOLON)

        return AssignmentNode(identifier_token.value, expr)

    def expression(self) -> ASTNode:
        """expression → term (('+'|'-') term)*"""
        node = self.term()

        while self.match(TokenType.PLUS, TokenType.MINUS):
            operator = self.current_token
            self.advance()
            right = self.term()
            node = BinaryOpNode(node, operator, right)

        return node

    def term(self) -> ASTNode:
        """term → factor (('*'|'/') factor)*"""
        node = self.factor()

        while self.match(TokenType.MULTIPLY, TokenType.DIVIDE):
            operator = self.current_token
            self.advance()
            right = self.factor()
            node = BinaryOpNode(node, operator, right)
```

```
        return node

    def factor(self) -> ASTNode:
        """factor → NUMBER | IDENTIFIER | '(' expression ')'"""
        token = self.current_token

        if self.match(TokenType.NUMBER):
            self.advance()
            # Converti a float se contiene punto, altrimenti int
            value = float(token.value) if '.' in token.value else
int(token.value)
            return NumberNode(value)

        elif self.match(TokenType.IDENTIFIER):
            self.advance()
            return IdentifierNode(token.value)

        elif self.match(TokenType.LPAREN):
            self.advance()  # Consuma '('
            expr = self.expression()
            self.expect(TokenType.RPAREN)  # Consuma ')'
            return expr

        else:
            self.error("Atteso NUMBER, IDENTIFIER o '('")

# Test del parser
def test_parser():
    code = "x = 10 + 5 * 2; y = (x - 3) / 2; result = x + y;"

    lexer = Lexer(code)
    tokens = lexer.tokenize()

    parser = Parser(tokens)
    ast = parser.parse()

    print("=== AST GENERATO ===")
    for i, stmt in enumerate(ast.statements):
        print(f"Statement {i+1}: {stmt}")
```

2.3 Interprete/Evaluator

```
"""
Interprete che esegue l'AST usando Visitor Pattern
Mantiene simboli globali per variabili
"""

class Interpreter:
```

```python
    def __init__(self):
        self.variables: Dict[str, float] = {}

    def visit(self, node: ASTNode):
        """Dispatcher principale del visitor pattern"""
        method_name = f'visit_{type(node).__name__}'
        method = getattr(self, method_name, self.generic_visit)
        return method(node)

    def generic_visit(self, node: ASTNode):
        """Fallback per nodi non supportati"""
        raise RuntimeError(f'Nessun metodo visit_{type(node).__name__}')

    def visit_ProgramNode(self, node: ProgramNode):
        """Esegue tutte le statement del programma"""
        result = None
        for statement in node.statements:
            result = self.visit(statement)
        return result

    def visit_NumberNode(self, node: NumberNode) -> float:
        """Restituisce il valore numerico"""
        return node.value

    def visit_IdentifierNode(self, node: IdentifierNode) -> float:
        """Lookup variabile nella symbol table"""
        if node.name in self.variables:
            return self.variables[node.name]
        else:
            raise NameError(f"Variabile '{node.name}' non definita")

    def visit_BinaryOpNode(self, node: BinaryOpNode) -> float:
        """Evalua operazioni binarie"""
        left_val = self.visit(node.left)
        right_val = self.visit(node.right)

        op_type = node.operator.type

        if op_type == TokenType.PLUS:
            return left_val + right_val
        elif op_type == TokenType.MINUS:
            return left_val - right_val
        elif op_type == TokenType.MULTIPLY:
            return left_val * right_val
        elif op_type == TokenType.DIVIDE:
            if right_val == 0:
                raise ZeroDivisionError("Divisione per zero")
            return left_val / right_val
        else:
            raise RuntimeError(f"Operatore sconosciuto: {op_type}")
```

```python
    def visit_AssignmentNode(self, node: AssignmentNode) -> float:
        """Assegna valore a variabile"""
        value = self.visit(node.expression)
        self.variables[node.identifier] = value
        return value

    def get_variables(self) -> Dict[str, float]:
        """Restituisce copia delle variabili correnti"""
        return self.variables.copy()

# === SISTEMA COMPLETO ===

class CalculatorEngine:
    """Interfaccia unificata per il sistema completo"""

    def __init__(self):
        self.interpreter = Interpreter()

    def execute(self, code: str) -> Dict[str, float]:
        """Esegue codice e restituisce variabili risultanti"""
        try:
            # Pipeline completa
            lexer = Lexer(code)
            tokens = lexer.tokenize()

            parser = Parser(tokens)
            ast = parser.parse()

            self.interpreter.visit(ast)

            return self.interpreter.get_variables()

        except Exception as e:
            print(f"Errore: {e}")
            return {}

    def debug_tokens(self, code: str):
        """Mostra analisi lessicale per debug"""
        lexer = Lexer(code)
        tokens = lexer.tokenize()

        print("=== ANALISI LESSICALE ===")
        for token in tokens:
            if token.type != TokenType.EOF:
                print(f"{token.type.name:12} | '{token.value}' |
L{token.line}:C{token.column}")

    def debug_ast(self, code: str):
        """Mostra AST per debug"""
```

```
        lexer = Lexer(code)
        tokens = lexer.tokenize()
        parser = Parser(tokens)
        ast = parser.parse()

        print("=== ANALISI SINTATTICA ===")
        for i, stmt in enumerate(ast.statements):
            print(f"Statement {i+1}: {stmt}")

# Test completo
def test_complete_system():
    calc = CalculatorEngine()

    print("=== SISTEMA CALCOLATORE COMPLETO ===\n")

    # Test di espressioni complesse
    programs = [
        "x = 10;",
        "y = x + 5 * 2;",
        "z = (x + y) / 3;",
        "result = z * 2 - x;",
    ]

    for program in programs:
        print(f"Eseguo: {program}")
        variables = calc.execute(program)
        print(f"Variabili: {variables}\n")

    # Test debug
    print("=== DEBUG COMPLETO ===")
    test_code = "a = (10 + 5) * 2; b = a / 3;"

    print(f"Codice: {test_code}\n")
    calc.debug_tokens(test_code)
    print()
    calc.debug_ast(test_code)
    print()
    result = calc.execute(test_code)
    print(f"Risultato finale: {result}")

if __name__ == "__main__":
    test_complete_system()
```

---

3. Considerazioni Pratiche e Ottimizzazioni

3.1 Gestione Errori Avanzata

```
class ParseError(Exception):
    def __init__(self, message: str, token: Token):
        self.message = message
        self.token = token
        super().__init__(f"{message} alla riga {token.line}, colonna
{token.column}")

class ErrorRecoveryParser(Parser):
    """Parser con recupero errori per IDE/editor"""

    def __init__(self, tokens: List[Token]):
        super().__init__(tokens)
        self.errors: List[ParseError] = []

    def error(self, message: str):
        """Registra errore ma continua parsing"""
        error = ParseError(message, self.current_token)
        self.errors.append(error)

        # Strategia di recupero: salta al prossimo ';' o EOF
        while (self.current_token and
               self.current_token.type not in [TokenType.SEMICOLON,
TokenType.EOF]):
            self.advance()

        if self.current_token and self.current_token.type ==
TokenType.SEMICOLON:
            self.advance()
```

3.2 Ottimizzazioni Performance

```
class OptimizedLexer(Lexer):
    """Lexer ottimizzato per file grandi"""

    def __init__(self, text: str):
        super().__init__(text)
        # Pre-computa tutte le newline per lookup O(1)
        self.newline_positions = [
            i for i, char in enumerate(text) if char == '\n'
        ]

    def get_line_column(self, position: int) -> tuple[int, int]:
        """Calcolo efficiente di riga/colonna"""
        # Binary search nelle newline
        import bisect
        line = bisect.bisect_left(self.newline_positions, position) + 1

        if line == 1:
```

```
            column = position + 1
        else:
            column = position - self.newline_positions[line - 2]

        return line, column
```

3.3 Estensioni Avanzate

```
# Supporto per funzioni
class FunctionCallNode(ASTNode):
    def __init__(self, name: str, arguments: List[ASTNode]):
        self.name = name
        self.arguments = arguments

# Supporto per array/liste
class ArrayNode(ASTNode):
    def __init__(self, elements: List[ASTNode]):
        self.elements = elements

# Supporto per controllo di flusso
class IfNode(ASTNode):
    def __init__(self, condition: ASTNode, then_stmt: ASTNode, else_stmt:
Optional[ASTNode] = None):
        self.condition = condition
        self.then_stmt = then_stmt
        self.else_stmt = else_stmt
```

---

4. Confronto con Librerie Esistenti

4.1 PLY (Python Lex-Yacc)

Quando usare: Progetti professionali, grammatiche complesse, compatibilità yacc

```
# Esempio minimo PLY
import ply.lex as lex
import ply.yacc as yacc

# Lexer PLY
tokens = ('NUMBER', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN', 'RPAREN')
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'
```

```
t_ignore = ' \t'

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

# Parser PLY con precedenza
precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
)

def p_expression_binop(p):
    '''expression : expression PLUS expression
                  | expression MINUS expression
                  | expression TIMES expression
                  | expression DIVIDE expression'''
    if p[2] == '+': p[0] = p[1] + p[3]
    elif p[2] == '-': p[0] = p[1] - p[3]
    elif p[2] == '*': p[0] = p[1] * p[3]
    elif p[2] == '/': p[0] = p[1] / p[3]

lexer = lex.lex()
parser = yacc.yacc()
```

4.2 Lark

Quando usare: Prototipazione rapida, sintassi pulita, flessibilità

```
# Esempio Lark (richiede: pip install lark)
from lark import Lark, Transformer

grammar = '''
    start: expr
    expr: expr "+" term   -> add
        | expr "-" term   -> sub
        | term
    term: term "*" factor -> mul
        | term "/" factor -> div
        | factor
    factor: NUMBER        -> number
          | "(" expr ")"

    NUMBER: /\d+/
    %ignore /\s+/
'''

class Calculator(Transformer):
```

```
    def add(self, args): return args[0] + args[1]
    def sub(self, args): return args[0] - args[1]
    def mul(self, args): return args[0] * args[1]
    def div(self, args): return args[0] / args[1]
    def number(self, args): return int(args[0])

parser = Lark(grammar)
calc = Calculator()
# Uso: calc.transform(parser.parse("2 + 3 * 4"))
```

---

5. Criteri di Scelta

5.1 Matrice Decisionale

Criterio	Da Zero	PLY	Lark	ANTLR
Velocità sviluppo	⭐⭐	⭐⭐⭐	⭐⭐⭐⭐	⭐⭐⭐
Performance	⭐⭐⭐⭐	⭐⭐⭐	⭐⭐	⭐⭐⭐
Controllo	⭐⭐⭐⭐	⭐⭐	⭐⭐	⭐
Manutenibilità	⭐⭐	⭐⭐⭐	⭐⭐⭐⭐	⭐⭐⭐⭐
Curva apprendimento	⭐⭐⭐	⭐⭐	⭐⭐⭐	⭐

5.2 Raccomandazioni Pratiche

Implementazione da Zero → Progetti didattici, linguaggi molto semplici, requisiti performance estremi

PLY → Progetti professionali stabili, team con esperienza yacc/lex, grammatiche medie

Lark → Prototipazione, DSL aziendali, sviluppo agile, sintassi moderna

ANTLR → Linguaggi complessi, grammatiche esistenti, tooling avanzato

---

6. Pattern di Implementazione Scalabili

6.1 Architettura Modulare

```
from abc import ABC, abstractmethod

class LexerInterface(ABC):
```

```
    @abstractmethod
    def tokenize(self, text: str) -> List[Token]: pass

class ParserInterface(ABC):
    @abstractmethod
    def parse(self, tokens: List[Token]) -> ASTNode: pass

class InterpreterInterface(ABC):
    @abstractmethod
    def evaluate(self, ast: ASTNode): pass

class LanguageProcessor:
    """Facade per sistema linguistico completo"""

    def __init__(self, lexer: LexerInterface,
                 parser: ParserInterface,
                 interpreter: InterpreterInterface):
        self.lexer = lexer
        self.parser = parser
        self.interpreter = interpreter

    def process(self, code: str):
        tokens = self.lexer.tokenize(code)
        ast = self.parser.parse(tokens)
        return self.interpreter.evaluate(ast)
```

6.2 Plugin System per Estensioni

```
class LanguageExtension:
    """Base per estensioni linguistiche"""

    def extend_lexer(self, lexer: Lexer): pass
    def extend_parser(self, parser: Parser): pass
    def extend_interpreter(self, interpreter: Interpreter): pass

class MathExtension(LanguageExtension):
    """Aggiunge funzioni matematiche"""

    def extend_lexer(self, lexer: Lexer):
        lexer.add_patterns([
            (r'sin|cos|tan|sqrt', TokenType.MATH_FUNCTION),
        ])

    def extend_interpreter(self, interpreter: Interpreter):
        import math
        interpreter.functions.update({
            'sin': math.sin,
            'cos': math.cos,
```

```
            'sqrt': math.sqrt,
        })
```