I. Introduzione: Decifrare il Linguaggio del Computer

Ogni volta che un computer elabora un comando, un programma o un testo, due componenti fondamentali operano dietro le quinte: il lexer e il parser. Questi strumenti agiscono come "traduttori", consentendo alla macchina di trasformare una sequenza di caratteri (il testo che viene scritto) in una struttura organizzata e comprensibile.

Cosa sono Lexer e Parser e perché sono Fondamentali?

Il **lexer**, noto anche come analizzatore lessicale, scanner o tokenizzatore, rappresenta la fase iniziale di questo processo.1 Il suo compito è prendere il testo grezzo in input e scomporlo in unità significative, chiamate *token*. Si può immaginare questo processo come la lettura di una frase, dove ogni parola e segno di punteggiatura viene identificato come un'unità distinta. L'analisi lessicale costituisce la prima fase del *front end* di un compilatore.1

All'interno dell'analisi lessicale, due concetti chiave sono il *lessema* e il *token*. Il **lessema** è la sequenza di caratteri che compone il valore testuale grezzo di un'unità, come ad esempio "123". Il **token**, invece, è una rappresentazione più astratta, tipicamente una coppia (tipo, valore), dove il "tipo" indica la categoria lessicale (ad esempio, `NUMERO` o `OPERATORE`) e il "valore" è il lessema stesso (ad esempio, `(NUMERO, "123")`). Il lexer spesso conserva il lessema originale, poiché queste informazioni possono essere cruciali per le fasi successive, come l'analisi semantica.

Il **parser**, o analizzatore sintattico, subentra dopo il lexer. Riceve la sequenza di token e verifica la loro correttezza sintattica, costruendo una struttura gerarchica che rappresenta il significato del testo.3 Questo processo, noto anche come *parsing*, è analogo a verificare se le parole identificate dal lexer formano frasi grammaticalmente corrette.5

La chiara distinzione dei ruoli tra lexer e parser illustra un principio fondamentale dell'ingegneria del software: la separazione delle preoccupazioni. Il lexer si concentra esclusivamente sull'identificazione delle unità lessicali, senza preoccuparsi di come queste si combinano in strutture più ampie. Viceversa, il parser si occupa della validazione della struttura, senza dover gestire i dettagli a livello di singolo carattere. Questa modularità semplifica enormemente la gestione della complessità di sistemi ampi come i compilatori. Se un errore lessicale si verifica (ad esempio, un carattere non riconosciuto), è responsabilità del lexer; se un errore sintattico (come un operatore fuori posto) viene rilevato, è il parser a gestirlo. Questa divisione delle responsabilità non solo facilita il debugging e la manutenzione, ma permette anche un riutilizzo più efficiente dei componenti.

Il Ruolo nel Processo di Compilazione: Una Panoramica

Lexer e parser sono le fasi iniziali del "front end" di un compilatore, un software che trasforma il codice sorgente scritto in un linguaggio di alto livello in una rappresentazione

intermedia, che può poi essere ottimizzata e convertita in codice macchina eseguibile da un computer.1

Il processo di compilazione è una sequenza strutturata di fasi, dove l'output di ciascuna fase diventa l'input per la fase successiva.5 Questa architettura a pipeline è un modello di ingegneria del software estremamente efficace per affrontare problemi complessi, suddividendoli in sotto-problemi sequenziali e specializzati. Il lexer e il parser sono i primi esempi concreti di questo approccio modulare, che garantisce che ogni componente si concentri su un compito specifico, contribuendo alla robustezza e alla manutenibilità dell'intero sistema.

Le fasi principali del compilatore includono:

1. **Analisi Lessicale (Lexing):** Il codice sorgente viene letto carattere per carattere e convertito in una sequenza di token significativi.5
2. **Analisi Sintattica (Parsing):** La sequenza di token viene verificata rispetto alla grammatica del linguaggio, producendo un *albero di sintassi* (parse tree).5
3. **Analisi Semantica:** Si controlla il "significato" delle istruzioni, come la compatibilità dei tipi di dati o la corretta dichiarazione delle variabili, producendo un *albero sintattico astratto (AST)*.1
4. **Generazione del Codice Intermedio:** L'AST viene trasformato in un formato più semplice e standardizzato, facilitando le ottimizzazioni successive.6
5. **Ottimizzazione del Codice Intermedio:** Vengono applicate tecniche per migliorare l'efficienza e le prestazioni del codice.6
6. **Generazione del Codice Target:** Il codice intermedio ottimizzato viene convertito nel linguaggio macchina o in un altro linguaggio di destinazione.6

Questa progressione sequenziale, dove ogni output alimenta la fase successiva, è fondamentale per la comprensione del funzionamento di un compilatore. La seguente tabella riassume questo flusso di dati.

Tabella 1: Fasi del Processo di Compilazione (Input/Output)

Fase del Compilatore	Attività Principale	Input	Output	Componente Chiave
Analisi Lessicale	Scomposizione del codice sorgente in token significativi.	Codice Sorgente (stringa di caratteri)	Sequenza di Token	Lexer (Scanner/Tokenizzatore)
Analisi Sintattica	Verifica della grammatica e costruzione della struttura del programma.	Sequenza di Token	Albero di Sintassi (Parse Tree)	Parser (Analizzatore Sintattico)

Analisi Semantica	Controllo del significato e delle regole del linguaggio (es. tipi).	Albero di Sintassi	Albero Sintattico Astratto (AST)	Analizzatore Semantico
Generazione Codice Intermedio	Trasformazione dell'AST in una rappresentazione più semplice.	Albero Sintattico Astratto (AST)	Codice Intermedio	Generatore di Codice Intermedio
Ottimizzazione Codice Intermedio	Miglioramento dell'efficienza del codice intermedio.	Codice Intermedio	Codice Intermedio Ottimizzato	Ottimizzatore
Generazione Codice Target	Conversione del codice ottimizzato nel linguaggio macchina.	Codice Intermedio Ottimizzato	Codice Macchina (o altro target)	Generatore di Codice Target

II. Fase 1: L'Analisi Lessicale (Lexing)

Il Lexer (o Scanner/Tokenizzatore): Il Primo Passo

Il lexer è il "lettore" iniziale del codice sorgente. La sua funzione principale è scansionare il testo carattere per carattere, raggruppando sequenze di caratteri che formano unità significative e ignorando elementi non rilevanti per la struttura lessicale, come gli spazi bianchi o i commenti (a meno che non siano esplicitamente parte di un token).1 Questo componente è spesso chiamato anche *scanner* o *tokenizzatore*.1 L'output del lexer è una sequenza lineare di token, che serve come input diretto per la fase successiva, l'analisi sintattica.2

Per illustrare, consideriamo la stringa "123 + 141 / 725". Il lexer elaborerebbe questa sequenza e produrrebbe una serie di token, ignorando gli spazi, come `(NUMERO, "123")`, `(OPERATORE, "+")`, `(NUMERO, "141")`, `(OPERATORE, "/")`, `(NUMERO, "725")`. Ogni token incapsula sia la categoria lessicale (il "tipo") sia il valore testuale originale (il "lessema").

Lessemi e Token: Le Unità Costitutive del Linguaggio

Come accennato, il **lessema** è la porzione di testo esatta che il lexer identifica come un'unità. Ad esempio, nella parola chiave `if`, `if` è il lessema. Nel numero `42`, `42` è il lessema. Il **token**, invece, è la rappresentazione astratta di questo lessema, una coppia che include il suo *tipo* (o categoria lessicale) e il *valore* (il lessema originale). Esempi di tipi di token comuni includono `IDENTIFICATORE` (per nomi di variabili come `pippo`), `NUMERO`,

`OPERATORE` (`+`, `-`, `*`, `/`), `PAROLA_CHIAVE` (`for`, `while`), `STRINGA`, e `PUNTEGGIATURA` (`;`, `(`, `)`).6

Il lexer mantiene le informazioni sul lessema originale (il valore) perché queste possono essere utili per l'analisi semantica successiva, che potrebbe aver bisogno del valore esatto per controlli di tipo o altre operazioni. Questa trasformazione dal lessema al token è un passo cruciale di astrazione del linguaggio. Il parser, che opera sui token, non ha più bisogno di preoccuparsi dei dettagli a livello di carattere (ad esempio, se "123" è formato dalle cifre '1', '2', '3'), ma può operare a un livello più alto, trattando "123", "456" e "7" tutti semplicemente come `NUMERO`. Questa astrazione semplifica enormemente il lavoro del parser, permettendogli di concentrarsi sulla struttura logica del linguaggio piuttosto che sui suoi dettagli superficiali, rendendo il processo di parsing più scalabile e leggibile.

Come Funziona un Lexer: Dalla Stringa al Flusso di Token

I lexer sono tipicamente basati su regole, spesso definite tramite espressioni regolari (regex).3 Ogni regola associa un pattern di caratteri a un tipo di token. Il lexer scorre l'input, cercando la corrispondenza più lunga possibile per un token. Se più pattern corrispondono, viene generalmente data priorità a quello definito prima o con una priorità maggiore. Una volta che un lessema viene riconosciuto, il lexer crea un token corrispondente e lo aggiunge alla sequenza di output, spostandosi poi al carattere successivo non ancora processato.

Per esempio, un lexer potrebbe avere le seguenti regole concettuali:

- Regola per `NUMERO`: `\d+` (una o più cifre)
- Regola per `OPERATORE_ADD`: `\+` (il simbolo più)
- Regola per `SPAZIO`: `\s+` (uno o più spazi, spesso ignorati tramite una direttiva `%ignore` o una gestione esplicita)

L'utilizzo delle espressioni regolari nei lexer non è una scelta arbitraria, ma è profondamente radicata nella teoria dei linguaggi formali. Le espressioni regolari sono equivalenti agli automi a stati finiti, macchine computazionali che possono riconoscere pattern in una stringa in modo estremamente efficiente. Questa connessione teorica è ciò che garantisce l'efficienza e l'affidabilità del processo di tokenizzazione.

III. Fase 2: L'Analisi Sintattica (Parsing)

Il Parser: Dare un Senso ai Token

Il parser riceve la sequenza di token generata dal lexer. Il suo compito principale è costruire una struttura gerarchica che rappresenti le relazioni tra questi token, verificando al contempo che la sequenza rispetti le regole grammaticali del linguaggio.3 Questo processo è comunemente chiamato *parsing*.5 Le regole che il parser utilizza sono definite in una *grammatica formale*, spesso impiegando notazioni standard come la BNF (Backus-Naur Form) o la sua estensione, la EBNF (Extended Backus-Naur Form).3

Ad esempio, una regola grammaticale potrebbe definire che un'espressione (`espressione`) può essere un semplice `numero` o una combinazione di `espressione` , `operatore` e `numero` (es. `espressione -> numero | espressione operatore numero`). Il parser riconoscerebbe `(NUMERO, "123") (OPERATORE, "+") (NUMERO, "456")` come un'espressione valida, ma rileverebbe un errore sintattico in una sequenza come `(OPERATORE, "+") (NUMERO, "123") (NUMERO, "456")` .

La grammatica, in questo contesto, non è solo un insieme di regole, ma una specifica formale di ciò che è sintatticamente corretto in un dato linguaggio. Essa funge da "contratto" tra il programmatore (o l'utente che fornisce l'input) e il sistema che deve interpretare il codice. Se il codice rispetta questo contratto grammaticale, il parser può elaborarlo; in caso contrario, il parser segnala un errore sintattico. Questo concetto di "contratto" è fondamentale per la robustezza e la prevedibilità di qualsiasi sistema di elaborazione del linguaggio.

L'Albero di Sintassi (Parse Tree): La Struttura Grezza

Il risultato immediato dell'analisi sintattica è l'*albero di sintassi*, noto anche come parse tree.3 Questo albero illustra l'intera derivazione della stringa di input secondo le regole della grammatica. Contiene tutti i token prodotti dal lexer e tutte le regole intermedie applicate durante il processo di parsing, offrendo una rappresentazione estremamente dettagliata e fedele del modo in cui l'input è stato riconosciuto. La sua utilità risiede principalmente nel debugging del parser e della grammatica, poiché mostra esattamente come l'input è stato interpretato e quali regole sono state applicate.

Tuttavia, un parse tree è una rappresentazione molto "verbosa". Ad esempio, ogni parentesi, ogni parola chiave come `if` , `then` , `else` verrebbe esplicitamente rappresentata, anche se il loro significato è già implicito nella struttura. Questa verbosità, sebbene utile per il debugging del parser stesso, lo rende meno efficiente per le fasi successive del compilatore, come l'analisi semantica o la generazione del codice. Questo porta alla necessità di una rappresentazione più compatta e significativa, l'Albero Sintattico Astratto (AST), che filtra il "rumore" sintattico per concentrarsi sul significato intrinseco.

L'Albero Sintattico Astratto (AST): La Rappresentazione Essenziale

L'*albero sintattico astratto* (AST) è una rappresentazione più compatta e astratta del codice sorgente.1 A differenza del parse tree, l'AST elimina i dettagli sintattici irrilevanti (come le parentesi o le parole chiave che non portano un significato semantico diretto) e si concentra sulla struttura essenziale del programma e sulle relazioni logiche tra gli elementi. L'AST è l'output dell'analisi semantica 6 e serve come input per le fasi successive del compilatore, come la generazione del codice intermedio.6

Il vantaggio principale dell'AST è la sua semplicità e leggerezza nell'elaborazione per le fasi successive del compilatore. Rappresenta il "cosa" del programma (la sua logica) piuttosto

che il "come" è stato scritto sintatticamente (la sua forma superficiale). L'AST funge da linguaggio universale interno per il compilatore. Indipendentemente dalle specificità sintattiche del linguaggio sorgente (ad esempio, l'uso di parentesi graffe in C++ o l'indentazione in Python), l'AST fornisce una rappresentazione standardizzata e pulita della logica del programma. Questa standardizzazione facilita l'implementazione delle fasi di analisi semantica, ottimizzazione e generazione del codice, poiché queste possono operare su una struttura coerente e priva di ambiguità sintattiche.

IV. Lexer e Parser in Python: Strumenti e Applicazioni

Python, pur essendo un linguaggio interpretato, offre un ecosistema ricco per la creazione di lexer e parser. Questo è particolarmente utile per lo sviluppo di linguaggi specifici di dominio (DSL), per l'analisi di file di configurazione, o per l'elaborazione del linguaggio naturale (NLP).

Librerie Comuni per Lexer e Parser in Python

Esistono diverse librerie in Python per la creazione di lexer e parser, ognuna con le proprie caratteristiche e approcci:

- `shlex`: Questa libreria standard di Python è progettata per l'analisi lessicale semplice, in particolare per sintassi che ricordano quelle delle shell Unix. È utile per scrivere "mini-linguaggi" o per il parsing di stringhe quotate.12 `shlex` consente di configurare caratteri di punteggiatura, spazi bianchi e regole di escape per personalizzare la tokenizzazione.12
- `PLY` **(Python Lex-Yacc)**: PLY è un'implementazione in Python dei classici strumenti Unix Lex e Yacc. È una libreria stabile e ben mantenuta, che offre funzionalità di base per la costruzione di lexer e parser. In PLY, il lexer e il parser sono definiti separatamente tramite funzioni Python. Il lexer converte il testo in token usando espressioni regolari, mentre il parser utilizza regole di produzione definite tramite funzioni che iniziano con `p_` per costruire un AST.10 PLY è noto per il suo buon supporto alla diagnostica e alla gestione degli errori nella grammatica.
- `Lark`: Lark è un generatore di parser moderno e flessibile per Python, che si distingue per l'uso delle Parsing Expression Grammars (PEGs).3 A differenza di PLY, Lark include un lexer integrato, semplificando la definizione della grammatica e delle regole lessicali in un'unica stringa.10 Lark supporta diversi algoritmi di parsing, tra cui Earley (capace di gestire qualsiasi grammatica context-free, anche ambigua) e LALR(1) (molto efficiente per la maggior parte dei linguaggi di programmazione).3 Le grammatiche sono scritte in formato EBNF e possono generare automaticamente un AST. Lark è spesso considerato più performante di PLY.13
- **Altri Strumenti**: Esistono numerosi altri strumenti e librerie, tra cui generatori di parser come **ANTLR** (che genera parser per Python e altri linguaggi), **APG**, **Lrparsing**, **PlyPlus** (costruito su PLY), e **Pyleri**. Per i parser basati su PEG, si trovano librerie come **Arpeggio**, **Canopy**, **Parsimonious**, **pyPEG**, **TatSu**, e **Waxeye**. Inoltre, esistono parser

combinatori come **Parsec.py**, **Parsy**, **Pyparsing**, e **Funcparserlib**, più adatti per esigenze di parsing più semplici. Per l'analisi del codice Python stesso, l'interprete Python fornisce moduli come `tokenize` e `ast`.

Esempi di Codice: Calcolatrice Semplice con PLY e Lark

Per illustrare l'applicazione pratica di lexer e parser, si presentano esempi di una calcolatrice semplice utilizzando sia PLY che Lark. Questi esempi dimostrano come definire le regole lessicali e sintattiche per interpretare espressioni matematiche.

Esempio con PLY: Una Calcolatrice Base

PLY richiede la definizione esplicita dei token e delle regole del parser. L'esempio seguente implementa una calcolatrice che gestisce espressioni aritmetiche di base (addizione, sottrazione, moltiplicazione, divisione) e l'assegnazione di variabili.

Python

```
import ply.lex as lex
import ply.yacc as yacc

# Definizione dei token
tokens = (
    'NAME', 'NUMBER',
    'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'EQUALS',
    'LPAREN', 'RPAREN',
)

# Espressioni regolari per i token semplici
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_EQUALS  = r'='
t_LPAREN  = r'\('
t_RPAREN  = r'\)'
t_NAME    = r'[a-zA-Z_][a-zA-Z0-9_]*'

# Un token NUMBER con un'azione. PLY chiama questa funzione quando trova un
numero.
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

```
# Ignora spazi e tabulazioni
t_ignore = ' \t'

# Gestione degli errori lessicali
def t_error(t):
    print(f"Carattere illegale: '{t.value}'")
    t.lexer.skip(1)

# Costruzione del lexer
lexer = lex.lex()

# Dizionario per memorizzare le variabili
names = {}

# Regole di precedenza per gli operatori (dal più basso al più alto)
precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
)

# Regola di partenza
def p_statement_assign(p):
    'statement : NAME EQUALS expression'
    names[p] = p
    p = p # Restituisce il valore assegnato

def p_statement_expr(p):
    'statement : expression'
    p = p

def p_expression_binop(p):
    '''expression : expression PLUS expression
| expression MINUS expression
| expression TIMES expression
| expression DIVIDE expression'''
    if p == '+':
        p = p + p
    elif p == '-':
        p = p - p
    elif p == '*':
        p = p * p
    elif p == '/':
```

```
        if p == 0:
            print("Errore: Divisione per zero!")
            p = None
        else:
            p = p / p

def p_expression_paren(p):
    'expression : LPAREN expression RPAREN'
    p = p

def p_expression_number(p):
    'expression : NUMBER'
    p = p

def p_expression_name(p):
    'expression : NAME'
    try:
        p = names[p]
    except LookupError:
        print(f"Nome non definito: '{p}'")
        p = 0

# Gestione degli errori sintattici
def p_error(p):
    if p:
        print(f"Errore di sintassi al token '{p.value}' alla riga
{p.lineno}")
    else:
        print("Errore di sintassi alla fine dell'input")

# Costruzione del parser
parser = yacc.yacc()

# Loop interattivo (REPL)
def main_ply():
    while True:
        try:
            s = input('PLY Calc > ')
        except EOFError:
            break
        if not s:
            continue
        result = parser.parse(s)
```

```
        if result is not None:
            print(result)

if __name__ == '__main__':
    print("Calcolatrice PLY: Inserisci espressioni (es. 'x = 5 + 3', 'x *
2'). Premi Ctrl+D per uscire.")
    main_ply()
```

Questo esempio mostra come PLY utilizza funzioni Python per definire sia i token (`t_...`) che le regole di produzione della grammatica (`p_...`). Il lexer (`lex.lex()`) e il parser (`yacc.yacc()`) vengono costruiti separatamente, riflettendo l'architettura classica di Lex e Yacc.10 Le regole di precedenza e associatività degli operatori sono definite esplicitamente, e vengono incluse funzioni per la gestione degli errori lessicali e sintattici.

Esempio con Lark: Una Calcolatrice con Variabili

Lark adotta un approccio più integrato, dove la grammatica è definita in una singola stringa e un `Transformer` viene utilizzato per l'interpretazione dell'AST. L'esempio seguente, tratto dalla documentazione di Lark, implementa una calcolatrice simile con supporto per le variabili.14

Python

```
from lark import Lark, Transformer, v_args
import operator

# Definizione della grammatica della calcolatrice
calc_grammar = """
   ?start: sum                                  // La regola di partenza può
essere una somma
| NAME "=" sum -> assign_var           // O un'assegnazione di variabile

   ?sum: product
| sum "+" product -> add                 // Addizione
| sum "-" product -> sub                 // Sottrazione

   ?product: atom
| product "*" atom -> mul             // Moltiplicazione
| product "/" atom -> div             // Divisione (divisione reale)

   ?atom: NUMBER -> number                      // Un numero
| "-" atom -> neg                      // Negazione unaria
| NAME -> var                          // Un nome di variabile
| "(" sum ")"                          // Espressione tra parentesi
```

```
    %import common.CNAME -> NAME                 // Importa il terminale
CNAME (identificatore) e lo rinomina NAME
    %import common.NUMBER                        // Importa il terminale
NUMBER (numeri interi/decimali)
    %import common.WS_INLINE                     // Importa lo spazio bianco
in linea
    %ignore WS_INLINE                            // Ignora lo spazio bianco
in linea
"""

@v_args(inline=True)    # Questo decoratore modifica le firme dei metodi del
Transformer
class CalculateTree(Transformer):
    # Mappa gli operatori della grammatica alle funzioni Python
corrispondenti
    add = operator.add
    sub = operator.sub
    mul = operator.mul
    div = operator.truediv  # Usa truediv per la divisione floating-point
    neg = operator.neg

    number = float  # Converte i lessemi NUMBER direttamente in float

    def __init__(self):
        self.vars = {}  # Dizionario per memorizzare le variabili

    def assign_var(self, name, value):
        self.vars[name] = value
        return value

    def var(self, name):
        try:
            return self.vars[name]
        except KeyError:
            raise Exception(f"Variabile non trovata: {name}")

# Creazione dell'istanza del parser Lark
# 'lalr' è un algoritmo di parsing efficiente per la maggior parte dei
linguaggi di programmazione
# 'transformer' associa la classe CalculateTree al parser per l'evaluazione
dell'AST
calc_parser = Lark(calc_grammar, parser='lalr', transformer=CalculateTree())
```

```
calc = calc_parser.parse

# Loop interattivo (REPL)
def main_lark():
    print("Calcolatrice Lark: Inserisci espressioni (es. 'x = 10', '1 + x *
-3'). Premi Ctrl+D per uscire.")
    while True:
        try:
            s = input('Lark Calc > ')
        except EOFError:
            break
        try:
            print(calc(s))
        except Exception as e:
            print(f"Errore: {e}")

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

In questo esempio Lark, la grammatica è definita in una stringa multilinea.14 Lark gestisce implicitamente la tokenizzazione tramite le direttive `%import` e `%ignore`.14 La classe `CalculateTree` estende `Transformer` e definisce metodi che corrispondono alle regole della grammatica. Il decoratore `@v_args(inline=True)` semplifica il passaggio dei valori dei nodi figli ai metodi del transformer, rendendo l'elaborazione dell'AST più diretta.14

Applicazioni Reali di Lexer e Parser

Le applicazioni di lexer e parser vanno ben oltre la semplice compilazione di linguaggi di programmazione:

- **Compilatori e Interpreti**: Sono il cuore di qualsiasi linguaggio di programmazione, traducendo il codice sorgente in istruzioni eseguibili.1
- **Linguaggi Specifici di Dominio (DSL)**: Permettono di creare mini-linguaggi personalizzati per risolvere problemi specifici in modo più intuitivo e conciso, come linguaggi di configurazione, linguaggi di scripting per automazione o linguaggi di query.3
- **Analisi Statica del Codice (Linters, Formatter, Type Checkers)**: Strumenti come `Pylint`, `Black`, `MyPy`, `Pyright`, `Pytype` e `Flake8` utilizzano tecniche di parsing per analizzare la qualità del codice, rilevare errori di sintassi, incongruenze di tipo, vulnerabilità di sicurezza, e garantire la conformità agli standard di stile (come PEP 8).1 Questi strumenti migliorano la manutenibilità e la leggibilità del codice, integrandosi spesso nelle pipeline CI/CD.16
- **Elaborazione del Linguaggio Naturale (NLP)**: La tokenizzazione è una fase fondamentale in NLP, dove il testo viene suddiviso in parole o sottoparole. Librerie come NLTK e SpaCy offrono tokenizer per diverse lingue e scopi, inclusi tokenizer basati su

parole, caratteri o sottoparole (es. Byte-level BPE, WordPiece).17 Il parsing sintattico in NLP aiuta a comprendere la struttura grammaticale delle frasi.19

- **Motori di Ricerca e Information Retrieval**: L'analisi lessicale è impiegata per indicizzare testi, identificare parole chiave (keyword) e rimuovere parole comuni (stop word) per migliorare l'efficienza della ricerca.20
- **Parsing di Dati Non Strutturati**: I parser sono utilizzati per estrarre informazioni da documenti non strutturati, come file di log, documenti HTML o PDF. Un esempio notevole è il *CV Parsing*, che analizza curriculum vitae in vari formati (PDF, DOCX, JPEG) per estrarre e organizzare dati specifici (dati personali, competenze, esperienze lavorative) in un formato strutturato, automatizzando la compilazione di form di candidatura e la creazione di database di talenti.21
- **Strumenti di Manipolazione del Codice**: Possono essere usati per prettyprinter (formattatori di codice), refactoring automatico, o per generare documentazione.

V. Conclusioni

Lexer e parser sono componenti essenziali nell'informatica, fungendo da ponte tra il linguaggio umano e quello macchina. La loro architettura modulare, che separa l'analisi lessicale da quella sintattica, è un esempio paradigmatico di come problemi complessi possano essere efficacemente gestiti attraverso la scomposizione in fasi specializzate e sequenziali. Questa separazione non solo ottimizza il processo di sviluppo e debugging, ma promuove anche la riusabilità dei componenti.

La trasformazione del testo grezzo in token e successivamente in alberi sintattici (parse tree e AST) rappresenta un'astrazione progressiva del linguaggio. L'AST, in particolare, emerge come una rappresentazione pulita e semanticamente ricca, fondamentale per le fasi successive della compilazione e per una vasta gamma di applicazioni che richiedono la comprensione strutturale del testo.

Il panorama Python offre strumenti potenti e flessibili come PLY e Lark, che consentono agli sviluppatori di implementare lexer e parser con relativa facilità. Mentre PLY segue il modello classico Lex/Yacc con definizioni separate, Lark si distingue per il suo approccio integrato basato su PEG e un lexer contestuale, spesso offrendo maggiore efficienza e una gestione più agevole dell'ambiguità.

Le applicazioni di lexer e parser sono pervasive e critiche in numerosi campi. Dalla compilazione di linguaggi di programmazione e la creazione di DSL, all'analisi statica del codice che garantisce qualità e sicurezza, fino all'elaborazione del linguaggio naturale e all'estrazione di informazioni da dati non strutturati (come nel CV parsing), questi strumenti sono indispensabili per dare significato e struttura ai dati testuali. La loro comprensione e applicazione sono competenze fondamentali per chiunque operi nel campo dello sviluppo software e dell'analisi dei dati.