Esercizi esame FLT laboratorio

Parte I
Generale

1 Domanda 1

Consideriamo un linguaggio, L, **come si specificano**:

1. il lessico di L;
2. la sintassi di L.

1.1 Risposta

1. Il lessico di un linguaggio definisce l'insieme di simboli validi che possono essere utilizzati per formare le frasi nel linguaggio stesso. Nei linguaggi artificiali, gli elementi lessicali possono essere classificati nelle seguenti categorie:
   - **Parole chiave**: termini riservati con un significato specifico all'interno del linguaggio (es. if, for, class, ecc.).
   - **Delimitatori e operatori**: simboli che separano elementi del linguaggio o eseguono operazioni (es. ;, +, ++, , =).
   - **Classi lessicali aperte**: categorie di elementi lessicali che possono contenere un numero teoricamente illimitato di istanze. Questi elementi devono rispettare regole definite da un linguaggio regolare (automi a stati finiti). Tra questi rientrano:
     - **Identificatori**: nomi di variabili, funzioni, classi, metodi, ecc.
     - **Costanti**: numeri interi, numeri reali, stringhe alfanumeriche, ecc.
   - **Commenti**: parte del codice sorgente che viene ignorata dal compilatore o dall'interprete dopo l'analisi lessicale.
2. La sintassi di un linguaggio è definita mediante una grammatica context-free (CFG - Context-Free Grammar). La sintassi stabilisce come i token, riconosciuti dall'analisi lessicale, possono essere combinati per formare strutture valide nel linguaggio. Il parser analizza la sequenza di token generando un albero sintattico (AST - Abstract Syntax Tree) che rappresenta la struttura gerarchica della frase e le regole grammaticali seguite nella sua costruzione.

2 Domanda 2

Cosa si fa durante:

1. l'Analisi Lessicale;
2. l'Analisi Sintattica;
3. l'Analisi Semantica (controllo dei tipi).

2.1 Risposta

1. L'analisi lessicale è la fase in cui il codice sorgente viene suddiviso in token, unità sintattiche fondamentali (parole chiave, operatori, identificatori, ecc.). L'analizzatore lessicale (lexer o scanner) esegue le seguenti operazioni:
   - **Riconosce i token validi** all'interno del testo sorgente, confrontandoli con il lessico del linguaggio.

- **Filtra caratteri non significativi**, come spazi bianchi e commenti, ignorandoli nelle fasi successive.
- **Effettua la conversione dei token** in una rappresentazione più adatta per l'analisi successiva, spesso associando ad ogni token una categoria e, se necessario, un valore (es. il valore numerico di una costante).
- **Gestisce la tabella dei simboli**, registrando informazioni su identificatori, costanti e altre entità significative.

2. L'analisi sintattica (parsing) verifica che la sequenza di token generata dall'analizzatore lessicale segua le regole grammaticali del linguaggio. Le sue principali funzioni sono:
   - **Costruzione dell'AST** (Abstract Syntax Tree): il parser organizza i token in una struttura gerarchica che rappresenta la sintassi del programma.
   - **Rilevamento di errori sintattici**: il parser individua errori come parentesi mancanti, operatori usati in modo scorretto o costrutti non validi.
   - **Preparazione dell'analisi semantica**: raccoglie informazioni sui nomi e le loro dichiarazioni (spesso tramite una tabella dei simboli).
3. L'analisi semantica verifica che il programma abbia un significato coerente secondo le regole del linguaggio. Una delle operazioni più importanti è il controllo dei tipi (type checking), che garantisce che le operazioni siano effettuate su dati compatibili (es. non si può sommare un intero a una stringa in un linguaggio tipato staticamente). Le principali operazioni di questa fase includono:
   - **Verifica della coerenza dei tipi** nelle espressioni e nelle assegnazioni.
   - **Controllo della corretta dichiarazione e utilizzo di variabili e funzioni.**
   - **Gestione del binding** (associazione tra nomi e definizioni):
     - **Dichiarazioni**: introducono un'associazione tra un nome e una definizione (es. int x;).
     - **Istruzioni**: possono avere effetti collaterali e sono soggette a restrizioni di tipo (es. una funzione void non può restituire un valore).
     - **Espressioni**: computano un valore e hanno un tipo ben definito.

     Nei linguaggi semplici, il controllo dei tipi può avvenire durante il parsing, mentre in linguaggi più complessi si effettua successivamente con una visita dell'AST.

3 Domanda 3

Quali sono gli input e gli output delle 3 analisi precedenti?

3.1 Risposta

1. **Analisi Lessicale (Lexical Analysis)**:
   - **Input**: codice sorgente come sequenza di caratteri.
   - **Output**: sequenza di token (unità lessicali come parole chiave, identificatori, operatori, costanti, ecc.), spesso accompagnata da informazioni aggiuntive (es. posizione nel codice, valore per costanti numeriche, riferimento alla tabella dei simboli).
2. **Analisi Sintattica (Parsing)**:
   - **Input**: sequenza di token prodotta dall'analizzatore lessicale.
   - **Output**: albero sintattico (AST - Abstract Syntax Tree) che rappresenta la struttura gerarchica del programma secondo le regole grammaticali del linguaggio. Se ci sono errori sintattici, vengono segnalati.
3. **Analisi Semantica**:

- **Input**: AST generato dal parser e la tabella dei simboli (contenente informazioni su variabili, funzioni, tipi, ecc.).
- **Output**:
  - AST arricchito con informazioni semantiche (es. tipi delle espressioni).
  - Eventuali errori semantici, come operazioni tra tipi incompatibili o utilizzo di variabili non dichiarate.
  - Tabella dei simboli aggiornata con informazioni aggiuntive (es. tipi delle variabili e delle funzioni).

Parte II
Scanner

4 Domanda 4

Quali classi lessicali fanno in generale parte di un linguaggio di programmazione?

4.1 Risposta

In un linguaggio di programmazione, i token sono suddivisi in diverse classi lessicali. Le principali classi lessicali che si trovano in quasi tutti i linguaggi di programmazione sono:

- **Parole chiave** Parole riservate con un significato specifico nel linguaggio, che non possono essere utilizzate come identificatori. **Esempi:** `if, else, while, for, return, int, float, class, def`
- **Delimitatori** Simboli utilizzati per separare elementi nel codice, come punteggiatura e simboli di raggruppamento. **Esempi:** `;, ,, (), {}, []`
- **Operatori** Simboli che eseguono operazioni aritmetiche, logiche o di assegnazione. **Esempi:**
  - Aritmetici: `+, -, *, /, %`
  - Logici: `&&, ||, !`
  - Relazionali: `==, !=, >, <, >=, <=`
  - Assegnazione: `=, +=, -=, *=, /=`
- **Classi lessicali aperte**
  - **Identificatori** Nomi definiti dall'utente per variabili, funzioni, classi, oggetti, ecc. **Esempi:** `x, nomeVariabile, funzione1, ClassePersonale`
  - **Costanti** Valori fissi che compaiono direttamente nel codice. **Esempi:**
    - Numeri interi e reali: `42, 3.14, -7`
    - Stringhe: `"ciao", 'hello'`
    - Booleani: `true, false`
- **Commenti** Testo ignorato dal compilatore/interprete, usato per documentare il codice. **Esempi:**
  - In C/C++/Java: `// commento` o `/* commento multilinea */`
  - In Python: `# commento`

5 Domanda 5

Cosa è un Token?

5.1 Risposta

Un token è l'unità lessicale fondamentale in un linguaggio di programmazione, ovvero una sequenza di caratteri che ha un significato specifico. Ogni token appartiene a una categoria lessicale (es. identificatori, operatori, parole chiave, numeri, ecc.).

L'analizzatore lessicale (lexer) suddivide il codice sorgente in token e li passa all'analizzatore sintattico (parser) per la successiva elaborazione.

Le espressioni regolari sono spesso usate per definire i pattern dei token.

6 Domanda 6

Fare esempi di Token.

6.1 Risposta

Degli esempi di token possono essere:

- **Parole chiave (Keyword)**: int, float, void, if, for, return
- **Delimitatori**: ';', '{', '}', '(', ')'
- **Numeri (Costanti numeriche)**: 0, 42, 3.14, -10
- **Operatori**: +, -, *, /, ==, !=, &&, ||, =
- **Identificatori (nomi di variabili, funzioni, classi, ecc.)**: x, y, numero, main

7 Domanda 7

Come (con quale classi di linguaggi) si specificano i Token dei linguaggi?

7.1 Risposta

I token dei linguaggi di programmazione vengono specificati utilizzando linguaggi regolari, che appartengono alla classe dei linguaggi regolari nella gerarchia di Chomsky. Alcuni esempi di definizione di token mediante espressioni regolari:

- **Identificatori**: [a-zA-Z_][a-zA-Z0-9_]* (iniziano con una lettera o _, seguiti da lettere, numeri o _)
- **Numeri interi**: [0-9]+
- **Numeri reali**: [0-9]+.[0-9]+
- **Parole chiave**: if | else | while | return | int | float (insieme di stringhe fisse)
- **Operatori**: \+ | \- | * | \/ | == | != | ¡= | ¿=

8 Domanda 8

Come si può implementare il riconoscimento lessicale?

8.1 Risposta

L'implementazione per il riconoscimento lessicale si può fare in tre modi diversi:

1. **Realizzazione procedurale**: un programma ad-hoc che riconosce tutti gli elementi lessicali producendo i corrispondenti token. Può essere implementata in diversi modi:
   (a) A partire dalle espressioni regolari, che descrivono le classi di token.
   (b) A partire dall'automa a stati finiti (DFA o NFA), che riconosce sequenze valide di caratteri e genera i token corrispondenti.
   (c) A partire dalla grammatica regolare, che può essere trasformata in un automa a stati finiti.

   La conversione da un'espressione regolare al codice per l'analisi si basa sulle seguenti corrispondenze:

   - Ogni **concatenazione** viene tradotta in una sequenza di operazioni.
   - Ogni **unione** viene gestita con una selezione (`if` o `switch`).

- Ogni **stella di Kleene** è tradotta con un ciclo.

2. **Realizzazione tabulare interpretata**: in questo caso, il DFA riconoscitore della grammatica G è rappresentato tramite una struttura dati (una tabella di transizioni). Un programma indipendente dalla grammatica G esegue l'automa leggendo questa tabella.

   Il processo di costruzione dell'automa da un'espressione regolare avviene nel seguente modo:

   - Si costruisce una tabella T tale che, dato uno stato s e un carattere c, se $T(s, c) = s'$ allora lo stato successivo è s'.
   - Si definisce una funzione che esegue l'automa, che **non dipende** dalla specifica espressione regolare.

   L'automa deve essere **deterministico** per garantire efficienza nell'analisi. Questo approccio è utilizzato dai generatori di analizzatori lessicali, come Flex e Lex.

3. **Generazione automatica con uno Scanner Generator**: strumenti come JFlex, Lex o Flex permettono di generare automaticamente un analizzatore lessicale. Essi prendono in input le espressioni regolari corrispondenti ai token e producono codice che implementa l'analizzatore.

Parte III
Parsing

9 Domanda 9

Definizione di grammatica $LL(1)$.

9.1 Risposta

Una grammatica è detta $LL(1)$ se può essere analizzata da un parser **top-down** con lookahead di un solo simbolo, senza necessità di backtracking.

- Il primo L indica che la lettura dell'input avviene da sinistra (**Left-to-right**).
- Il secondo L indica che viene costruita la derivazione più a sinistra (**Leftmost derivation**).
- Il numero 1 indica che il parser utilizza un solo simbolo di lookahead per determinare quale produzione applicare.

Formalmente, una grammatica è $LL(1)$ se per ogni simbolo non-terminale A con produzioni:

$$A \to p_1 \mid p_2 \mid \ \dots \ \mid p_n$$

gli insiemi **Predict** associati a ciascuna produzione devono essere **mutuamente disgiunti**, ovvero:

$$Pred_i \cap Pred_j = \emptyset \quad \text{per ogni } 1 \leq i \neq j \leq n.$$

L'insieme **Predict** di una produzione $A \to \alpha$ si calcola come segue:

- Se α inizia con un terminale a, allora $Predict(A \to \alpha) = \{a\}$.
- Se α inizia con un non-terminale B, allora $Predict(A \to \alpha) = First(B)$.
- Se α può derivare ϵ (stringa vuota), allora bisogna includere anche $Follow(A)$:

$$Predict(A \to \alpha) = First(\alpha) \cup Follow(A) \quad \text{se } \epsilon \in First(\alpha).$$

Un linguaggio si dice $LL(1)$ se esiste almeno una grammatica $LL(1)$ che lo genera.

10 Domanda 10

Per quali ragioni una grammatica può non essere $LL(1)$?

10.1 Risposta

Una grammatica può non essere $LL(1)$ per le seguenti ragioni:

1. **Forte Ambiguità:** Se la grammatica è ambigua, esistono più derivazioni sinistre per la stessa stringa, rendendo impossibile determinare un'unica produzione sulla base di un solo token di lookahead.
2. **Fattorizzazione comune a sinistra:** Se due o più produzioni di un non-terminale iniziano con lo stesso prefisso, il parser non può decidere quale produzione scegliere solo in base al primo token. **Esempio:**

   $$A \to \alpha\beta_1 \mid \alpha\beta_2$$

   In questo caso, il parser non può distinguere quale produzione usare solo guardando α.

3. **Ricorsione sinistra:** Se un non-terminale può derivare una stringa che inizia con sé stesso, il parser $LL(1)$ entra in un loop infinito. **Esempio:**

$$A \to A\alpha \mid \beta$$

Qui, A può generare una stringa che inizia con A, causando un ciclo nella derivazione.

4. **First e Follow non disgiunti:** Una grammatica non è $LL(1)$ se per un non-terminale A esiste una produzione $A \to \epsilon$ (produzione che può derivare la stringa vuota), e l'intersezione tra i simboli in $First(A)$ e in $Follow(A)$ non è vuota. In questo caso, il parser non può distinguere se deve espandere la produzione o meno. **Esempio:**

$$A \to \epsilon \mid aB$$

Se il simbolo a è sia in $First(A)$ che in $Follow(A)$, il parser avrà ambiguità nella scelta della produzione.

11 Domanda 11

Come si può specificare un parser Top-Down?

11.1 Risposta

Il **parsing top-down** è una tecnica di analisi sintattica che costruisce l'albero di derivazione partendo dal simbolo iniziale della grammatica e procedendo verso le foglie. Si basa sulla ricostruzione della derivazione **left-most** (sinistra), scegliendo le produzioni sulla base del **lookahead**, cioè il simbolo successivo dell'input.

Il parsing top-down può essere implementato in due modi principali:

1. **Parsing a Discesa Ricorsiva**: Un insieme di funzioni mutuamente ricorsive viene scritto manualmente per analizzare l'input, basandosi sulla grammatica. Ogni funzione corrisponde a un simbolo non-terminale e sceglie la produzione da applicare in base al simbolo di lookahead. Questo metodo è semplice da implementare per grammatiche $LL(1)$, ma non può gestire grammatiche con **ricorsione sinistra**, che devono essere riscritte in forma equivalente.

2. **Parsing LL basato su Tabelle**: Invece di utilizzare ricorsione diretta, il parsing LL con tabella utilizza una struttura dati (una **tabella di parsing**) e uno stack esplicito per simulare l'espansione della derivazione. Il parser consulta la tabella per determinare la produzione da applicare sulla base del lookahead.

Il parsing LL richiede la costruzione della **tabella Predict**, che indica per ogni simbolo non terminale e simbolo di input quale produzione deve essere applicata. Questa tabella si basa sul calcolo degli insiemi **FIRST** e **FOLLOW**.

Parte IV
AST e Symbol Table

12 Domanda 12

Cosa è e a cosa serve definire un Abstract Syntax Tree per una grammatica di un linguaggio di programmazione?

12.1 Risposta

L'**Abstract Syntax Tree** (AST) è una rappresentazione strutturata e semplificata della sintassi di un programma. A differenza dell'albero di parsing completo, l'AST omette dettagli irrilevanti per l'analisi semantica, come simboli di separazione (es. parentesi, virgole) e regole sintattiche ridondanti.

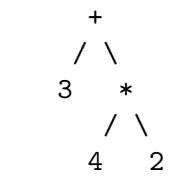
L'AST viene costruito dopo l'analisi sintattica e ha diversi scopi:

- **Analisi Semantica**: verifica della correttezza del programma, come il controllo dei tipi e la risoluzione dei nomi.
- **Ottimizzazione del Codice**: trasformazioni per migliorare l'efficienza prima della generazione del codice.
- **Generazione del Codice**: l'AST viene trasformato in codice macchina, bytecode o rappresentazioni intermedie per l'esecuzione.

Esempio: Consideriamo l'espressione:

```
3 + 4 * 2
```

L'AST corrispondente è:



A differenza dell'albero sintattico completo, l'AST rispetta la precedenza degli operatori senza includere dettagli superflui come parentesi.

13 Domanda 13

Quale è la differenza fra AST e parsing tree per una stringa di un linguaggio di programmazione? Fare un esempio di un parse tree e di un AST per una istruzione **if-else** con la sintassi Java.

13.1 Risposta

Il **parse tree** (o *albero di derivazione*) rappresenta l'intera struttura sintattica di un'espressione, includendo tutti i simboli della grammatica formale, sia terminali che non terminali. Contiene dettagli inutili per l'analisi semantica, come regole sintattiche intermedie.

L'**AST** (Abstract Syntax Tree) è una versione semplificata del parse tree, in cui:

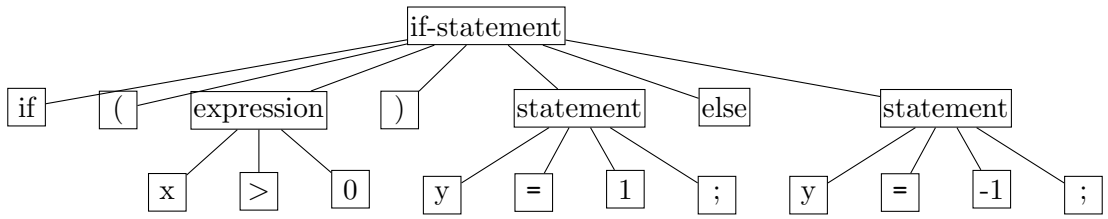
- Vengono rimossi i simboli non terminali che non sono necessari.
- La struttura dell'albero segue la logica del programma piuttosto che le regole sintattiche.
- Ogni nodo rappresenta un'operazione o un costrutto significativo del linguaggio.

Esempio: Consideriamo la seguente istruzione Java:

```
if (x > 0)
    y = 1;
else
    y = -1;
```

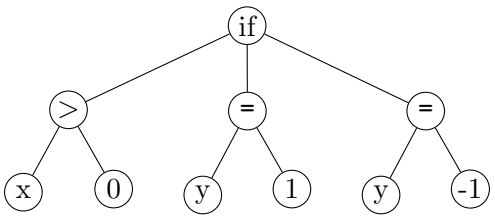
13.1.1 Parse Tree

Il **parse tree** rappresenta tutte le regole sintattiche coinvolte:



13.1.2 Abstract Syntax Tree (AST)

L'**AST** rimuove i dettagli sintattici e conserva solo la struttura logica:



In questo caso, l'AST ha eliminato parentesi, parole chiave e simboli ridondanti, mantenendo solo la struttura essenziale per l'analisi semantica e la generazione di codice.

14 Domanda 14

Una Symbol Table (tabella dei simboli) è una struttura dati utilizzata durante la compilazione per memorizzare e gestire le informazioni relative agli identificatori presenti nel programma sorgente. Essa è condivisa tra le diverse fasi della compilazione e consente di recuperare informazioni essenziali sugli identificatori quando vengono usati nel codice.

Informazioni contenute nella Symbol Table:
Ogni identificatore memorizzato nella tabella è associato a diversi attributi, tra cui:

- Nome dell'identificatore (variabili, funzioni, classi, ecc.).
- Tipo di dato (es. int, float, string).
- Ambito (Scope) in cui è definito (locale, globale, ecc.).
- Posizione di memoria o offset (se necessario per l'allocazione).
- Modificatori di accesso (pubblico, privato, protetto nelle classi).
- Parametri (per funzioni), se l'identificatore è una funzione.
- Eventuali valori di default.

Utilizzo della Symbol Table:
Durante la compilazione, la tabella dei simboli viene utilizzata per:

1. Verificare che ogni identificatore sia stato dichiarato prima dell'uso.
2. Associare ogni riferimento a una variabile o funzione con la sua definizione e i suoi attributi.

3. Gestire la risoluzione dei nomi nelle diverse fasi della compilazione.
4. Supportare l'analisi semantica, ad esempio nel controllo dei tipi e della corretta associazione dei parametri nelle chiamate a funzione.
5. Ottimizzare il codice, fornendo informazioni utili per la generazione del codice intermedio e finale.

15 Domanda 15

Come può essere implementata una Symbol Table?

15.1 Risposta

Una Symbol Table è una struttura dati che funge da dizionario, mappando nomi (identificatori di variabili, funzioni, classi, ecc.) a informazioni associate (tipo, ambito, posizione in memoria, ecc.). Poiché l'operazione più frequente è la ricerca, è fondamentale scegliere una struttura dati efficiente.

Le implementazioni più comuni sono:

1. **Lista/Array di record**
   - Ogni elemento dell'array contiene il nome dell'identificatore e le informazioni associate.
   - La ricerca avviene scansionando sequenzialmente la lista.
   - Complessità della ricerca: $O(n)$ nel caso peggiore.
2. **Array ordinato + Ricerca binaria**
   - Gli elementi vengono mantenuti ordinati, consentendo la ricerca binaria.
   - Complessità della ricerca: $O(\log n)$, ma gli inserimenti sono costosi ($O(n)$ nel caso peggiore per mantenere l'ordine).
3. **Albero Binario di Ricerca (BST - Binary Search Tree)**
   - Struttura gerarchica che permette ricerche efficienti.
   - Complessità della ricerca: $O(\log n)$ per alberi bilanciati, ma può degradare a $O(n)$ in caso di alberi sbilanciati.
4. **Tabella Hash (Hash Table)**
   - Gli identificatori vengono mappati in una tabella usando una funzione di hash.
   - Complessità della ricerca: $O(1)$ nel caso medio, ma $O(n)$ nel caso peggiore (se si verificano troppe collisioni).

Parte V
Analisi di tipo e visitor

16 Domanda 16

Cosa si fa con l'Analisi di Tipo? Come si specifica?

16.1 Risposta

L'Analisi di Tipo (Type Checking) verifica che il programma rispetti le regole di tipizzazione imposte dal linguaggio di programmazione. In particolare, controlla:

- **Compatibilità tra tipi nelle operazioni** (es. sommare un intero e una stringa non è consentito in molti linguaggi).
- **Assegnamenti validi tra variabili** (es. assegnare un valore di tipo int a una variabile float può essere permesso o meno a seconda del linguaggio).
- **Tipi di ritorno delle funzioni** coerenti con quanto dichiarato.
- **Corretta risoluzione dei tipi nei parametri delle funzioni**.

Se il controllo fallisce, il compilatore/interprete segnala un errore di tipo.

L'Analisi di Tipo può essere effettuata:

- **Staticamente** (a tempo di compilazione) in linguaggi come C, Java, Rust.
- **Dinamicamente** (a tempo di esecuzione) in linguaggi come Python, JavaScript.

Nei linguaggi semplici, l'analisi di tipo può avvenire durante il parsing, calcolando il tipo delle componenti e utilizzando la Symbol Table come struttura globale. Per i linguaggi più complessi, l'analisi di tipo viene eseguita visitando (anche più volte) l'AST del programma.

Abbiamo due principali approcci per implementare l'Analisi di Tipo:

1. **Metodo Diretto**: si aggiunge il metodo ***abstract*** *TypeDescriptor calcResType()* alla classe base *NodeAST* e lo si implementa in ogni sottoclasse concreta dell'AST. Questo metodo viene chiamato durante la visita dell'AST per calcolare il tipo delle espressioni e verificare eventuali errori di tipo.
2. **Pattern Visitor**: invece di inserire direttamente il controllo nei nodi dell'AST, si definisce una classe separata (un Visitor) che attraversa l'AST e si occupa dell'analisi di tipo. Questo approccio migliora la modularità e separa la logica di analisi dalla struttura dell'AST.

17 Domanda 17

A cosa serve il Pattern Visitor? Descriverne la struttura.

17.1 Risposta

Il pattern di design **Visitor** permette di definire nuove operazioni sugli elementi di una gerarchia di classi senza modificarne la struttura. Questo è utile quando si vuole **separare il comportamento dagli oggetti stessi**, evitando di modificare direttamente le loro classi.

17.1.1 Problemi risolti dal Visitor

- Separare la logica di elaborazione dagli oggetti visitati, facilitando l'aggiunta di nuove operazioni senza modificare le classi esistenti.
- Evitare una proliferazione di metodi nelle classi della gerarchia.
- Usare il **doppio dispatch**, che consente di selezionare l'operazione corretta in base al tipo concreto dell'oggetto visitato.

17.1.2 Struttura del Pattern Visitor

Il pattern Visitor è composto da:

- **Interfaccia Visitor**: definisce un metodo 'visit(Element e)' per ogni tipo concreto della gerarchia di classi.
- **Element (nodi della gerarchia)**: definisce il metodo 'accept(Visitor v)', che richiama il metodo appropriato del Visitor.
- **ConcreteVisitor**: implementa le operazioni da eseguire sugli elementi della gerarchia.

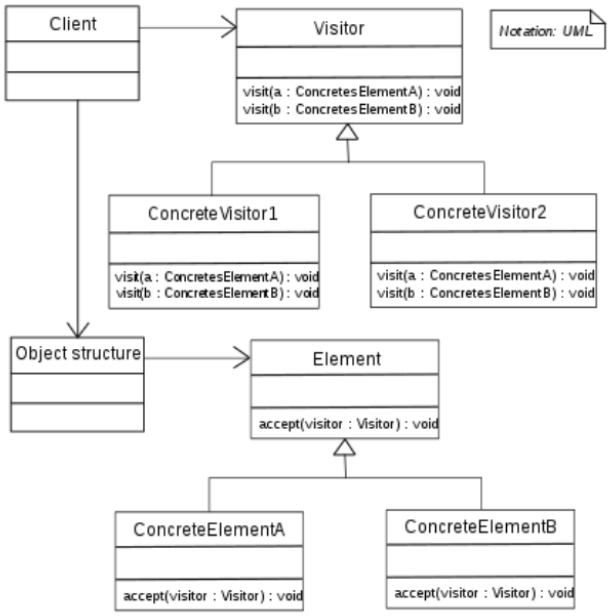


Figura 1: Diagramma UML del Pattern Visitor

17.1.3 Esempio di utilizzo (pseudocodice in Java)

```
interface Visitor {
    void visit(NumberNode n);
    void visit(OperatorNode o);
}

class ASTNode {
    abstract void accept(Visitor v);
}

class NumberNode extends ASTNode {
    int value;
    void accept(Visitor v) { v.visit(this); }
}

class OperatorNode extends ASTNode {
    String operator;
    ASTNode left, right;
    void accept(Visitor v) { v.visit(this); }
}

class PrintVisitor implements Visitor {
    void visit(NumberNode n) { System.out.print(n.value); }
```

```
    void visit(OperatorNode o) {
        System.out.print("(");
        o.left.accept(this);
        System.out.print(" " + o.operator + " ");
        o.right.accept(this);
        System.out.print(")");
    }
}
```

In questo esempio, **PrintVisitor** visita i nodi di un AST e stampa l'espressione corrispondente.

17.1.4 Vantaggi e Svantaggi

Vantaggi:

- Facilita l'aggiunta di nuove operazioni senza modificare la struttura dell'AST.
- Migliora la separazione tra struttura dati e logica di elaborazione.

Svantaggi:

- Difficile da applicare se la gerarchia di classi cambia frequentemente.
- Aggiungere un nuovo tipo di nodo richiede la modifica di tutti i Visitor esistenti.

Parte VI
Esercizi

18 Domanda 18

Consideriamo il seguente linguaggio di espressioni intere e floating point:
$inum$ $[0-9]+$
$fnum$ $[0-9]+.[0-9]+$

$Expr \rightarrow Expr\ \ plus\ \ Expr \mid Expr\ \ times\ \ Expr \mid Val$
$Val \rightarrow inum \mid fnum$

Assumiamo che le operazioni *plus* e *times* possano avvenire solo se i due operandi sono interi oppure floating point. Scrivere:

1. una espressione corretta;
2. una con errori lessicali;
3. una senza errori lessicali ma con errori sintattici;
4. ed una con solo errori semantici.

18.1 Risposta

1. **Espressione corretta** (nessun errore lessicale, sintattico o semantico):

   $$3\ plus\ 5\ times\ 2$$

   - Sintatticamente corretta: segue la grammatica fornita.
   - Semanticamente corretta: le operazioni avvengono tra numeri dello stesso tipo.

2. **Espressione con errori lessicali** (simboli non conformi alla grammatica):

   $$3\ pluss\ 5\ times\ 2.0$$

   - Errore lessicale: "pluss" non è un token riconosciuto.
   - L'errore viene individuato nella fase di analisi lessicale.

3. **Espressione senza errori lessicali ma con errori sintattici** (non rispetta la struttura della grammatica):

   $$plus\ 3\ times\ 5$$

   - Errore sintattico: la produzione $Expr \rightarrow plus\ \ Expr\ \ Expr$ non esiste nella grammatica fornita.
   - La fase di parsing non può costruire un albero sintattico valido.

4. **Espressione con solo errori semantici** (rispetta la grammatica ma viola le regole di tipizzazione):

   $$3\ plus\ 2.5$$

   - Errore semantico: *plus* viene applicato a un intero e a un float, ma la specifica richiede che entrambi gli operandi abbiano lo stesso tipo.
   - L'analisi semantica rileva l'errore poiché il controlo dei tipi fallisce.

19 Domanda 19

Data la grammatica:

1. $S \to A\, C\, \$$
2. $C \to c$
3. $C \to \varepsilon$
4. $A \to A\, B\, C\, d\, c$
5. $A \to BQ$
6. $B \to b\, B$
7. $B \to \varepsilon$
8. $Q \to q$
9. $Q \to \varepsilon$

1. Fare la tabella **Predict** per la grammatica;
2. Dire se la grammatica è $LL(1)$;
3. Fare la derivazione sinistra della stringa $b\, c\, d\, c\, c\, d\, c\, \$$ (se la stringa appartiene al linguaggio).

19.1 Risposta

1. Mi accorgo che per A è presente una ricorsione sinistra perché $A \to A...$, quindi per prima cosa devo rimuoverla.

   $A \to BQA'$
   $A' \to B\, C\, d\, c\, A' \mid \varepsilon$

LHS	RHS	DER ε	FIRST	FOLLOW	PREDICT
S	AC$	SI	{b,c,d, $}	———	{b,c,d, $}
C	c	NO	{c}	{d, $}	{c}
C	ε	SI	$\emptyset$	{d, $}	{d, $}
A	BQA’	SI	{b,c,d}	{c, $}	{b,c,d, $}
A’	BCdcA’	NO	{b,c,d}	{a}	{b,c,d}
A’	ε	SI	$\emptyset$	{a}	{a}
B	bB	NO	{b}	{q, c}	{b}
B	ε	SI	$\emptyset$	{q, c}	{q,c}
Q	q	NO	{q}	{b,c,d}	{q}
Q	ε	SI	$\emptyset$	{b,c,d}	{b,c,d}

2. Per controllare se la grammatica è $LL(1)$ devo calcolare le intersezioni tra i predict della stessa produzione, se sono tutti disgiunti posso affermare che la grammatica è $LL(1)$.
   $C \Rightarrow \{c\} \cap \{d, \$\} = \emptyset$
   $A' \Rightarrow \{b, c, d\} \cap \{a\} = \emptyset$
   $B \Rightarrow \{b\} \cap \{q, c\} = \emptyset$
   $Q \Rightarrow \{q\} \cap \{b, c, d\} = \emptyset$

Dopo aver calcolato tutte le intersezioni e visto che sono tutte disgiunte posso affermare che la grammatica è $LL(1)$.

3. La derivazione sinistra della stringa $b\,c\,d\,c\,c\,d\,c\,\$$ è:

$$S \xrightarrow{1} AC\$ \xrightarrow{4} BQA'C\$ \xrightarrow{7} bBQA'C\$ \xrightarrow{8} bQA'C\$ \xrightarrow{10} bA'C\$ \xrightarrow{5} bBCdcA'C\$ \xrightarrow{8} bCdcA'C\$$$
$$\xrightarrow{2} bcdcA'C\$ \xrightarrow{5} bcdcBCdcA'C\$ \xrightarrow{8} bcdcCdcA'C\$ \xrightarrow{2} bcdccdcA'C\$ \xrightarrow{6} bcdccdcC\$ \xrightarrow{3} bcdccdc\$$$

20 Domanda 20

Dire se le seguenti grammatiche sono $LL(1)$ oppure no:

1. $S \to A\,B\,c\,\$$
2. $A \to a$
3. $A \to \varepsilon$
4. $B \to b$
5. $B \to \varepsilon$

1. $S \to A\,b\,\$$
2. $A \to a$
3. $A \to B$
4. $A \to \varepsilon$
5. $B \to b$
6. $B \to \varepsilon$

20.1 Risposta

LHS	RHS	DER ε	FIRST	FOLLOW	PREDICT
S	ABc$	SI	{a,b,c,$}	——	{a,b,c,$}
A	a	NO	{a}	{b,c,$}	{a}
A	ε	SI	$\emptyset$	{b,c,$}	{b,c,$}
B	b	NO	{b}	{c}	{b}
B	ε	SI	$\emptyset$	{c}	{c}

$A \Rightarrow \{a\} \cap \{b, c, \$\} = \emptyset$
$B \Rightarrow \{b\} \cap \{c\} = \emptyset$
Essendo entrambi degli insiemi disgiunti la grammatica è $LL(1)$.

LHS	RHS	DER ε	FIRST	FOLLOW	PREDICT
S	Ab$	SI	{a,b,$}	——	{a,b,$}
A	a	NO	{a}	{b}	{a}
A	B	SI	{b}	{b}	{b}
A	ε	SI	$\emptyset$	{b}	{b}
B	b	NO	{b}	{b}	{b}
B	ε	SI	$\emptyset$	{b}	{b}

$A \Rightarrow \{a\} \cap \{b\} \cap \{b\} = \emptyset$
$B \Rightarrow \{b\} \cap \{b\} = \{b\}$
Essendo che nel secondo caso non abbiamo degli insiemi disgiunti la grammatica NON è $LL(1)$.

21 Domanda 21

Data la grammatica:

1. $S \rightarrow E\,\$$
2. $E \rightarrow T\,R$
3. $R \rightarrow +\,E$
4. $R \rightarrow \varepsilon$
5. $T \rightarrow F\,D$
6. $D \rightarrow \varepsilon$
7. $D \rightarrow *\,T$
8. $F \rightarrow n$
9. $F \rightarrow (\,E\,)$
10. $F \rightarrow \varepsilon$

1. Fare la tabella **Predict** per la grammatica;
2. Dire se la grammatica è $LL(1)$;
3. Fare la derivazione sinistra della stringa ( ) $ (se la stringa appartiene al linguaggio).

21.1 Risposta

1. Calcolo la tabella dei predict.

LHS	RHS	DER ε	FIRST	FOLLOW	PREDICT
S	E$	SI	{n,(,*,+,$}	——	{n,(,*,+,$}
E	TR	SI	{n,(,*,+}	{$,)}	{n,(,*,+, $,)}
R	+E	NO	{+}	{$,)}	{+}
R	ε	SI	$\emptyset$	{$,)}	{$,)}
T	FD	SI	{n,(, *}	{+, $,)}	{n, (, *, +, $,)}
D	ε	SI	$\emptyset$	{+,$,)}	{+, $,)}
D	*T	NO	{*}	{+,$,)}	{*}
F	n	NO	{n}	{*, +, $,)}	{n}
F	(E)	NO	{(}	{*, +, $,)}	{(}
F	ε	SI	$\emptyset$	{*, +, $,)}	{*, +, $,)}

2. è $LL(1)$?
   $R \Rightarrow \{+\} \cap \{\$,)\} = \emptyset$
   $D \Rightarrow \{+, \$,)\} \cap \{*\} = \emptyset$
   $F \Rightarrow \{n\} \cap \{(\} \cap \{*, +, \$,)\} = \emptyset$

   Essendo tutti disgiunti posso affermare che la grammatica è $LL(1)$.

3. La derivazione sinistra della stringa ( ) $ è:

   $S \overset{1}{\rightarrow} E\$ \overset{2}{\rightarrow} TR\$ \overset{5}{\rightarrow} FDR\$ \overset{9}{\rightarrow} (E)DR\$ \overset{2}{\rightarrow} (TR)DR\$ \overset{5}{\rightarrow} (FDR)DR\$ \overset{10}{\rightarrow} (DR)DR\$ \overset{6}{\rightarrow} (R)DR\$ \overset{4}{\rightarrow} ()DR\$ \overset{6}{\rightarrow} ()R\$ \overset{4}{\rightarrow} ()\$$

22 Domanda 22

Dire come mai la seguente grammatica non è $LL(1)$ e se possibile definire una equivalente $LL(1)$

1. $DL \rightarrow DL\,;\,D$
2. $DL \rightarrow D$
3. $D \rightarrow T\ IdL$
4. $IdL \rightarrow IdL\,,\,id$
5. $IdL \rightarrow id$
6. $T \rightarrow int$
7. $T \rightarrow bool$

Il punto e virgola e la virgola a destra nelle produzioni e *id*, *int* e *bool* sono simboli terminali.

22.1 Risposta

La grammatica non può essere $LL(1)$ perché è presente la ricorsione sinistra nelle produzioni 1 e 4, per definire una equivalente $LL(1)$ devo iniziare rimuovendo la ricorsione sinistra.

1. $S \to DL\$$
2. $DL \to D\ DL'$
3. $DL' \to ;\ D\ DL'$
4. $DL' \to \varepsilon$
5. $D \to T\ IdL$
6. $IdL \to id\ IdL'$
7. $IdL' \to ,\ id\ IdL'$
8. $IdL' \to \varepsilon$
9. $T \to int$
10. $T \to bool$

LHS	**RHS**	**DER ε**	**FIRST**	**FOLLOW**	**PREDICT**
S	DL $	NO	{int, bool}	—	{int, bool}
DL	D DL'	NO	{int, bool}	{$}	{int, bool}
DL'	; D DL'	NO	{;}	{$}	{;}
DL'	ε	SI	$\emptyset$	{$}	{$}
D	T IdL	NO	{int, bool}	{; , $}	{int, bool}
IdL	id IdL'	NO	{id}	{; , $}	{id}
IdL'	, id IdL'	NO	{,}	{; , $}	{,}
IdL'	ε	SI	$\emptyset$	{; , $}	{; , $}
T	int	NO	{int}	{id}	{int}
T	bool	NO	{bool}	{id}	{bool}

$DL' \Rightarrow \{;\} \cap \{\$\} = \emptyset$
$IdL' \Rightarrow \{,\} \cap \{;, \$\} = \emptyset$
$T \Rightarrow \{int\} \cap \{bool\} = \emptyset$
Essendo tutti degli insiemi disgiunti posso affermare che la grammatica è $LL(1)$.

23 Domanda 23

Consideriamo la grammatica:
$Expr \to Expr\ \ plus\ \ Expr \mid Expr\ \ times\ \ Expr \mid Val$
$Val \to inum \mid fnum$

Definire una grammatica per lo stesso linguaggio che permetta il parsing top-down e definire i **Predict** delle produzioni.

23.1 Risposta

Per rendere la grammatica adatta al parsing top-down, dobbiamo rimuovere la ricorsione sinistra, che impedisce il riconoscimento $LL(1)$.

Possiamo riscrivere la grammatica introducendo un non-terminale ausiliario $Expr'$ per gestire le operazioni binarie con associatività sinistra:
$S \to Expr\$$
$Expr \to Val\ \ Expr'$

$Expr' \rightarrow plus\ \ Val\ \ Expr' \mid times\ \ Val\ \ Expr' \mid \varepsilon$
$Val \rightarrow inum \mid fnum$
Ora la grammatica è adatta al parsing top-down perché non ha ricorsione sinistra diretta.

LHS	**RHS**	**DER** ε	**FIRST**	**FOLLOW**	**PREDICT**
S	Expr $	NO	{inum, fnum}	$\emptyset$	{inum, fnum}
Expr	Val Expr'	NO	{inum, fnum}	{$}	{inum, fnum}
Expr'	plus Val Expr'	NO	{plus}	{$}	{plus}
Expr'	times Val Expr'	NO	{times}	{$}	{times}
Expr'	ε	SI	$\emptyset$	{$}	{$}
Val	inum	NO	{inum}	{plus, times}	{inum}
Val	fnum	NO	{fnum}	{plus, times}	{fnum}