ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Utilizzo di Neverlang per la modellazione di Domain Specific Languages

Tesi di laurea in:
Programmazione ad Oggetti

Relatore
Prof. Viroli Mirko

Candidato
Terenzi Mirco

Correlatore
Prof. Aguzzi Gianluca

Anno Accademico 2023-2024

Abstract

Nel campo informatico, la riusabilità del codice è un aspetto fondamentale, associato a numerosi vantaggi come la riduzione dei tempi di sviluppo e una maggiore affidabilità, poiché è sufficiente che il codice sia testato una sola volta. Neverlang è un framework progettato per lo sviluppo di linguaggi di programmazione che adotta questa filosofia come elemento chiave, favorendo una modellazione *feature-oriented* volta a promuovere il riutilizzo del codice e, in particolare, dei componenti linguistici. Lo scopo di Neverlang è scomporre gli aspetti funzionali di un linguaggio in una serie di funzionalità, chiamate *features*, per poi creare Domain-Specific Languages (DSL) che implementano solamente le caratteristiche necessarie. Inoltre, l'utilizzo di DSL permette di diminuire l'*abstraction gap*, ossia la "distanza" tra lo spazio concettuale del problema e il linguaggio utilizzato per codificarlo, ed il "carico cognitivo" necessario per imparare tale linguaggio.

L'obiettivo di questo progetto di tesi è ricreare le funzionalità di SQL in modo modulare e indipendente, rendendo possibile la costruzione di un DSL selezionando soltanto le caratteristiche richieste dal contesto. In questo documento viene illustrato l'intero processo di progettazione e sviluppo del progetto, spiegando e motivando le scelte progettuali adottate. Infine, vengono descritte le possibili estensioni future del progetto, evidenziando i potenziali scenari di utilizzo del linguaggio sviluppato.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Il presente lavoro di tesi si concentra sull'utilizzo di Neverlang, un framework per la creazione di linguaggi di programmazione modulari, applicato alla modellazione di Domain-Specific Languages (DSL). I DSL sono linguaggi specializzati, progettati per rispondere a specifiche esigenze di un dominio applicativo, offrendo un approccio più naturale rispetto ai linguaggi di programmazione *general-purpose*. Attraverso Neverlang è possibile costruire DSL componendo moduli che rappresentano le varie caratteristiche del linguaggio, favorendo la riusabilità e l'estendibilità dei componenti.

Il progetto sviluppato in questa tesi mira a ricreare le funzionalità di SQL utilizzando Neverlang, con un focus particolare sulla modularità. Questo approccio è necessario per rispondere alle esigenze didattiche del progetto: l'obiettivo è creare un insieme di *features* che implementino singole caratteristiche di SQL, in modo da poter costruire una famiglia di linguaggi SQL che soddisfi le necessità poste a lezione, mostrando solo una sottoparte del linguaggio complessivo. L'elaborato affronta l'intero processo di sviluppo del DSL, dalla progettazione alla validazione.

Struttura della Tesi Di seguito viene brevemente descritta la struttura dei capitoli che compongono la tesi. Nel Capitolo 2 è illustrato il contesto in cui è stato sviluppato il progetto, introducendo i principali strumenti utilizzati, come Neverlang e Java, e le informazioni necessarie per comprendere il lavoro svolto. Nel Capitolo 3 sono elencati gli obiettivi del progetto insieme ad un'analisi dei requisiti funzionali e non funzionali. Qui viene spiegato come deve funzionare

il linguaggio, quali operazioni deve supportare e come deve essere strutturato. Nel Capitolo 4 è descritta l'architettura del progetto, illustrando come è stato modellato il linguaggio e come le varie componenti interagiscono tra loro. Viene approfondita l'implementazione delle funzionalità richieste, descrivendo come sono state realizzate ed integrate all'interno del DSL, e fornendo dettagli sulle principali tecnologie utilizzate durante lo sviluppo. Infine, nel Capitolo 5 viene esaminata la valutazione del progetto, analizzando i risultati ottenuti e le tecniche utilizzate per garantire la qualità del codice prodotto. Nella parte conclusiva, si discute il lavoro svolto e si propongono possibili sviluppi futuri del progetto.

Capitolo 2

Contesto

2.1 Domain-Specific Languages

In modo del tutto opposto rispetto ai linguaggi general-purpose, progettati per poter essere utilizzati in ogni contesto con un'efficienza e un grado d'espressività relativamente uguali, i Domain-Specific Language (DSL) sono ottimizzati per uno specifico ambito e risultano essere, in molti casi, una soluzione molto più naturale rispetto a quella fornita dai primi [10]. Tra gli esempi più comuni di DSL troviamo SQL, LaTeX (utilizzato anche per la scrittura di questo documento) e CSS.

Nonostante la definizione di DSL sia chiara, non è altrettanto immediato definire se un linguaggio sia o meno un DSL. In questo caso, vi sono alcuni principi chiave da osservare [8]:

- Un DSL è un linguaggio di programmazione e, come tale, la sua struttura dovrebbe essere progettata in modo da essere facile da comprendere per gli esseri umani e, al tempo stesso, eseguibile da un calcolatore.

- Essendo un linguaggio, deve avere un senso di fluidità e la sua espressività deve essere derivata non solo da un'espressione individuale, ma anche dalla combinazione di più istruzioni.

- Coerentemente alla definizione, un DSL dovrebbe implementare l'insieme minimo di caratteristiche necessarie per poter supportare il dominio applicativo di interesse ed evitare funzioni non fondamentali che potrebbero rendere

il linguaggio più difficile, sia da utilizzare sia da comprendere. L'obiettivo è quello di ridurre al minimo la complessità e, di conseguenza, la curva d'apprendimento del linguaggio.

L'utilizzo di questa tipologia di linguaggi di programmazione comporta una serie di vantaggi:

- **Produttività**: Essendo il livello d'astrazione maggiore, di conseguenza, lo è anche la produttività. Questo perché è possibile utilizzare direttamente i concetti propri del dominio applicativo, non essendo limitati dalla necessità di mantenere una generalità atta ad ottenere un linguaggio applicabile in molteplici contesti [12]. In particolare, viene ridotto l'*abstraction gap* e cioè la "distanza" tra lo spazio concettuale del problema e il linguaggio di programmazione utilizzato per codificare tale problema[1]. Come indicato da Paul Hudak [10] ed illustrato nella Figura 2.1, assumendo che il costo di sviluppo di un programma sia lineare, possiamo ipotizzare che il costo iniziale richiesto sia maggiore nel caso si utilizzasse un DSL rispetto a metodologie più convenzionali (considerando il caso in cui il linguaggio andasse sviluppato, se ne venisse utilizzato uno esistente tale costo sarebbe significativamente minore). Ciò nonostante, la pendenza della curva è considerevolmente più bassa e quindi, da un determinato punto in poi, l'utilizzo di DSL porterebbe un risparmio significativo.

- **Qualità del codice**: L'utilizzo di DSL favorisce anche una migliore qualità del codice. Infatti, il linguaggio può includere regole direttamente trasposte dal dominio all'interno del quale è applicato. In questo modo risulta molto più difficile, talvolta impossibile, ottenere dei risultati non attesi. Ad esempio, Antti Raunio, capo ingegnere del progetto EADS [15], afferma che "la qualità del codice generato è chiaramente migliore [...] perché il linguaggio di modellazione è stato progettato per adattarsi all'architettura del

[1] In linguaggi di basso livello tale differenza è notevole. Ad esempio, nel linguaggio C, concetti astratti come quello di "magazzino" devono essere modellati tramite gli strumenti messi a disposizione (potrebbe essere modellato tramite un array di elementi "prodotto"). Con l'utilizzo di DSL tale distanza viene ridotta in modo rilevante permettendo di "scrivere negli stessi termini" con i quali si pensa al probelama da risolvere.

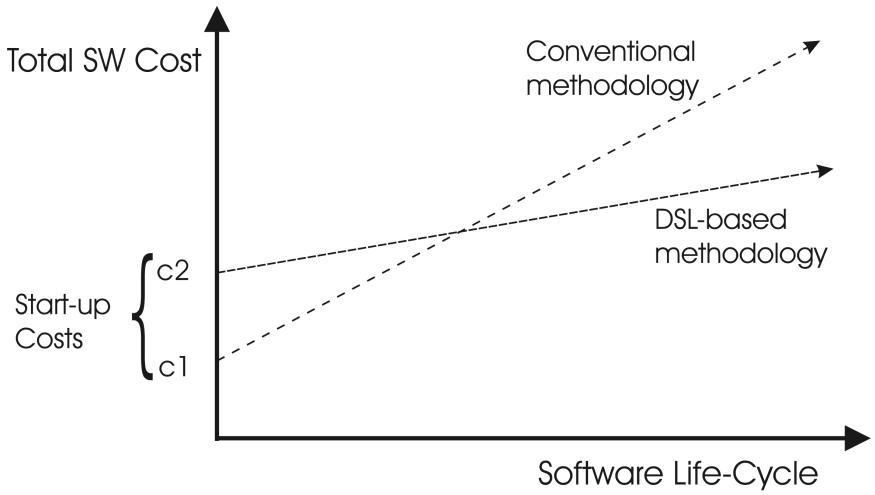

Figura 2.1: Grafico rappresentante il costo di sviluppo di un software in funzione del tempo, confrontando l'utilizzo di un DSL con quello di un linguaggio general-purpose.

nostro terminale"[2]. Inoltre, l'offuscamento della reale complessità del problema, dovuto all'utilizzo di DSL, consente ai nuovi sviluppatori di lavorare ad un alto livello d'astrazione, senza dover conoscere tutti i dettagli inerenti all'implementazione del linguaggio [15].

- **Migliore manutenibilità**: Sebbene l'uso di DSL non renda l'implementazione necessariamente meno complessa di quanto si possa ottenere utilizzando un linguaggio *general-purpose*, la manutenibilità del codice risulta essere accentuata [13]. Infatti, considerando il volume del codice, l'utilizzo di DSL comporta una minor quantità di codice da comprendere, facilitandone la modifica. Inoltre, è possibile ignorare il problema di mantenere coerente ciò che è definito dalla grammatica con la struttura gerarchica definita dall'Abstract Syntax Tree (AST) in quanto quest'ultimo si evolve con la prima [3].

[2]Di seguito riportata l'affermazione citata, in lingua originale:"the quality of the generated code is clearly better, simply because the modelling language was designed to fit our terminal architecture"

2.2 Grammatica non contestuale

Nel campo dell'informatica, una grammatica formale è costituita da regole che descrivono precisamente come vengono generati i simboli di un linguaggio formale, partendo da un insieme finito chiamato alfabeto. Ogni regola sintattica, detta anche produzione, è espressa nella forma $A \to x$, dove A è un simbolo non terminale e x è una combinazione di simboli terminali e/o non terminali.

In particolare, la grammatica non contestuale, o Context-Free Grammar (CFG), è un tipo specifico di grammatica formale che deriva il proprio nome dal fatto che le regole sintattiche mantengono la loro validità indipendentemente dai simboli che precedono o seguono il simbolo non terminale a cui si applicano. Questa caratteristica è dovuta al fatto che le regole sintattiche di una grammatica non contestuale ammettono soltanto una variabile non terminale sul lato sinistro della regola [14].

Una grammatica libera dal contesto viene definita da una quadrupla $G = (N, T, P, S)$, dove N è l'insieme dei simboli non terminali, T è l'insieme dei simboli terminali, P è l'insieme delle regole di produzione e S è il simbolo iniziale. Per illustrare meglio il funzionamento di una CFG, viene proposto l'esempio di $G = (\{S\}, \{0, 1\}, P, S)$, dove P è definito come segue:

$$S \to \epsilon \mid S0 \mid S1$$

Le tre regole definite all'interno della grammatica G proposta consentono a S di generare una stringa vuota o di concatenare un simbolo binario (0 o 1) ad una stringa già generata. In questo modo, partendo dal simbolo iniziale S e applicando le regole definite, è possibile generare tutte le stringhe binarie.

In conclusione, le grammatiche non contestuali assumono un ruolo fondamentale nella teoria dei linguaggi formali, in particolare nella definizione e nell'analisi dei linguaggi di programmazione. La CFG è utilizzata principalmente per la modellazione della sintassi, ma viene impiegata anche nella costruzione di interpreti e compilatori [14].

2.3 Compilazione del codice

Il compilatore è un software fondamentale nel campo della programmazione e dell'informatica, il cui compito è tradurre il codice sorgente (scritto in un linguaggio di alto livello) in un linguaggio di basso livello (tipicamente il linguaggio macchina o codice oggetto) che può essere eseguito dal calcolatore.

Il processo di compilazione è suddiviso in diverse fasi, ognuna delle quali ha un ruolo specifico nel convertire il codice sorgente in un programma eseguibile. Le fasi principali sono di seguito descritte [1]:

Analisi lessicale Durante la fase dell'analisi lessicale, talvolta definita anche *scanning*, la sequenza di caratteri di input viene analizzata e divisa in porzioni significative, chiamate lessemi. Per ogni lessema viene prodotto come output un token, definito come coppia di valori `<nome-token, valore-attributo>`, che viene processato dalla fase successiva.

Analisi sintattica La fase dell'analisi sintattica, più comunemente indicata come *parsing* (allo stesso modo, il programma che svolge queste operazioni è chiamato *parser*) o in italiano parsificazione, prevede che i token siano utilizzati per creare una rappresentazione ad albero che descriva efficacemente la struttura grammaticale dell'input. Un esempio di questo tipo di rappresentazione è costituito dall'albero sintattico. Al suo interno, ogni nodo rappresenta un'operazione e i nodi ad esso discendenti, chiamati anche figli, rappresentano gli operandi dell'operazione definita dal primo. La parsificazione è svolta principalmente secondo due metodologie [9]:

- l'analisi **top-down** consiste nell'emulare il processo di produzione della frase. In questo modo, partendo dal simbolo iniziale si procede per fasi ad evolvere l'output in modo da farlo corrispondere alla frase ottenuta come input. Il nome indica che l'albero sintattico viene costruito partendo dall'alto, ossia dal nodo radice, proseguendo verso il basso;
- l'analisi **bottom-up**, al contrario, prevede di invertire il processo di produzione della frase, avendo come obiettivo quello di ridurre la frase di input al simbolo iniziale.

Analisi semantica In questo stadio della compilazione, il fine è analizzare la correttezza semantica del programma. Più nello specifico, vengono utilizzati l'albero sintattico ottenuto dalla fase precedente e la tabella dei simboli per controllare che l'input sia semanticamente coerente con quanto è definito dal linguaggio. Una parte fondamentale dell'analisi semantica è quella del controllo del tipo (in inglese *type-checking*), durante la quale il compilatore si accerta che il valore assegnato ad una variabile sia ammissibile con il tipo di tale variabile e, allo stesso modo, che gli operandi utilizzati in un'operazione siano compatibili con l'operazione stessa. Talora, le specifiche del linguaggio permettono delle conversioni di tipo chiamate coercizioni.

Generazione di codice intermedio Durante la compilazione, il compilatore genera una o più rappresentazioni intermedie. L'albero sintattico precedentemente descritto rappresenta una delle varie forme che tali rappresentazioni possono assumere. In particolare, devono essere rispettate due proprietà significative: le rappresentazioni devono essere facili da generare e facili da trasporre in codice target (ossia il linguaggio di basso livello obiettivo della traduzione).

Ottimizzazione e generazione del codice Successivamente, si procede a perfezionare, per quanto possibile, il codice intermedio in modo che si possa poi ottenere un codice finale migliore. Infine, il codice intermedio viene utilizzato per generare il codice target.

2.4 Programmazione feature-oriented

La programmazione orientata alle funzionalità, comunemente nota come Feature-Oriented Programming (FOP), è un paradigma di programmazione focalizzato sulla modularizzazione del software. L'obiettivo principale della FOP è facilitare la gestione e lo sviluppo di programmi complessi, consentendo ai programmatori di scomporli in unità modulari. Questa metodologia è ampiamente applicata in contesti di linee produttive di software e viene utilizzata per gestire la creazione di diverse versioni di un programma, adatte alle specifiche richieste dei clienti [2]. Inoltre, la FOP è impiegata per costruire sistemi configurabili, che prevedono

l'attivazione o la disattivazione di funzionalità specifiche e offrono un elevato grado di personalizzazione.

Le caratteristiche principali della FOP sono le seguenti:

- **Riusabilità**: Confrontando la FOP con l'approccio più classico della Object-Oriented Programming (OOP) si nota come la prima offra una maggiore modularità e flessibilità del codice[3]. Di conseguenza, la riusabilità è amplificata poiché per ogni funzionalità implementata il nucleo funzionale e la parte relativa alle interazioni sono mantenute separate [16].

- **Gestione della complessità**: La FOP permette di gestire progetti di considerevole complessità grazie allo sviluppo modulare del codice in unità distinte. Queste unità possono essere sviluppate e mantenute in modo indipendente, riducendo significativamente l'interdipendenza e la complessità del sistema complessivo. Inoltre, la separazione delle funzionalità semplifica l'evoluzione del software, poiché ogni modifica può essere limitata a una singola funzionalità senza influenzare l'intero sistema [16].

- **Adattabilità**: La FOP consente una facile configurazione e personalizzazione dei sistemi software attraverso la selezione e la composizione delle funzionalità. Questo approccio permette di creare diverse varianti di un prodotto software a seconda dei requisiti specifici, facilitando l'adattamento a nuovi contesti o a esigenze mutate nel tempo [2].

2.5 Neverlang

Neverlang Language Workbench è un framework sviluppato presso l'Università di Milano dal professor Cazzola e dai suoi collaboratori, il cui scopo è favorire lo sviluppo di linguaggi di programmazione, in particolare seguendo il paradigma della FOP.

[3]Come discusso da Christian Prehofer, una delle differenze principali risiede nella composizione di più varianti di un componente software: la OOP prevede una gerarchia di classi, ognuna delle quali estende un'altra classe, mentre la FOP prevede l'implementazione di singole *features* per poi integrarle dove richiesto [16, pp. 466-468].

È basato sull'idea che i linguaggi di programmazione abbiano un'intrinseca divisione modulare in più caratteristiche, o *features*, ciascuna delle quali è implementata da un componente specifico. In accordo con tale visione, l'obiettivo del framework è definire i linguaggi tramite una divisione in frammenti, chiamati *modules*, ognuno dei quali si occupa di implementare una specifica caratteristica e, infine, tramite la combinazione dei diversi moduli, ottenere un linguaggio di programmazione specifico per il contesto applicativo richiesto, ossia un DSL [4].

Quando il codice viene compilato, ogni costrutto viene inserito in una rappresentazione chiamata AST, o albero sintattico astratto. L'AST è simile all'albero sintattico descritto in precedenza ma con la fondamentale differenza che non rappresenta ogni dettaglio della reale sintassi. È essenzialmente una versione alternativa e semplificata che consente di focalizzarsi sulla logica descritta dal linguaggio.

All'interno di ogni modulo vengono definite due parti principali:

- la **sintassi**, utilizzando una grammatica formale non contestuale;
- la **semantica**, in funzione della sintassi e sfruttando i vari elementi non terminali e i loro attributi. Inoltre, il comportamento del componente può essere suddiviso in diverse fasi, ciascuna definita da un ruolo specifico all'interno del modulo. Anche l'ordine d'esecuzione dei ruoli può essere specificato e le tre tipologie di visite predefinite sono:
  - Semi-automatica: Tale strategia prevede che i nodi siano visitati partendo dal nodo radice e discendendo ai nodi figli finché non viene individuata una definizione semantica. Una volta trovata, il controllo della visita viene trasferito all'esecuzione di tale azione.
  - *Post-order*: In questo caso la visita dell'albero predilige la discesa in profondità, ciò significa che l'esecuzione delle azioni definite dalla semantica dei vari nodi è posticipata a dopo che tutti i nodi figli sono stati valutati.
  - Giustapposizione: Nei due metodi precedenti, ad ogni ruolo corrisponde una visita all'albero. Al contrario, quando due ruoli sono giustapposti, la loro esecuzione viene eseguita "in una volta" e cioè tutte le azioni

```
module if_syntax {
    role(sytax) {
        Statement ← 'if' '(' Expression ')' '{' Statement '}' 'else' '{' Statement
            '}';
        Statement ← 'if' '(' Expression ')' '{' Statement '}';
    }
}

module if_typechecking {
    role(type-checking) {
        0 { if (!$1.type.equals("Boolean")) System.err.println("ERROR: Expression
            must be a boolean"); }
        4 { if (!$5.type.equals("Boolean")) System.err.println("ERROR: Expression
            must be a boolean"); }
    }
}

module if_eval {
    role(evaluation) {
        0 { if (new Boolean($1.eval)) $4.eval else $3.eval; }
        4 { if (new Boolean($5.eval)) $6.eval }
    }
}

slice if {
    module if_syntax with role syntax
    module if_typechecking with role type-checking
    module if_eval with role evaluation
}
```

Listato 2.1: Esempio di implementazione introduttiva del costrutto *if-else*.

corrispondenti a tali ruoli sono eseguite in sequenza in una singola visita all'albero.

Successivamente, i componenti del linguaggio vengono definiti combinando definizioni di sintassi e semantica provenienti da diversi moduli all'interno di costrutti denominati *slice*, ognuno dei quali contiene la definizione di un singolo componente del linguaggio. Infine, il linguaggio viene generato combinando i vari *slice* insieme [17].

Al Listato 2.1 è illustrato un esempio di implementazione del costrutto condizionale *if-else* in Neverlang. Sono definiti tre ruoli: `syntax`, `type-checking` ed `evaluation`, ognuno dei quali è definito in un modulo separato (ma potrebbe anche essere definito in file diversi) e, successivamente, unito agli altri all'interno dello *slice* `If`. Nell'esempio, il costrutto è definito sia in forma contratta (omettendo la parte di *else*) sia in forma estesa. I terminali sono rappresentati dagli elementi racchiusi tra apici, mentre i restanti elementi sono non terminali. I rife-

rimenti ai non terminali sono indicati tramite una notazione numerica, partendo dall'alto verso il basso e da sinistra a destra. Ad esempio, lo `0` presente all'interno del ruolo di `type-checking` indica che la semantica descritta è riferita al primo non terminale (ossia `Statement`). Analogamente, all'interno della definizione della parte semantica, `$i` si riferisce all'i-esimo non terminale.

Tra i vantaggi principali di Neverlang troviamo [5]:

- **Modularità**: Ognuno dei moduli che compongono il linguaggio viene compilato separatamente, permettendo di utilizzarne uno o più di uno (in tal caso aggregandoli in uno *slice*) all'interno di altri linguaggi.

- **Riutilizzo**: Neverlang offre la possibilità di riutilizzare frammenti di linguaggio in più di un contesto. Ad esempio, un frammento può utilizzare la sintassi di un altro frammento definito in precedenza e ridefinirne la semantica, o viceversa. Inoltre, è possibile ridefinire l'ordine dei simboli non terminali utilizzati nella sintassi o nella semantica importata.

- **Estensibilità**: L'architettura modulare utilizzata all'interno di Neverlang facilita l'estensione di linguaggi esistenti. Per aggiungere nuove funzionalità non è necessario modificare il codice, ma è sufficiente integrare un nuovo *slice*.

2.6 Java

In aggiunta a Neverlang, per la realizzazione del progetto è stato utilizzato Java: un linguaggio di programmazione ad alto livello, orientato agli oggetti e a tipizzazione statica, sviluppato da Sun Microsystems nel 1991. È molto diffuso e ben supportato, con una vasta comunità di sviluppatori e una grande quantità di librerie. Uno degli obiettivi principali di Java è quello di essere il più possibile autonomo rispetto alla piattaforma di esecuzione, permettendo di scrivere una volta il codice e farlo eseguire su qualsiasi Java Virtual Machine (JVM), indipendentemente dall'architettura del calcolatore [11].

Java è stato utilizzato per la realizzazione del progetto in quanto Neverlang è progettato per essere completamente integrato con esso. Il suo compilatore (nl-

gc) è stato sviluppato per poter convertire il codice scritto utilizzando il DSL di Neverlang in un nuovo codice supportato dalla JVM. Inoltre, Neverlang permette di utilizzare Java (ma non solo; anche Scala, ad esempio, è supportato) come linguaggio per la definizione della semantica all'interno dei moduli del DSL. Ciò è possibile in quanto gli accessi a variabili non terminali, definiti all'interno della sintassi, sono sostituiti dallo specifico plug-in con accessi alla reale rappresentazione interna del linguaggio. In particolare, l'accesso alle variabili viene effettuato tramite una chiamata all'n-esimo figlio dell'AST [6].

Capitolo 3

Analisi e Requisiti

3.1 Obiettivi

Il progetto si propone di ricreare la sintassi e la logica di Structured Query Language (SQL) all'interno di un DSL implementato utilizzando il framework Neverlang Language Workbench. L'obiettivo principale è modellare il linguaggio seguendo un approccio incrementale ed implementando le varie funzionalità in modo tale che possano essere aggiunte o rimosse dal progetto in modo indipendente.

Il linguaggio deve semplificare la gestione di un ampio quantitativo di dati e favorirne la manipolazione tramite le operazioni sviluppate al suo interno. Lo scopo è aiutare l'utente finale ad estrarre le informazioni di cui ha bisogno a partire dai dati già in suo possesso. Per raggiungere questo obiettivo, il programma deve implementare una serie di procedimenti necessari per la creazione, la modifica e l'eliminazione di tabelle, all'interno delle quali sono memorizzati i dati. Inoltre, è necessario gestire l'inserimento di tali dati, nonché la loro eventuale alterazione od eliminazione. Infine, il linguaggio dovrà fornire le operazioni richieste per selezionare, manipolare ed aggregare l'insieme di informazioni presenti nel database.

3.2 Requisiti funzionali

Il DSL dovrà implementare un insieme di caratteristiche tipiche di SQL. Di seguito vengono elencate le principali:

- **Gestione delle tabelle**: Per modellare la base di dati secondo le proprie esigenze, l'utente ha a disposizione una serie di operazioni per creare o modificare le tabelle all'interno del database:

  - La funzione `CREATE TABLE` permette di creare una nuova tabella e di specificare le colonne che la compongono. Ciascuna colonna può avere vincoli sui dati che conterrà. Ad esempio, il vincolo espresso di default durante la fase di creazione della tabella riguarda il tipo dei dati inseriti in ogni colonna, il quale dovrà essere coerente con quanto specificato nella definizione della tabella. In aggiunta, è possibile indicare altre caratteristiche che il dato inserito dovrà rispettare, come l'unicità (ossia che non sia uguale a un altro dato già presente nella tabella), la non nullità (l'informazione non può essere omessa) oppure la definizione dell'attributo come chiave della tabella, comprendendo i vincoli di unicità e non nullità precedentemente descritti.
  - L'operazione `ALTER TABLE` permette di modificare le tabelle già presenti nel database. Quando tale costrutto viene utilizzato, è necessario specificare la natura della modifica, ossia indicare se si desidera aggiungere o rimuovere una colonna dalla tabella.
  - `DELETE TABLE` permette di rimuovere una tabella dalla base dati, eliminando anche i dati in essa contenuti.

- **Gestione dei dati**: Analogamente a quanto descritto per le tabelle, è importante fornire la possibilità di gestire i dati al loro interno. Le funzioni necessarie da implementare nel progetto sono l'esatta trasposizione di quelle descritte per la gestione delle tabelle. In particolare, il costrutto `INSERT INTO` permette di inserire dati all'interno di una tabella, `UPDATE` consente di modificare uno o più attributi di un sottoinsieme di dati (ottenuto tramite una precisa selezione) e, infine, `DELETE` permette di eliminare una o più tuple.

- **Manipolazione dei dati**: Un aspetto fondamentale di un linguaggio di questo tipo è l'utilizzo dell'insieme di informazioni contenute nel database per estrarre nuovi dati. Questa operazione può essere eseguita grazie all'implementazione dei seguenti comandi nella sintassi del DSL:

  - Tra le istruzioni più importanti troviamo `SELECT` e `FROM`. La prima permette di specificare un sottoinsieme di attributi da ottenere, mentre la seconda viene utilizzata per indicare la tabella sorgente dalla quale esportare gli attributi selezionati.
  - Il comando `WHERE` consente, utilizzandolo in combinazione con espressioni booleane o specifici selettori (ad esempio `IS NULL`), di filtrare le tuple della tabella dalla quale si stanno estraendo i dati.
  - Infine, `ORDER BY` viene utilizzato per indicare come ordinare le tuple del risultato. È possibile specificare anche più attributi per determinare l'ordine in caso di uguaglianza dell'attributo precedente.

- **Aggregazione di dati**: L'aggregazione di dati è il processo di combinare più elementi di informazione per ottenere un unico risultato sintetico. Quando si lavora con una quantità consistente di informazioni, è spesso utile raggruppare questi dati in insiemi basati su uno o più attributi comuni. Il comando `GROUP BY` consente di effettuare questa operazione, specificando gli attributi in base ai quali raggruppare i dati. Una volta creati questi gruppi, si possono applicare delle funzioni predefinite, chiamate funzioni di aggregazione, per combinare le informazioni all'interno di ogni gruppo in un'unica rappresentazione. Nello specifico si può calcolare la somma, la media o il conteggio degli elementi del sottogruppo (rispettivamente utilizzando `SUM`, `AVG` e `COUNT`), oppure la selezione dell'elemento con il valore massimo o minimo in ogni collezione di tuple (tramite `MAX` e `MIN`).

Al Listato 3.1, sono forniti esempi di codice SQL utilizzati per le operazioni descritte all'interno della sezione corrente. Tali esempi rappresentano in modo chiaro le funzionalità e la sintassi che il linguaggio dovrà implementare.

```sql
/* Esempio di query SQL per la gestione di tabelle in un database */
CREATE TABLE Customer (
    CustomerID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Email VARCHAR(100) NOT NULL,
    Age INT,
    City VARCHAR(50)
);

ALTER TABLE Customer
ADD PhoneNumber VARCHAR(20);

ALTER TABLE Customer
DROP Email;

DROP TABLE Customer;

/* Esempi di comandi per l'inserimento, l'aggiornamento e la cancellazione dei
    dati */
INSERT INTO Customer (CustomerID, FirstName, LastName, Email)
VALUES (1, 'John', 'Doe', 'john.doe@example.com');

UPDATE Customer
SET FirstName = 'Jane'
WHERE CustomerID = 1;

DELETE FROM Customer
WHERE CustomerID = 1;

/* Esempi di query SQL per la manipolazione dei dati */
SELECT * FROM Customer;

SELECT *
FROM Customer
WHERE FirstName = 'John' AND LastName = 'Doe'
ORDER BY LastName DESC, FirstName ASC;

/* Esempi di query con funzioni di aggregazione */
SELECT LastName, COUNT(*)
FROM Customer
GROUP BY LastName;

SELECT City, AVG(Age)
FROM Customer
GROUP BY City;

/* Esempio di query contente tutte le clausole definite dal linguaggio */
SELECT City, COUNT(CustomerID)
FROM Customer
WHERE Age >= 30 AND Age <= 50
GROUP BY City
ORDER BY City;
```

Listato 3.1: Esempio di codice scritto in SQL, che illustra un caso di utilizzo per ciascuna operazione richiesta nel progetto.

3.3 Requisiti non funzionali

Nel contesto del progetto, vengono considerati i seguenti requisiti non funzionali:

- **Modularità e riutilizzabilità**: Tra i requisiti essenziali, il progetto prevede un'implementazione dei comandi descritti nella Sezione 3.2 in modo incrementale, modulare e riutilizzabile. In questo modo, ogni caratteristica del linguaggio sarà definita per incrementare l'indipendenza e limitare, quindi, le dipendenze da moduli esterni. Tale approccio favorisce non solo l'aggiunta e la rimozione di funzionalità al suo interno, ma anche il riutilizzo di componenti in altri progetti.

- **Facilità di evoluzione e manutenzione**: Un ulteriore requisito riguarda l'eventuale evoluzione e manutenzione del linguaggio. Neverlang supporta un'evoluzione agile delle specifiche del linguaggio, rendendo possibile la modifica di sintassi e semantica senza dover rifattorizzare l'intero progetto. Questa caratteristica è particolarmente utile per adattare il progetto rapidamente alle nuove esigenze o integrazioni.

- **Facilità di sviluppo**: Le API del framework Neverlang sono progettate per rendere la definizione del linguaggio più intuitiva e meno soggetta a errori. Per tale motivo, è ragionevole attendersi non solo un processo di sviluppo più veloce, ma anche una maggiore qualità del linguaggio prodotto.

- **Compatibilità con SQL**: Trattandosi di un linguaggio modellato ispirandosi ad SQL, il progetto dovrà essere in grado di interpretare e processare query SQL, permettendo l'esecuzione delle operazioni desiderate in modo efficiente e preciso, in modo compatibile con gli standard del linguaggio esistente.

3.4 Modellazione del dominio

Il dominio applicativo del progetto prevede la memorizzazione delle informazioni all'interno di strutture che ne facilitino la manipolazione tramite le operazioni implementate nel linguaggio. La struttura chiave di Neverlang-SQL è il database, al cui interno sono memorizzate le tabelle utilizzate dall'utente. Ciascuna tabella

è identificata da un nome univoco che deve essere diverso da quello delle altre tabelle per garantire una corretta distinzione all'interno del database. Ogni tabella contiene una o più colonne, ciascuna identificata tramite un nome e potenzialmente soggetta a dei vincoli che dovranno essere rispettati dagli attributi dei dati inseriti. Un esempio di questi vincoli è che il tipo dell'attributo inserito deve corrispondere a quello specificato nella dichiarazione della tabella, ma esistono inoltre altri vincoli più specifici, come quello di unicità (il valore non può già esistere nella tabella) o quello di non nullità (il valore non può essere omesso). Infine, l'inserimento delle informazioni all'interno della tabella avviene tramite le tuple, ognuna delle quali rappresenta una "riga" della tabella, e quindi un singolo dato. Gli attributi della tupla devono necessariamente corrispondere a quelli richiesti dalla definizione della tabella in cui viene inserita.

Gli elementi appena descritti e le loro relazioni sono sintetizzati nella Figura 3.1. Le operazioni corrispondenti alla semantica di ogni clausola del linguaggio utilizzano il modello appena descritto per poter modificare la struttura del database, delle tabelle e dei dati, o per effettuare delle estrazioni di informazioni interrogando la base dati.

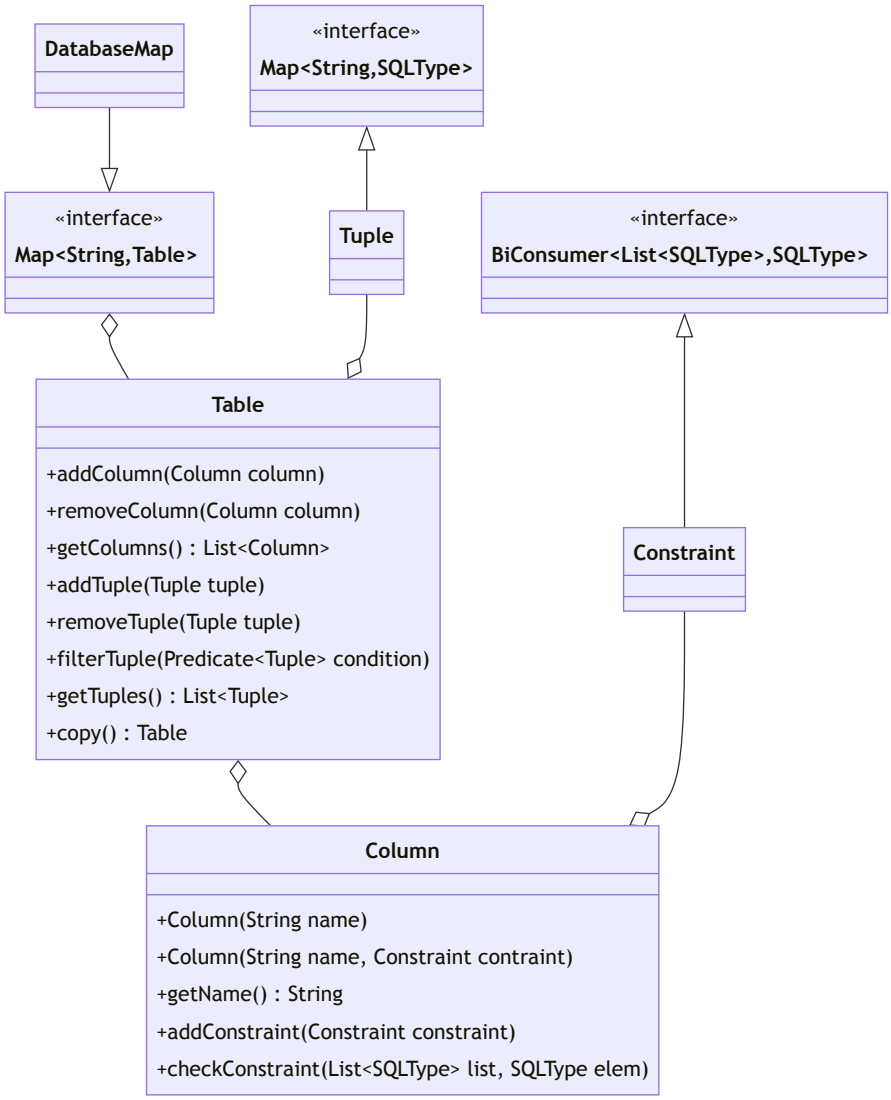

Figura 3.1: Schema UML dell'analisi del problema, al suo interno sono rappresentate le entità principali coinvolte nella memorizzazione dei dati e le relazioni tra di esse.

Capitolo 4

Design e Implementazione

4.1 Architettura

Ad alto livello, l'architettura del progetto proposto è rappresentata nella Figura 4.1. Essa si suddivide in due sezioni principali, ciascuna delle quali ricopre un ruolo fondamentale per l'implementazione del DSL:

- Parte implementata in Neverlang: Questa sezione si occupa della definizione delle regole sintattiche di SQL, descrivendo come i vari simboli che compongono la grammatica del DSL sono utilizzati e combinati, nonché prodotti. Oltre alla sintassi, Neverlang è utilizzato per specificare la semantica del linguaggio, ovvero il comportamento di ciascun costrutto SQL durante l'esecuzione. Ogni elemento di Neverlang-SQL (ad esempio, `CREATE TABLE`, `SELECT`, `WHERE`, ecc.) è rappresentato utilizzando un modulo Neverlang dedicato, all'interno del quale sono definite sia la grammatica che le azioni semantiche necessarie per la corretta interpretazione ed esecuzione delle funzioni ad esso associate.

- Parte implementata in Java: Le classi implementate in questa sezione del progetto sono progettate principalmente per gestire i dati memorizzati nelle strutture create appositamente per il dominio sul quale opera SQL. Queste classi modellano entità come il database, le tabelle, le colonne, le tuple, e i tipi di dati, fornendo metodi per manipolare ciascuna di esse in modo conforme alle specifiche del linguaggio.

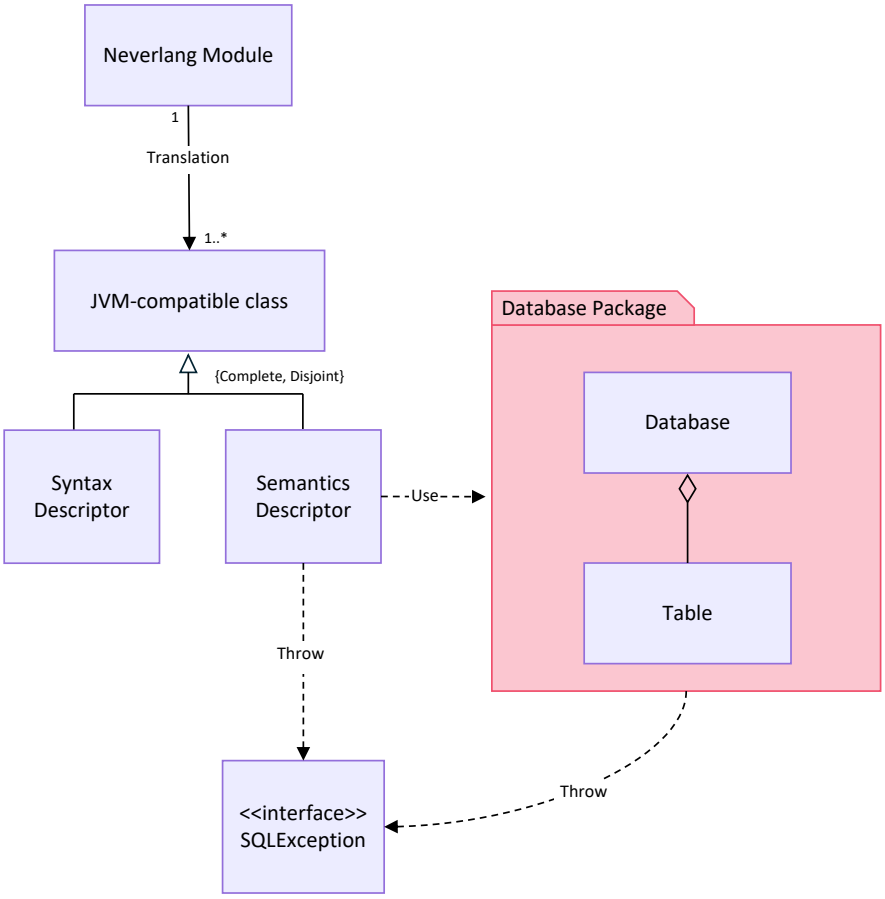

Figura 4.1: Diagramma dell'architettura del progetto, illustra le principali dipendenze tra i componenti e i vari moduli.

Come mostrato nel diagramma in Figura 4.1, ogni modulo di Neverlang (e le relative specifiche sintattiche e semantiche) viene tradotto in file sorgenti compatibili con la JVM tramite il compilatore del framework, chiamato *nlgc*. I file così generati sono successivamente utilizzati da *javac*[1] per produrre file di classe, impiegati per rappresentare gli oggetti Java [17, p. 19]. In particolare, il codice generato si suddivide principalmente in due categorie: sintattiche e semantiche. Infatti, per ogni modulo con annessa una specifica semantica, Neverlang produce una classe chiamata convenzionalmente `<module-name>$role$syntax`. Allo stesso modo, per ogni ruolo indicato e per ogni non terminale definito all'interno del mo-

[1] Javac rappresenta il compilatore principale utilizzato dal linguaggio di programmazione Java; si occupa di tradurre il codice sorgente in un formato eseguibile dalla JVM.

dulo, viene prodotta una classe atta a descriverne il comportamento, denominata `<module-name>$role$<role-name>$<node-number>` [17, pp. 21-22]. Le classi di sintassi sono utilizzate per verificare che le regole grammaticali siano rispettate nelle query SQL fornite in input. Questo processo di validazione si propone di controllare che la struttura delle query segua correttamente le specifiche di SQL, prevenendo errori comuni di sintassi. Di maggiore rilevanza per l'esecuzione del linguaggio è la parte riguardante il comportamento di ogni elemento, gestita attraverso le classi di semantica. Ciascuna di queste classi è responsabile di una fase specifica dell'elaborazione ed esecuzione del nodo dell'AST al quale sono riferite. Queste classi contengono la logica che determina il comportamento esatto di Neverlang-SQL durante l'esecuzione delle query, permettendo inoltre di interagire con la parte modellata in Java e che modella il dominio all'interno del quale il DSL viene applicato.

Le classi implementate tramite il linguaggio di programmazione Java, come `Database`, `Table`, `Column`, e `Tuple`, sono strettamente integrate con le classi semantiche generate dal compilatore di Neverlang. Questa integrazione permette alle ultime di eseguire operazioni dirette sul dominio modellato, quali la modifica della struttura delle tabelle, l'inserimento e la manipolazione dei dati, e l'esecuzione di query complesse per ottenere informazioni specifiche richieste dall'utente.

Inoltre, sia le classi Java che quelle generate da Neverlang sono progettate per gestire le eccezioni. Il design proposto in questa sezione dell'elaborato include la generazione e la gestione di eccezioni personalizzate, ognuna delle quali rappresenta un errore specifico che può verificarsi durante l'analisi o l'esecuzione delle query, come errori di sintassi, accessi non validi ai dati od operazioni non consentite.

4.2 Sintassi

La progettazione della sintassi di Neverlang-SQL si basa su una struttura definita suddividendo il linguaggio in cinque macro-aree principali: i fondamenti del linguaggio, la gestione delle tabelle, la gestione dei dati, la gestione delle interrogazioni al database e l'aggregazione di dati. Ciascuna delle sezioni elencate è definita per soddisfare precisi requisiti del linguaggio ed è implementata incrementalmente,

in modo da ottenere una versione funzionante del DSL che viene migliorata con ogni aggiunta.

4.2.1 Fondamenti del linguaggio

Il design del linguaggio è fondato sull'interpretazione di ogni query ricevuta in input come una sequenza di operazioni indipendenti l'una dall'altra, le quali possono essere eseguite in modo sequenziale. Questa struttura sintattica modulare consente di ottenere una grande flessibilità nell'esecuzione di operazioni complesse e nella combinazione di più comandi SQL in un'unica istruzione. All'interno di questa rappresentazione, il ruolo fondamentale è ricoperto dal simbolo non terminale `Operation`, il quale rappresenta una singola operazione da eseguire, come la creazione di una tabella o l'inserimento di dati.

Unione di più operazioni Per supportare la concatenazione di più operazioni in una singola query, viene utilizzato il non terminale `OperationList`. Questo simbolo può rappresentare una singola operazione o una sequenza di operazioni, ognuna delle quali è separata dall'operatore punto e virgola (`;`). Inoltre, `OperationList` è anche l'elemento utilizzato per definire l'assioma[2] del linguaggio: `Program`.

La definizione ricorsiva della lista, illustrata in Figura 4.2, permette di definire sequenze di operazioni di lunghezza arbitraria, offrendo agli utenti una discreta flessibilità per la scrittura delle query. Tale struttura non solo facilita l'esecuzione di operazioni multiple ma agevola anche l'estensione delle regole grammaticali del DSL: nuovi tipi di operazioni potranno essere aggiunti a quelle già esistenti senza dover modificare la sintassi di base proposta nel paragrafo corrente. Concludendo, l'approccio modulare proposto non solo migliora la chiarezza del codice ma aumenta anche la manutenibilità del linguaggio, permettendo l'aggiornamento o la modifica di singole operazioni senza influire sull'intero sistema.

Di seguito viene riportata la definizione della grammatica discussa, illustrando come i vari simboli sono definiti:

```
<Program> ::= <OperationList>
<OperationList> ::= [<OperationList> ";"] <Operation>
```

[2]Definito anche "simbolo di partenza", convenzionalmente rappresenta il punto di partenza di ogni linguaggio definito utilizzando Neverlang.

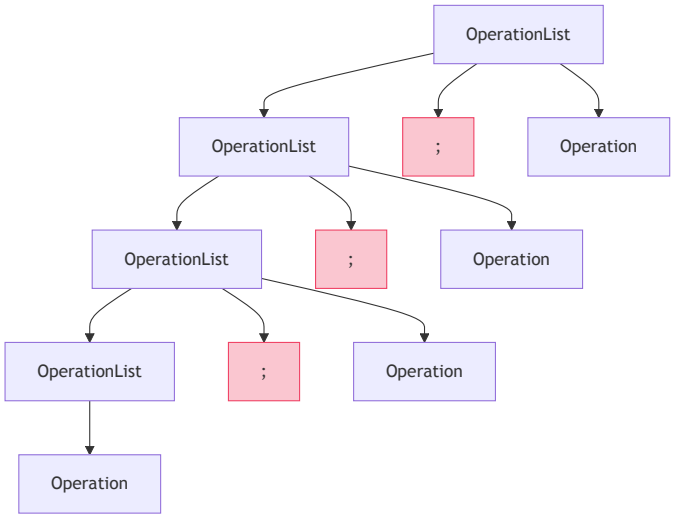

Figura 4.2: Esempio di albero sintattico di una lista di operazioni

Per poter modellare una grammatica maggiormente flessibile, è stato introdotto anche il simbolo `EmptyOperation`. La scelta è finalizzata a evitare una struttura grammaticale rigida, permettendo così all'utente finale di decidere se concludere l'ultima operazione della query con il simbolo terminale (rappresentato dal punto e virgola) oppure no, senza che l'esecuzione generi un errore.

Identificare gli elementi Un altro componente fondamentale del linguaggio è il simbolo `Id`, che rappresenta un identificatore univoco per gli elementi del database, come tabelle e colonne. `Id` è modellato tramite un'espressione regolare (*regex*) che consente di selezionare stringhe composte da almeno un carattere alfabetico, seguito opzionalmente da ulteriori caratteri, numeri o un carattere di sottolineatura. Questa flessibilità permette a `Id` di adattarsi a un ampio insieme di convenzioni di denominazione, garantendo che gli identificatori siano sempre validi all'interno del contesto del database. L'utilizzo di una *regex* permette di validare gli identificatori in modo efficiente durante l'esecuzione, tramite l'uso di pattern o parole chiave, riducendo il rischio di errori dovuti a nomi non validi e migliorando così la robustezza del linguaggio[3].

[3]Uno specifico componente di Neverlang, chiamato DEXTER, include al suo interno un analizzatore lessicale che permette di identificare i lessemi a tempo d'esecuzione [7, 17].

4.2.2 Gestione delle tabelle

Le operazioni necessarie per gestire efficacemente le tabelle all'interno del linguaggio Neverlang-SQL sono: `CREATE TABLE`, `ALTER TABLE` e `DROP TABLE`. Queste operazioni forniscono agli utenti la capacità di definire nuove tabelle, modificare quelle esistenti o rimuoverle dal database. La sintassi delle operazioni descritte è la seguente:

```
<Operation> ::= "CREATE" "TABLE" <Id> "(" <ColumnList> ")"
<Operation> ::= "ALTER" "TABLE" <Id> ("ADD" <Column> |"DROP" <Id>)
<Operation> ::= "DROP" "TABLE" <Id>
```

Ogni operazione rispetta il principio della modularità discusso nella Sezione 4.2.1, producendo un elemento `Operation` che rappresenta un'azione completa e indipendente all'interno del DSL. Questa indipendenza permette alle operazioni di essere utilizzate singolarmente o concatenate con altre operazioni, offrendo flessibilità nella realizzazione delle query.

Considerando che `CREATE TABLE` è il costrutto con la sintassi più complessa tra quelli definiti, esso viene utilizzato come esempio per illustrare il funzionamento di queste operazioni.

Nella Figura 4.3 viene presentato un AST generato dall'esecuzione di una query, la quale ha come obiettivo la creazione di una nuova tabella all'interno del database. In rosso sono indicati i simboli terminali, mentre in blu sono indicati i simboli non terminali. La struttura dell'albero riflette la gerarchia e le dipendenze tra i diversi componenti della query, mostrando come ciascun elemento della sintassi viene interpretato dal linguaggio. In particolare, l'operazione d'esempio richiede la creazione di una tabella chiamata Customer, contenente due attributi. Il primo attributo, definito come chiave primaria (che non può essere nullo e deve essere unico per ogni tupla inserita nella tabella), rappresenta il numero intero identificativo di ogni cliente ed è chiamato "CustomerID". Questo attributo è progettato per garantire l'unicità di ogni cliente all'interno del database, essendo un elemento essenziale per mantenere l'integrità dei dati. Il secondo attributo previsto per la tabella Customer è la colonna "Email", che è destinata a contenere l'indirizzo email di ogni cliente. Questa colonna è definita come una stringa con una

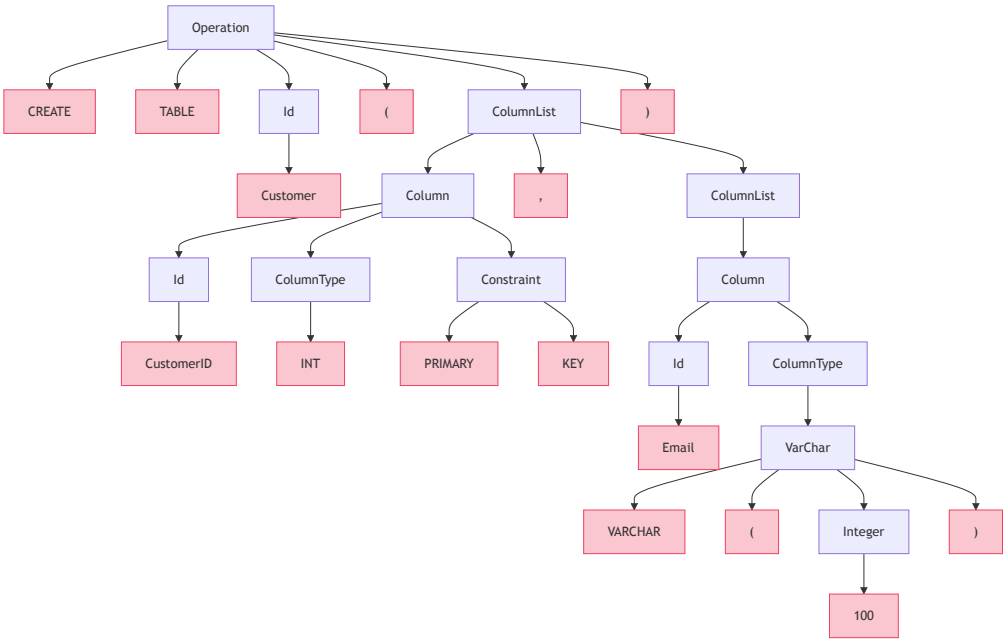

Figura 4.3: Illustrazione rappresentante un esempio di albero sintattico generato per la creazione di una tabella.

lunghezza massima di 100 caratteri, garantendo così che tutti gli indirizzi email inseriti rispettino un formato standardizzato e un limite di lunghezza specificato.

All'interno dell'albero proposto possiamo individuare anche altri importanti elementi utilizzati per definire in modo chiaro la tabella da inserire:

- **Tipologie di dati**: Sono definiti i tipi di dati supportati per le colonne della tabella ed includono `INT`, `FLOAT`, `VARCHAR` e `BOOLEAN`. Questi tipi di dati rappresentano le principali opzioni per quanto concerne la gestione delle informazioni all'interno di un database, consentendo la rappresentazione di numeri interi, numeri decimali, stringhe di testo e valori booleani. In particolare, la definizione della stringa prevede anche la precisazione del numero massimo di caratteri accettati[4].

- **Vincoli**: Quando si definisce una tabella, e in particolare i suoi attributi, possono essere esplicitati una serie di vincoli che devono essere rispettati dai

[4]Il numero massimo di caratteri di `VARCHAR` rappresenta un particolare esempio di vincolo implicito sui dati da inserire nella colonna. Un altro esempio è che il tipo del dato deve corrispondere al tipo specificato nella dichiarazione della colonna.

dati per garantirne l'integrità. I vincoli supportati includono `UNIQUE`, che garantisce l'unicità dei valori in una colonna, `NOT NULL`, che assicura che una colonna non contenga valori nulli, e `PRIMARY KEY`, che identifica univocamente ogni riga all'interno della tabella e non permette valori duplicati.

4.2.3 Gestione dei dati

La gestione dei dati in Neverlang-SQL include le operazioni fondamentali per manipolare i dati all'interno delle tabelle del database. Le principali operazioni supportate in questa macro-area sono `INSERT`, `UPDATE` e `DELETE`. Rispettivamente, queste operazioni permettono agli utenti di aggiungere nuove righe, modificare i dati esistenti e rimuovere righe dalle tabelle. La sintassi di queste operazioni è:

```
<Operation> ::= "INSERT" "INTO" <Id> "(" <IdList> ")" "VALUES" "(" <ValueList> ")"
<Operation> ::= "DELETE" <SelectedData>
<Operation> ::= "UPDATE" <Id> "SET" <SetList> "WHERE" <BoolExpression>
```

Ogni operazione è progettata per essere intuitiva e riflettere le operazioni comunemente utilizzate nei sistemi SQL. L'operazione `INSERT` permette di specificare una lista di colonne in cui inserire i valori e una lista di valori corrispondenti. L'operazione `UPDATE` consente di modificare i valori esistenti in una o più colonne specificando una condizione per filtrare le righe da aggiornare. Infine, l'operazione `DELETE` permette di rimuovere un sottoinsieme di righe da una tabella.

L'inclusione di queste operazioni all'interno di Neverlang-SQL consente agli utenti di gestire i dati in modo flessibile e robusto, garantendo al contempo che tutte le modifiche ai dati siano conformi alle regole di integrità definite all'interno del database. Ogni operazione di gestione dei dati è progettata per essere un'operazione `Operation` indipendente, che può essere utilizzata singolarmente o combinata con altre operazioni in una singola query.

Liste di oggetti Le liste `IdList`, `ValueList` e `SetList` sono progettate in modo analogo alla lista di operazioni `OperationList` descritta nella Sezione 4.2.1. Questa progettazione consente di rappresentare sequenze flessibili di identificatori, valori e assegnazioni nelle query SQL, permettendo di specificare un numero variabile di elementi secondo le necessità dell'utente. Ogni lista può contenere uno o

più elementi, utilizzando una struttura ricorsiva che facilita la costruzione di query articolate e complesse.

Tipi di valori I valori utilizzati nelle query sono definiti in base a un insieme di tipologie predefinite, ciascuna caratterizzata da un formato specifico che il linguaggio è in grado di riconoscere. I tipi di valori supportati in Neverlang-SQL includono:

- **Stringhe**: Le stringhe sono rappresentate come sequenze di caratteri racchiuse tra virgolette singole o doppie. Neverlang-SQL riconosce le stringhe che possono includere caratteri speciali, garantendo che l'intero contenuto tra le virgolette sia trattato come un unico valore di testo.

- **Interi**: I valori interi sono numeri composti da una sequenza di cifre, che possono essere preceduti da un segno positivo o negativo. All'interno del progetto, sono considerati validi i valori che rappresentano quantità discrete, formati correttamente come numeri interi e senza parti decimali.

- **Numeri in virgola mobile**: Questi valori rappresentano numeri con una parte decimale. Possono includere un segno positivo o negativo e supportano anche la notazione scientifica con un esponente (ad esempio "1.23e-2" rappresenta il numero 1.23×10^{-2}, ossia 0.0123). Questo consente di rappresentare una vasta gamma di valori numerici, inclusi numeri molto piccoli o molto grandi con precisione.

- **Booleani**: I valori booleani sono rappresentati dalle parole chiave `TRUE` o `FALSE`. Questi valori sono utilizzati per indicare stati di verità binari e vengono riconosciuti come tali solo quando corrispondono esattamente a una delle due parole chiave, garantendo un'interpretazione corretta nelle operazioni logiche.

- **Valori Nulli**: Il valore `NULL` indica l'assenza di un valore. È una parola chiave specifica che Neverlang-SQL riconosce per rappresentare la mancanza di dati in un campo. Questo valore è trattato in modo distinto dagli altri tipi, permettendo di gestire dati mancanti o opzionali all'interno del database.

Espressioni booleane Il linguaggio sviluppato supporta tre tipologie principali di funzioni booleane, ognuna progettata per offrire flessibilità e potenza espressiva nella costruzione di query complesse. Queste funzioni sono fondamentali per il filtraggio e l'estrazione dei dati dal database, consentendo agli utenti di definire criteri di selezione precisi e ottimizzati.

- La prima tipologia di funzioni booleane riguarda i confronti diretti tra attributi di una colonna e un valore specificato dall'utente. Queste funzioni permettono di eseguire confronti utilizzando operatori come `=`, `<>`, `<`, `>`, `<=`, e `>=`. Ad esempio, è possibile selezionare tutte le righe di una tabella in cui il valore di una colonna specifica è maggiore di un certo numero, o dove un campo di testo corrisponde esattamente a una stringa specificata. Questa capacità di confronto diretto è essenziale per operazioni di filtraggio di base, fornendo un controllo sui dati che vengono recuperati dal database.

- La seconda tipologia si concentra sulle funzioni booleane native di SQL, come `BETWEEN`, `IN` e `IS NULL`. Queste funzioni sono particolarmente utili per eseguire controlli più complessi e specifici. Ad esempio, la funzione `BETWEEN` permette di verificare se un valore rientra all'interno di un determinato intervallo, mentre `IN` consente di verificare se un valore appartiene a un insieme predefinito di valori. Queste funzioni forniscono un modo semplice e intuitivo per esprimere condizioni che coinvolgono intervalli o insiemi, rendendo le query più leggibili e facili da scrivere. Infine, la funzione `IS NULL` è utilizzata per ricercare tuple con precisi attributi non dichiarati.

- La terza tipologia è rappresentata dalla capacità di combinare espressioni booleane pre-esistenti tramite operatori logici come `AND`, `OR` e `NOT`. Questa funzionalità consente agli utenti di creare condizioni più complesse ed articolate, combinando più criteri di selezione in una singola operazione. Ad esempio, è possibile costruire una query che seleziona tutte le righe in cui un attributo è maggiore di un certo valore e un altro attributo corrisponde a una stringa specifica, oppure che recupera tutte le righe in cui un attributo non è uguale a un determinato valore o rientra in un intervallo specificato. L'uso di `NOT` permette di escludere determinati risultati, invertendo la logica delle condizioni definite.

Queste operazioni sono particolarmente utili per la gestione della selezione dei dati da aggiornare nelle operazioni `UPDATE`, in cui la precisione nella definizione dei criteri di modifica è fondamentale per garantire la modifica richiesta dall'utente finale. Inoltre, le stesse funzioni booleane saranno utilizzate dagli operatori di selezione, di cui si parlerà nella Sezione 4.2.4, per selezionare le informazioni che l'utente vuole ottenere dal database.

4.2.4 Interrogazioni al database

Le interrogazioni al database rappresentano una delle funzionalità essenziali di Neverlang-SQL, consentendo agli utenti di estrarre, filtrare e ordinare dati in base a criteri specifici. Le operazioni di interrogazione implementate nel linguaggio includono `SELECT`, `FROM`, `WHERE` e `ORDER BY`. Queste operazioni permettono di recuperare informazioni strutturate dalle tabelle, applicare filtri per ottenere solo i dati rilevanti e ordinare i risultati secondo criteri definiti dall'utente. La combinazione di queste operazioni permette di costruire query complesse e dettagliate, rispondendo a diverse esigenze di analisi dei dati.

La grammatica per le operazioni di interrogazione in Neverlang-SQL è progettata in modo da fornire una struttura flessibile e modulare. Le regole grammaticali per le interrogazioni sono le seguenti:

```
<Operation> ::= <SelectedData
<SelectedData> ::= "FROM" <Id>
<SelectedData> ::= "SELECT" ("*"|<IdList>) <SelectedData>
<SelectedData> ::= <SelectedData> "WHERE" <BoolExpr>
<SelectedData> ::= <SelectedData> "ORDER" "BY" <OrderList>
```

Questa definizione grammaticale consente di costruire query che selezionano, filtrano e ordinano i dati in modo estremamente flessibile. Un elemento chiave di questa progettazione è l'uso del simbolo `SelectedData` sia come input che come output per le diverse operazioni. Questo approccio garantisce la coerenza nella sintassi delle query e permette di combinare le operazioni in modo fluido, senza restrizioni rigide sulla loro sequenza. La scelta di strutturare le operazioni in modo che ogni operazione di interrogazione utilizzi e restituisca il simbolo `SelectedData` è fondamentale per mantenere la flessibilità del linguaggio. Tramite questa configurazione, l'output di una clausola come `WHERE` può essere utilizzato direttamente

come input per un'altra operazione, come `ORDER BY` o `SELECT`. Questo significa che le operazioni possono essere combinate in modo flessibile, senza che una dipenda necessariamente dall'altra. Ad esempio, una query potrebbe iniziare con la selezione di tutte le colonne da una tabella (`SELECT`), seguita da un filtro sui dati recuperati (`WHERE`) e, infine, ordinare i risultati secondo una specifica colonna (`ORDER BY`). In alternativa, potrebbe omettere l'operazione `WHERE` e includere solo `ORDER BY`. Grazie all'uso ricorsivo di `SelectedData`, ogni operazione aggiunge una trasformazione sui dati senza la necessità di definire esplicitamente tutte le altre operazioni all'interno della grammatica.

Di seguito sono riassunti i principali vantaggi ottenuti dall'utilizzo di tale struttura:

- **Modularità** e coerenza: Utilizzando lo stesso simbolo `SelectedData` per rappresentare il risultato di qualsiasi operazione di interrogazione, il linguaggio mantiene una coerenza sintattica che semplifica l'apprendimento e l'uso da parte degli utenti. La modularità della grammatica permette di definire operazioni complesse senza introdurre ambiguità o conflitti sintattici.

- **Estensibilità del linguaggio**: La struttura progettuale facilita l'espansione del linguaggio. Poiché ogni nuova operazione di interrogazione può essere definita come una trasformazione che prende in input `SelectedData` e restituisce un altro `SelectedData`, è possibile aggiungere nuove operazioni seguendo lo stesso modello senza modificare la struttura di base del linguaggio. Ad esempio, l'aggiunta della clausola `GROUP BY`, che verrà approfondita nella Sezione 4.2.5.

- **Flessibilità nella costruzione delle query**: Gli utenti possono combinare liberamente le operazioni di interrogazione, utilizzandone un sottoinsieme a scelta, senza essere vincolati da una sequenza rigida di operazioni. Questo rende il linguaggio particolarmente adatto per scenari complessi di analisi dei dati, dove le esigenze di interrogazione possono variare notevolmente.

- **Facilità di manutenzione ed evoluzione**: Grazie alla sua struttura modulare, il linguaggio è facile da mantenere e aggiornare. Nuove funzionalità

possono essere aggiunte senza richiedere significative riscritture del codice esistente. Questo rende Neverlang-SQL non solo robusto ma anche sostenibile nel lungo termine.

L'approccio alle operazioni di interrogazione proposto, basato sull'uso ricorsivo e flessibile del simbolo `SelectedData`, offre un equilibrio ideale tra potenza espressiva e semplicità sintattica. Questo design non solo rende il linguaggio facile da imparare e utilizzare, ma ne facilita anche l'espansione futura per supportare nuove funzionalità e requisiti di interrogazione avanzati. La capacità di combinare operazioni senza restrizioni rigide rende il linguaggio considerevolmente più versatile per l'interrogazione del database.

4.2.5 Aggregazione di dati

L'aggregazione di dati è una funzionalità avanzata di Neverlang-SQL che permette di raggruppare i dati e calcolare valori aggregati come conteggi, somme, medie, minimi e massimi. Queste operazioni sono essenziali per l'analisi dei dati, consentendo agli utenti di ottenere informazioni riassuntive da collezioni di dati di grandi dimensioni e di esplorare pattern e tendenze al loro interno.

Per supportare l'aggregazione di dati, è necessario adattare la clausola `SELECT` in modo che sia compatibile con i requisiti imposti dalla clausola `GROUP BY`. Questa scelta è dovuta al fatto che, quando i due operatori sono utilizzati in combinazione, gli `Id` specificati dalla `SELECT` non rappresentano più solamente gli attributi da estrarre ma anche quelli da generare tramite l'aggregazione. Per questo motivo, la grammatica è definita come segue:

```
<SelectedData> ::= "SELECT" <IdList> <SelectedData> "GROUP" "BY" <IdList>;
```

Per operare correttamente, è necessario un controllo rigoroso tra gli identificatori (`Id`) elencati nella clausola `SELECT` e quelli presenti nella clausola `GROUP BY`. In SQL standard, tutti gli identificatori specificati nella clausola `SELECT` devono essere o inclusi nella clausola `GROUP BY` o essere utilizzati all'interno di una funzione di aggregazione (come `SUM`, `AVG`, `MAX`, `MIN`, `COUNT`). Questo vincolo garantisce

che le query restituiscano risultati significativi e coerenti, poiché solo i dati che sono stati raggruppati possono essere utilizzati per calcolare valori aggregati[5].

Le funzioni di aggregazione sono trattate come elementi identificativi (`Id`) all'interno della grammatica del progetto, il che consente di utilizzarle in modo flessibile all'interno delle query. La grammatica per le funzioni di aggregazione è definita come segue:

```
<Id> ::= "COUNT" "(" (<Id>|"*") ")"
<Id> ::= ("SUM"|"AVG"|"MAX"|"MIN") "(" <Id> ")"
```

Queste definizioni mostrano come le funzioni di aggregazione siano modellate. In particolare, la funzione `COUNT` può essere applicata a un attributo o all'intera tabella usando l'asterisco (`*`), per contare tutte le righe che soddisfano i criteri della query. Le altre funzioni di aggregazione, come `SUM`, `AVG`, `MAX` e `MIN`, sono applicate specificamente a una colonna, calcolando rispettivamente la somma, la media, il valore massimo e il valore minimo dei dati raggruppati.

Modellando le funzioni di aggregazione come identificatori, Neverlang-SQL consente di integrarli direttamente nelle query come parte degli identificatori nella clausola `SELECT`. Questo approccio semplifica la sintassi e mantiene la coerenza del linguaggio, consentendo agli utenti di includere funzioni di aggregazione insieme ad altri identificatori nella stessa lista di selezione.

4.3 Semantica

La semantica del DSL ricopre il ruolo di definire il comportamento operativo che le diverse costruzioni sintattiche del linguaggio dovranno rispettare durante l'esecuzione. Mentre la sintassi stabilisce le regole su come devono essere strutturate le query per essere considerate valide, la semantica si concentra su cosa tali query effettivamente fanno quando vengono eseguite. In particolare, l'implementazione della sintassi è realizzata utilizzando il linguaggio di programmazione Java e le sue librerie, nonché le classi modellate ad-hoc per il dominio applicativo del progetto[6].

[5] Il vincolo descritto è fondamentale e l'obiettivo dell'implementazione semantica del costrutto è la sua corretta individuazione

[6] Questo comportamento è necessario perché Neverlang include solo una piccola libreria di funzioni di utilità. In generale, viene incoraggiato l'utilizzo delle librerie standard appartenenti all'ecosistema Java [17].

All'interno del capitolo corrente sono esplorate le diverse fasi del processo semantico, che comprendono il controllo della struttura e la valutazione del nodo corrente.

4.3.1 Fase di controllo della struttura

A causa dell'alto grado di libertà concesso dalla sintassi delle operazioni utilizzate nelle interrogazioni del database[7], Neverlang-SQL include una fase preliminare di controllo della struttura. In questa fase, il linguaggio verifica che la struttura complessiva della query sia accettabile e coerente con le regole semantiche definite. Ad esempio, non è permesso includere una clausola `ORDER BY` senza prima specificare una tabella da cui selezionare i dati, né è possibile filtrare una tabella tramite l'operazione `WHERE` senza specificare come utilizzarne l'output, ad esempio, tramite `SELECT` o `DELETE`.

Il controllo della struttura assicura che le query rispettino una logica operativa sequenziale appropriata, impedendo che vengano eseguite operazioni che non abbiano un contesto di riferimento valido o che manchino di informazioni essenziali per l'elaborazione. Questa fase è fondamentale per evitare errori comuni e garantire che le query siano costruite in modo da poter essere valutate correttamente nelle fasi successive.

4.3.2 Fase di valutazione

La fase di valutazione rappresenta il nucleo del processo semantico di Neverlang-SQL. In questa fase, la maggior parte delle regole sintattiche del linguaggio vengono analizzate per determinare l'azione esatta che ciascun elemento della query deve eseguire. Durante la valutazione, vengono interpretati i vari componenti della query, come selezioni, filtraggi, ordinamenti e aggregazioni, per determinare come interagiscono tra loro e quali operazioni specifiche devono essere eseguite sul database. Questa fase traduce essenzialmente le istruzioni della query da una rap-

[7]Il riferimento è alla scelta implementativa delle operazioni utilizzate nell'interrogazione del database, che utilizzano, sia come input che come output, un elemento `SelectedData` (Sezione 4.2.4).

presentazione sintattica a un insieme di operazioni concrete che manipolano i dati o le tabelle nel database.

Di seguito viene proposto un esempio per illustrare come le query vengono tradotte in codice Java. La query di partenza utilizzata è:

```
SELECT * FROM Book
WHERE Price > 12 AND Available = TRUE;
```

Inizialmente, le condizioni definite tramite operatori relazionali vengono tradotte in predicati Java, poi aggregati utilizzando l'operatore logico `AND`. Di seguito viene riportata l'implementazione di questi due elementi[8]:

```
module RelationalExpression {
    reference syntax {
        BoolExpr ← Id ">" Value;
    }
    role(evaluation) {
        0 {
            $0.scope = $1.value;
            Predicate<Tuple> relation = tuple -> {
                SQLType value = tuple.get($1.value);
                return value != null && value.compareTo($GT[2].value) > 0;
            };
            $0.relation = relation;
        }
    }
}

module BoolConcatenation {
    reference syntax:
        BoolExpr ← BoolExpr "AND" BoolExpr;
    role(evaluation) {
        0 {
            Predicate<Tuple> expr1 = $1:relation;
            Predicate<Tuple> expr2 = $2:relation;
            Predicate<Tuple> filter = obj -> expr1.test(obj) && expr2.test(obj);
            $0.relation = filter;
        }
    }
}
```

Successivamente, la prima operazione eseguita è `FROM`. Questa clausola consente di ottenere una copia della tabella identificata dall'`Id` fornito dall'utente. Dopo aver ottenuto la tabella iniziale, l'ordine d'esecuzione delle altre clausole presen-

[8]Per brevità, vengono mostrate solo le parti di codice rilevanti per l'esempio. In particolare, come operatore relazionale viene utilizzato `>`, mentre come operatore logico viene utilizzato `AND`.

ti non è predefinito. Nell'esempio analizzato sono presenti le clausole `SELECT` e `WHERE`. La prima viene utilizzata per selezionare gli attributi richiesti dall'utente. In questo caso, significa specificare quali attributi mantenere e quali rimuovere dalla tabella ottenuta come input[9]. L'altra operazione, `WHERE`, consente di filtrare le tuple della tabella utilizzando il predicato costruito dalla valutazione delle condizioni booleane espresse dall'utente. Infine, dopo un numero non definito a priori di manipolazioni dei dati, il risultato ottenuto viene restituito all'utente. Al Listato 4.1 sono mostrate le implementazioni dei vari elementi utilizzati.

Per garantire che la query sia eseguibile correttamente, il progetto implementa anche un controllo sulla struttura del codice fornito. Questo controllo è necessario poiché le regole sintattiche definite nella Sezione 4.2.4, riguardanti le operazioni di manipolazione dei dati, sono poco restrittive e consentono l'utilizzo delle varie clausole SQL senza obblighi precisi riguardo al loro ordine o alla loro combinazione. Ciò potrebbe portare alla scrittura ed esecuzione di query che, pur essendo sintatticamente corrette, risultano semanticamente errate. Un esempio di tale scenario è la query `FROM Book WHERE Available = TRUE`. Questa query rispetta tutte le regole sintattiche, ma non ha un significato logico corretto, poiché non specifica chiaramente come utilizzare la sottotabella ottenuta selezionando le tuple della tabella `Book` con l'attributo booleano `Available` di valore `TRUE`.

Per risolvere questo problema, è stata implementata una fase di controllo della struttura, la cui implementazione è presente al Listato 4.2. In questa fase, l'elemento iniziale `FROM` definisce la query come "non terminale", mentre le operazioni di manipolazione si limitano a copiare tale valore booleano dall'operazione precedente. Quando, in una concatenazione di operazioni, viene utilizzata una clausola terminale, questo parametro viene aggiornato per garantire che la query possa essere riconosciuta come valida. Nell'esempio, l'operazione terminale è rappresentata da `SELECT`, che specifica che il risultato deve essere restituito all'utente.

[9]Nel caso illustrato, l'operazione `SELECT *` richiede la visualizzazione di tutte le colonne della tabella, quindi non viene effettuata alcuna modifica alla tabella ottenuta.

```
module FromEvaluation {
    role(evaluation) {
        0 {
            eval $1;
            if (!$$DatabaseMap.containsKey($1.value)) {
                throw new EntityNotFound(
                    "Cannot find the table \"" + $1.value + "\". "
                    + "Make sure it exists and that its name
                    is spelled correctly."
                );
            }
            $0.table = $$DatabaseMap.get($1.value).copy();
            $0.ref = $1.value;
        }
    }
}

module WhereEvaluation {
    role(evaluation) {
        0 {
            Table table = $1.table;
            table.filterTuple((Predicate<Tuple>) $2.relation);
            $0.table = table;
            $0.ref = $1.ref;
        }
    }
}

module SelectEvaluation {
    role(evaluation) {
        0 {
            eval $1;
            $0.table = $1.table;
            $0.ref = $1.ref;
        }
    }
}

module DataOperationEvaluation {
    role(evaluation) {
        0 {
            System.out.println($1:table.toString());
        }
    }
}
```

Listato 4.1: Implementazione dei ruoli di *evaluation* delle operazioni utilizzate all'interno dell'esempio.

```
module FromStructChecking {
    role(struct-checking) {
        0 {
            $0.isTerminal = false;
        }
    }
}

module WhereStructChecking {
    role(struct-checking) {
        0 {
            $0.isTerminal = $1:isTerminal;
        }
    }
}

module SelectStructChecking {
    role(struct-checking) {
        0 {
            $0.isTerminal = true;
        }
    }
}

module DataOperationStructChecking {
    role(struct-checking) {
        0 {
            if (!(Boolean) $1:isTerminal) {
                throw new SyntaxError(
                    "Invalid SQL statement; "
                    + "expected DELETE, INSERT, SELECT or UPDATE"
                );
            }
        }
    }
}
```

Listato 4.2: Implementazione dei ruoli di *struct-checking* delle operazioni utilizzate all'interno dell'esempio.

4.4 Tecnologie utilizzate

4.4.1 Gradle

Gradle[10] è un sistema di automazione della compilazione del codice, open source, ampiamente utilizzato per progetti in Java, ma supporta anche linguaggi come Scala e JavaScript, ed è lo standard per la programmazione in Kotlin e per lo sviluppo di applicazioni per dispositivi Android. Supporta la gestione delle dipendenze, la compilazione del codice, l'esecuzione di test e la creazione di pacchetti eseguibili. È estremamente configurabile ed estendibile, consentendo di personalizzare il processo di build in base alle specifiche esigenze del singolo progetto.

All'interno di Neverlang-SQL, l'utilizzo di Gradle ha contribuito ad agevolare il processo di compilazione del codice scritto in Neverlang, a velocizzare il testing del linguaggio ottenuto ed ha ricoperto un ruolo fondamentale per la gestione delle dipendenze, in particolare con le librerie di Neverlang.

4.4.2 JUnit5

JUnit[11] è uno dei framework di unit testing più diffusi per il linguaggio di programmazione Java e fornisce un ambiente di test sistematico e automatizzato che aiuta a verificare la correttezza del codice.

In particolare, all'interno del progetto è stato utilizzato JUnit Jupyter per testare se il linguaggio generato con Neverlang funzionasse correttamente. Questo approccio ha consentito di risparmiare una notevole quantità di tempo in quanto, tramite dei test significativi, è stato possibile comprendere rapidamente se le soluzioni implementate fossero corrette oppure no. L'integrazione di JUnit con Gradle ha permesso di eseguire i test in modo continuo e di generare report di test dettagliati, agevolando l'individuazione e la risoluzione dei *bug*.

[10]Documentazione di Gradle: https://docs.gradle.org/current/userguide/userguide.html
[11]Documentazione di JUnit5: https://junit.org/junit5/docs/current/user-guide/

4.4.3 Git

Git[12] è un software per il controllo di versione distribuito. Viene ampiamente utilizzato per la sua efficienza e flessibilità e permette a uno o più sviluppatori di lavorare ad un progetto gestendone multiple versioni e facilitando la collaborazione con terzi.

Durante lo sviluppo del linguaggio, Git (in particolare tramite la piattaforma GitHub) ha permesso di gestire efficacemente le versioni del codice e di tracciare ogni singola modifica apportata al progetto. Git ha dunque permesso di sperimentare nuove funzionalità e di correggere i bug generati durante lo sviluppo senza il rischio di compromettere il codice stabile. Inoltre, sfruttando gli strumenti per la gestione dei branch, è stato possibile sviluppare e testare nuove funzionalità in un ambiente isolato, per poi integrarle nel progetto principale solamente dopo essere state testate.

4.4.4 CheckStyle

CheckStyle[13] è uno strumento di analisi del codice open source, progettato per aiutare gli sviluppatori a scrivere codice Java seguendo le convenzioni di stile. CheckStyle verifica che il codice rispetti un insieme di regole predefinite, come quelle riguardanti la formattazione, la nomenclatura, la gestione delle eccezioni, ecc. Utilizzando questo strumento, è possibile rilevare automaticamente le violazioni dello stile di codice durante il processo di build, riducendo così la probabilità di errori e migliorando la leggibilità del codice.

Nel progetto Neverlang-SQL, CheckStyle è stato utilizzato per garantire che il codice Java (escluso quello generato dal compilatore di Neverlang, nlgc) rispettasse le convenzioni di stile standard. Questo ha contribuito a mantenere un codice pulito e leggibile, migliorando la qualità del codice prodotto.

[12]Documentazione di Git: https://git-scm.com/doc
[13]Documentazione di CheckStyle: https://checkstyle.sourceforge.io

4.4.5 SpotBugs

SpotBugs[14] è uno strumento di analisi del codice per il linguaggio Java che rileva bug e vulnerabilità potenziali. SpotBugs si concentra sull'identificazione di possibili errori nel codice, come dereferenziazioni di nulli, errori logici, problemi di sicurezza e altre pratiche di programmazione sconsigliate che potrebbero causare malfunzionamenti o vulnerabilità del software.

Nel contesto del progetto Neverlang-SQL, SpotBugs è stato utilizzato per esaminare il codice Java associato ai moduli di Neverlang e alle classi Java, e per identificare eventuali difetti o punti deboli nel codice del linguaggio. Questo strumento ha permesso di migliorare la robustezza e la sicurezza del progetto, identificando rapidamente potenziali problemi prima che potessero causare malfunzionamenti nel progetto.

[14]Documentazione di SpotBugs: https://spotbugs.readthedocs.io/en/stable/

Capitolo 5

Validazione e Conclusioni

5.1 Validazione

La validazione del progetto realizzato ha interessato principalmente due tipi di controlli: il test sulla struttura dati creata e il test sull'output fornito. L'obiettivo del controllo è stato quello di determinare se l'implementazione realizzata corrispondesse a ciò che era ragionevolmente atteso che il software facesse.

5.1.1 Test della struttura

Il controllo della base di dati realizzata è stato eseguito sul database creato durante l'esecuzione, costruito usando il file utilizzato come sorgente o lo script fornito al test dall'assioma del linguaggio Neverlang, ossia il simbolo `Program`. In questo modo, nei test implementati nel progetto, è stato sufficiente utilizzare l'attributo `db` del nodo radice dell'AST per verificare che la struttura ottenuta corrispondesse a quella attesa dai comandi oggetto di testing.

Nel Listato 5.1 viene fornito un esempio introduttivo di come i controlli della struttura del database e delle tabelle siano stati realizzati. Il punto fondamentale è quello riportato alla riga 12, che corrisponde all'ottenimento dell'attributo contenente il database creato dal codice utilizzato come input. In modo del tutto analogo, all'interno dei test "strutturali" del progetto, contenuti nella classe `TableTests.java`, sono eseguiti dei codici d'esempio per verificare se l'output ottenuto è quello desiderato.

```
/**
 * Test case to verify that the table is added to the database correctly.
 * @param node the root of the AST
 */
@Test
void testReturnsDB(@NeverlangUnitParam(source = "CREATE TABLE Product (" +
            "   ProductID INT, " +
            "   ProductName VARCHAR(100), " +
            "   Price FLOAT, " +
            "   InStock BOOLEAN " +
            ");") ASTNode node) {
    var db = node.getAttributes().get("db");
    assertInstanceOf(DatabaseMap.class, db);
    assertTrue(((DatabaseMap) db).containsKey("Product"));
}
```

Listato 5.1: Metodo utilizzato per il test del codice prodotto. In particolare, verifica se il nodo radice restituisce un database e se al suo interno contiene la tabella creata.

Per poter eseguire i test descritti, è stato fondamentale l'utilizzo della libreria di Neverlang progettata per interagire con JUnit[1], che è stata aggiunta come dipendenza del progetto tramite Gradle. Grazie all'utilizzo di questa estensione e delle annotazioni `@ExtendWith`, `@NeverlangUnit` e `@NeverlangUnitParam` definite al suo interno, è stato possibile effettuare i test generando in modo automatizzato l'albero sintattico prodotto da Neverlang.

5.1.2 Test dell'output fornito

Il controllo dell'output fornito è stato più complesso, poiché le operazioni di output non modificano la struttura effettiva del database, ma ne creano una temporanea utilizzata per generare l'output secondo le richieste dell'utente. Prendendo come esempio la query di selezione `SELECT Nome, Cognome FROM Customers` e assumendo che la tabella sorgente contenga più attributi rispetto ai due richiesti, il controllo della struttura della tabella non sarebbe sufficiente, in quanto includerebbe anche altri attributi non pertinenti.

Prima implementazione Per risolvere il problema descritto, inizialmente è stato aggiunto nel progetto un attributo al nodo radice chiamato "output", utilizzato per salvare la tabella temporanea generata dalla query di input, con un funziona-

[1]Repository dell'estensione neverlang-junit: https://maven.adapt-lab.di.unimi.it

mento simile a quello descritto nella Sezione 5.1.1 riguardo al test della struttura della tabella. Tuttavia, questo approccio si è rivelato poco flessibile, poiché richiedeva di mantenere un secondo database fittizio, che poteva generare errori se l'utente avesse voluto creare un database con lo stesso nome. Inoltre, obbligava la parte semantica a gestire un aspetto non richiesto dal progetto e introduceva delle problematiche riguardo alla memorizzazione dell'output di più query contemporaneamente.

Seconda implementazione L'implementazione utilizzata nei test sfrutta un metodo presente nell'estensione per JUnit fornita da Neverlang, chiamato `getASTNodeFromResourceFile`. Questo metodo è stato utilizzato per generare l'albero sintattico all'interno del test, anziché ottenere il nodo radice come parametro tramite l'annotazione `@NeverlangUnitParam`. Al Listato 5.2 viene mostrato come l'implementazione del metodo `testOutput` utilizzi il percorso del file contenente il test e la stringa corrispondente all'output atteso, ottenuti come parametri, per verificare che l'output generato corrisponda a quello atteso. In particolare, alla riga 43, tramite la libreria JUnit, viene verificato che il nodo radice dell'albero sintattico esista e non abbia un valore nullo. Per quanto riguarda l'output, questo viene reindirizzato dall'output standard di sistema a un nuovo *output stream* utilizzato dal metodo. Successivamente, l'output catturato viene confrontato con quello atteso ed infine viene ripristinato alla destinazione originale.

```java
/**
 * Get the AST node from a resource file.
 * @param file the file path
 * @return the AST node
 */
private ASTNode getASTNodeFromResourceFile(String file) {
    try {
        File resourceFile = this.getResourceFile(file);
        String testFile = resourceFile.getAbsolutePath();
        String source = FileUtils.fileToString(testFile);
        return DataTests.lang.exec(source, resourceFile);
    } catch (Exception e) {
        Assertions.fail("Loading resource raised Exception " + e);
        return null;
    }
}

/**
 * Get the file from the resources' folder.
 * @param file the file path
 * @return the file
 * @throws URISyntaxException if the URI is invalid
 */
private File getResourceFile(String file) throws URISyntaxException {
    URL url = Thread.currentThread()
            .getContextClassLoader()
            .getResource(file);
    return new File(Objects.requireNonNull(url).toURI());
}

/**
 * Test the output of a file.
 * @param file the file path
 * @param expectedOutput the expected output
 */
private void testOutput(String file, String expectedOutput) {
    PrintStream originalOut = System.out;
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    PrintStream printStream = new PrintStream(outContent, true, StandardCharsets.
        UTF_8);
    System.setOut(printStream);

    try {
        ASTNode node = getASTNodeFromResourceFile(file);
        assertNotNull(node);
        assertEquals(
                expectedOutput + "\n",
                outContent.toString(StandardCharsets.UTF_8)
        );
    } finally {
        System.setOut(originalOut);
    }
}
```

Listato 5.2: Porzione di codice implementata per poter confrontare l'output generato con quello atteso all'interno dei test.

5.2 Conclusioni

In questo lavoro di tesi[2] è stato esplorato l'utilizzo del framework Neverlang per la modellazione di Domain-Specific Languages, principalmente integrando funzionalità tipiche di SQL per la gestione e manipolazione di database. L'approccio modulare offerto da Neverlang ha consentito di costruire un linguaggio altamente flessibile, all'interno del quale le singole funzionalità possono essere aggiunte o modificate in modo incrementale. È stato evidenziato come modularità, riusabilità e flessibilità siano stati elementi chiave per la corretta realizzazione del progetto.

L'obiettivo principale del progetto era creare un linguaggio che permettesse agli utenti di interagire con i database in modo più naturale rispetto all'approccio offerto da linguaggi general-purpose, riducendo il gap di astrazione tra il linguaggio ed il dominio applicativo. Il DSL sviluppato rispecchia questa filosofia ed offre un approccio specifico e ottimizzato per la gestione di tabelle, dati e interrogazioni, mantenendo al contempo una struttura facilmente modificabile.

Attraverso l'analisi e la progettazione proposte in questa tesi, è stato evidenziato come le caratteristiche di Neverlang permettano di creare linguaggi altamente flessibili e adattabili. I test effettuati hanno confermato l'efficacia delle scelte progettuali, dimostrando che il linguaggio implementato non solo soddisfa i requisiti definiti nella parte iniziale dell'elaborato, ma offre anche margini di miglioramento futuri grazie all'implementazione modulare.

Sviluppi futuri Allo stato attuale, il progetto mira a fornire una base solida che racchiuda le caratteristiche principali di SQL. Partendo dal progetto così definito, è possibile operare su di esso effettuando miglioramenti ed aggiungendo nuove *features*. Di seguito vengono riassunti i principali sviluppi futuri:

- **Integrazione di funzionalità avanzate**: Attualmente, il progetto propone un set di operazioni limitato che può essere esteso con l'aggiunta di nuove funzionalità. L'introduzione della clausola `JOIN` tra più tabelle, la gestione degli indici, la possibilità di auto-incrementare i dati, ecc., sono esempi di implementazioni che aumenterebbero le capacità del DSL e ne incrementerebbero l'applicabilità anche in contesti di database più complessi.

[2]Repository del progetto: https://github.com/mircoterenzi/neverlang-sql

- **Implementazione di viste e gestione della coerenza dei dati**: Estendere il DSL per includere il supporto alla creazione e gestione delle viste, consentendo agli utenti di definire tabelle virtuali sui dati, semplificando così le query. Al tempo stesso, è necessario gestire la coerenza dei dati, garantendo che le viste siano sempre aggiornate e coerenti con i dati sottostanti, soprattutto in scenari di modifica od eliminazione di dati.

- **Sviluppo di un'interfaccia grafica**: Enfatizzando l'aspetto didattico del linguaggio sviluppato, un possibile sviluppo potrebbe essere l'implementazione di una Graphical User Interface (GUI) che permetta agli utenti di utilizzare il DSL senza dover scrivere direttamente codice, rendendo il linguaggio accessibile a un pubblico più ampio, inclusi coloro che hanno un'esperienza limitata nel campo della programmazione.

Bibliografia

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[3] Claus Brabrand and Pierre-Etienne Moreau. *Proceedings of the of the Tenth Workshop on Language Descriptions, Tools and Applications (LDTA 2010)*. Association for Computing Machinery, New York, NY, USA, 2010.

[4] Walter Cazzola. Neverlang 2: Language workbench. Ultimo accesso: 23 Agosto 2024.

[5] Walter Cazzola. Domain-specific languages in few steps. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Software Composition*, pages 162–177, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[6] Walter Cazzola and Edoardo Vacchi. Neverlang 2 – componentised language development for the jvm. In Walter Binder, Eric Bodden, and Welf Löwe, editors, *Software Composition*, pages 17–32, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[7] Walter Cazzola, Edoardo Vacchi, et al. Dexter and neverlang: a union towards dynamicity. *Proceedings of ICOOOLPS*, 12, 2012.

[8] Martin Fowler. *Domain-specific languages*. Addison-Wesley Professional, 2010.

[9] Dick Grune and Ceriel JH Jacobs. Parsing techniques (monographs in computer science), 2006.

[10] Paul Hudak. Domain specific languages. *Handbook of programming languages*, 3(39-60):21, 1997.

[11] IBM. What is java? Ultimo accesso: 23 Agosto 2024.

[12] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation*. Wiley-IEEE Computer Society Pr, 2008.

[13] Paul Klint, Tijs van der Storm, and Jurgen Vinju. On the impact of dsl tools on the maintainability of language implementations. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, New York, NY, USA, 2010. Association for Computing Machinery.

[14] Peter Linz and Susan H Rodger. *An introduction to formal languages and automata (7th Edition)*. Jones & Bartlett Learning, 2022.

[15] MetaCase. Eads case study, 2007.

[16] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 419–443, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[17] Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages Systems & Structures*, 43:1–40, 2015.