



**UNIVERSITÀ  
DEGLI STUDI  
DI BERGAMO**

# Quantum IDE: Design e Implementazione di un Quantum Language Compiler

Maatoug Sabrin, mat.1065576

November 8, 2025

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduzione</b>                                       | <b>3</b>  |
| 1.1      | Contesto e motivazione . . . . .                          | 3         |
| 1.2      | Obiettivi . . . . .                                       | 3         |
| <b>2</b> | <b>Filosofia generale di sviluppo</b>                     | <b>3</b>  |
| 2.1      | Scelte tecnologiche . . . . .                             | 4         |
| 2.2      | Processo di Sviluppo . . . . .                            | 4         |
| <b>3</b> | <b>Diagrammi UML</b>                                      | <b>5</b>  |
| 3.1      | Diagramma dei casi d'uso . . . . .                        | 5         |
| 3.2      | Diagramma di deploy . . . . .                             | 6         |
| 3.3      | Diagramma delle classi . . . . .                          | 7         |
| <b>4</b> | <b>Grammatica non decorata</b>                            | <b>8</b>  |
| 4.1      | Regole sintattiche . . . . .                              | 8         |
| 4.2      | Regole lessicali( <code>QuantumLexer.g</code> ) . . . . . | 11        |
| <b>5</b> | <b>Grammatica decorata</b>                                | <b>13</b> |
| <b>6</b> | <b>Controlli semantici</b>                                | <b>15</b> |
| 6.1      | Tabella dei simboli e gestione degli scope . . . . .      | 16        |
| 6.2      | Dichiarazioni e riferimenti . . . . .                     | 16        |
| 6.3      | Controllo dei gate . . . . .                              | 16        |
| 6.4      | Misura e inferenza dei tipi . . . . .                     | 16        |
| 6.5      | Coerenza globale del programma . . . . .                  | 17        |
| 6.6      | Tipologia di errori rilevati . . . . .                    | 17        |
| 6.7      | Inferenza dinamica dei tipi . . . . .                     | 17        |
| <b>7</b> | <b>Verifica e validazione del compilatore</b>             | <b>17</b> |
| 7.1      | Strategia di testing . . . . .                            | 17        |
| 7.2      | File di test . . . . .                                    | 17        |
| 7.3      | Implementazione dei test . . . . .                        | 18        |
| 7.4      | Risultati . . . . .                                       | 18        |
| <b>8</b> | <b>Quantum IDE</b>  | <b>18</b> |
| 8.1      | Architettura generale . . . . .                           | 18        |
| 8.2      | Collegamento front-end e back-end . . . . .               | 19        |
| 8.3      | Integrazione con Qiskit . . . . .                         | 20        |
| <b>9</b> | <b>Conclusioni</b>  | <b>20</b> |
| 9.1      | Sviluppi futuri . . . . .                                 | 20        |

# 1 Introduzione

## 1.1 Contesto e motivazione

Negli ultimi anni, l'informatica quantistica ha suscitato un crescente interesse sia in ambito accademico sia industriale, grazie al suo potenziale di risolvere problemi complessi in tempi significativamente più brevi rispetto ai computer classici. La programmazione quantistica, tuttavia, presenta sfide uniche, dovute alla natura intrinsecamente probabilistica dei qubit e alla necessità di gestire operazioni quantistiche in modo rigoroso e preciso.

La complessità dei linguaggi quantistici e la scarsità di strumenti affidabili rendono fondamentale lo sviluppo di compilatori e parser in grado di analizzare, trasformare e ottimizzare programmi quantistici. Questo progetto nasce proprio con l'intento di fornire un supporto concreto agli sviluppatori, facilitando la scrittura e la verifica dei programmi quantistici.

## 1.2 Obiettivi

Il progetto si propone di realizzare un compilatore e un parser per un linguaggio di programmazione quantistica, seguendo un approccio modulare, scalabile e facilmente estendibile. Gli obiettivi principali sono:

- Creare un sistema robusto per l'analisi sintattica e semantica dei programmi quantistici.
- Fornire una rappresentazione intermedia dei programmi tramite un albero sintattico (AST) per facilitare trasformazioni e ottimizzazioni.
- Garantire leggibilità e manutenibilità del codice attraverso un'architettura modulare e chiara.
- Integrare strategie di testing efficaci per assicurare correttezza e affidabilità.

Questa documentazione illustra le scelte progettuali, l'architettura del compilatore, la struttura del codice e le strategie di verifica adottate, con l'obiettivo di offrire una panoramica completa del progetto, combinando aspetti teorici e pratici della programmazione quantistica.

## 2 Filosofia generale di sviluppo

La progettazione e lo sviluppo del compilatore per il linguaggio quantistico sono stati guidati da alcuni principi fondamentali, volti a garantire robustezza, chiarezza e flessibilità del sistema. La filosofia adottata si basa su tre linee guida principali: modularità, leggibilità e testabilità.

- Modularità: il sistema è stato strutturato in moduli distinti, ciascuno responsabile di un compito specifico, come l'analisi lessicale, il parsing, la costruzione dell'albero sintattico (AST) e l'applicazione dei gate quantistici. Questa separazione consente di isolare le responsabilità, facilitare la manutenzione e permettere l'estensione del compilatore con nuove funzionalità senza impatti significativi sul codice esistente.
- Leggibilità e manutenibilità: particolare attenzione è stata dedicata alla scrittura di codice chiaro e documentato. Nomi di variabili e funzioni descrittivi, commenti esplicativi e una struttura coerente dei moduli rendono il sistema più comprensibile sia agli sviluppatori attuali sia a eventuali futuri collaboratori.

- Testabilità e affidabilità: il progetto integra fin dall'inizio strategie di testing per verificare la correttezza delle varie componenti. Unit test e test di integrazione assicurano che ogni modulo funzioni correttamente da solo e in combinazione con gli altri. Questo approccio incrementa l'affidabilità del compilatore e facilita l'identificazione tempestiva di eventuali errori.

## 2.1 Scelte tecnologiche

Le tecnologie adottate per il progetto sono state selezionate in funzione della loro solidità, della compatibilità con gli obiettivi e della semplicità di integrazione nel flusso di lavoro.

- **ANTLR**: utilizzato per definire il lexer e il parser del linguaggio quantistico. ANTLR ha permesso di generare automaticamente il codice Java necessario per la costruzione dell'albero sintattico e l'implementazione dei controlli semantici.
- **Eclipse**: impiegato per l'implementazione dei controlli semantici e lo sviluppo della logica del compilatore. Eclipse ha fornito un ambiente completo per la scrittura, il debug e l'esecuzione del codice Java generato da ANTLR.
- **JUnit**: utilizzato per il testing completo del progetto, includendo test per il lexer, il parser e la parte semantica. L'integrazione dei test ha garantito la correttezza e l'affidabilità del sistema.
- **Visual Studio Code, Vite e Spring Boot**: utilizzati per lo sviluppo dell'interfaccia grafica e dell'architettura applicativa. Visual Studio Code ha fornito l'ambiente di sviluppo per la realizzazione dei file JSON, CSS e dei componenti del frontend, eseguito tramite Vite, che garantisce un caricamento rapido e un aggiornamento in tempo reale del codice. Il backend è stato sviluppato con Spring Boot, responsabile della logica applicativa e dell'esposizione delle API necessarie al frontend. L'applicazione adotta un'architettura *client-server*, in cui Vite gestisce il frontend e Spring Boot le funzionalità lato server.
- **LaTeX e Overleaf**: impiegati per la stesura della documentazione, garantendo coerenza tipografica e facilità di gestione dei riferimenti interni.
- **Draw.io, StarUML**: utilizzato per creare diagrammi UML ad alto livello che illustrano l'architettura del sistema e le interazioni tra i vari moduli.

Questa combinazione di strumenti ha permesso di concentrare gli sforzi sulla logica del compilatore e sulla qualità del software, mantenendo al contempo una documentazione chiara e aggiornata e un'interfaccia grafica funzionale e facilmente utilizzabile.

## 2.2 Processo di Sviluppo

Il progetto è stato sviluppato seguendo un approccio incrementale e sperimentale, adattato alla natura autonoma del lavoro. Non è stata adottata una metodologia formale standard, come Agile o Waterfall; invece, ogni componente del sistema è stato sviluppato singolarmente e testato in maniera indipendente prima di essere integrato nel complesso.

Il flusso di lavoro seguito prevedeva i seguenti passaggi:

1. **Sviluppo modulare**: ciascun modulo, come il lexer, il parser, i controlli semantici e l'interfaccia grafica, è stato implementato separatamente.
2. **Testing locale**: ogni modulo è stato sottoposto a test unitari con JUnit o test funzionali specifici, per verificarne la correttezza e l'affidabilità.

3. **Integrazione graduale:** dopo la verifica individuale, i moduli sono stati progressivamente integrati, testando l'interazione tra componenti e correggendo eventuali errori.
4. **Verifica complessiva:** al completamento di tutte le integrazioni, sono stati eseguiti test complessivi per assicurare il corretto funzionamento dell'intero sistema.

Questo approccio ha permesso di mantenere il controllo sulla qualità del software, ridurre il rischio di errori durante l'integrazione e garantire la stabilità del progetto pur lavorando in autonomia.

In sintesi, la filosofia generale di sviluppo del progetto si fonda su un equilibrio tra rigore tecnico e flessibilità, mirando a costruire un sistema robusto, estendibile e facilmente manutenibile.

## 3 Diagrammi UML

### 3.1 Diagramma dei casi d'uso

Il diagramma dei casi d'uso rappresenta le principali funzionalità offerte dal sistema e le interazioni tra gli attori e l'applicazione. Nel contesto del progetto, l'attore principale è l'**Utente**, che interagisce con l'applicazione per scrivere, eseguire e gestire il codice quantistico. Le funzionalità sono organizzate in due macro-aree: una relativa all'**editor di codice**, che gestisce le operazioni di scrittura e analisi, e una relativa all'**esecuzione e visualizzazione dei risultati**, che comprende la generazione del circuito e la simulazione.

Le principali operazioni a disposizione dell'utente sono:

- **Scrivere codice** nell'editor integrato.
- **Usufruire dell'autocompletamento** e dell'evidenziazione di errori lessicali e sintattici.
- **Eseguire il codice** e rilevare eventuali errori semantici.
- **Visualizzare il circuito quantistico** generato dal codice.
- **Simulare il circuito** su un simulatore quantistico integrato.
- **Salvare o caricare** file di codice da locale.

Nel diagramma, l'attore **Utente** è connesso ai vari casi d'uso sopra descritti. I casi d'uso relativi all'analisi del codice (*analisi lessicale, sintattica e semantica*) sono a loro volta collegati alla componente del **Compilatore**, mentre quelli relativi alla simulazione e alla visualizzazione del circuito fanno riferimento al **Modulo di esecuzione quantistica**.

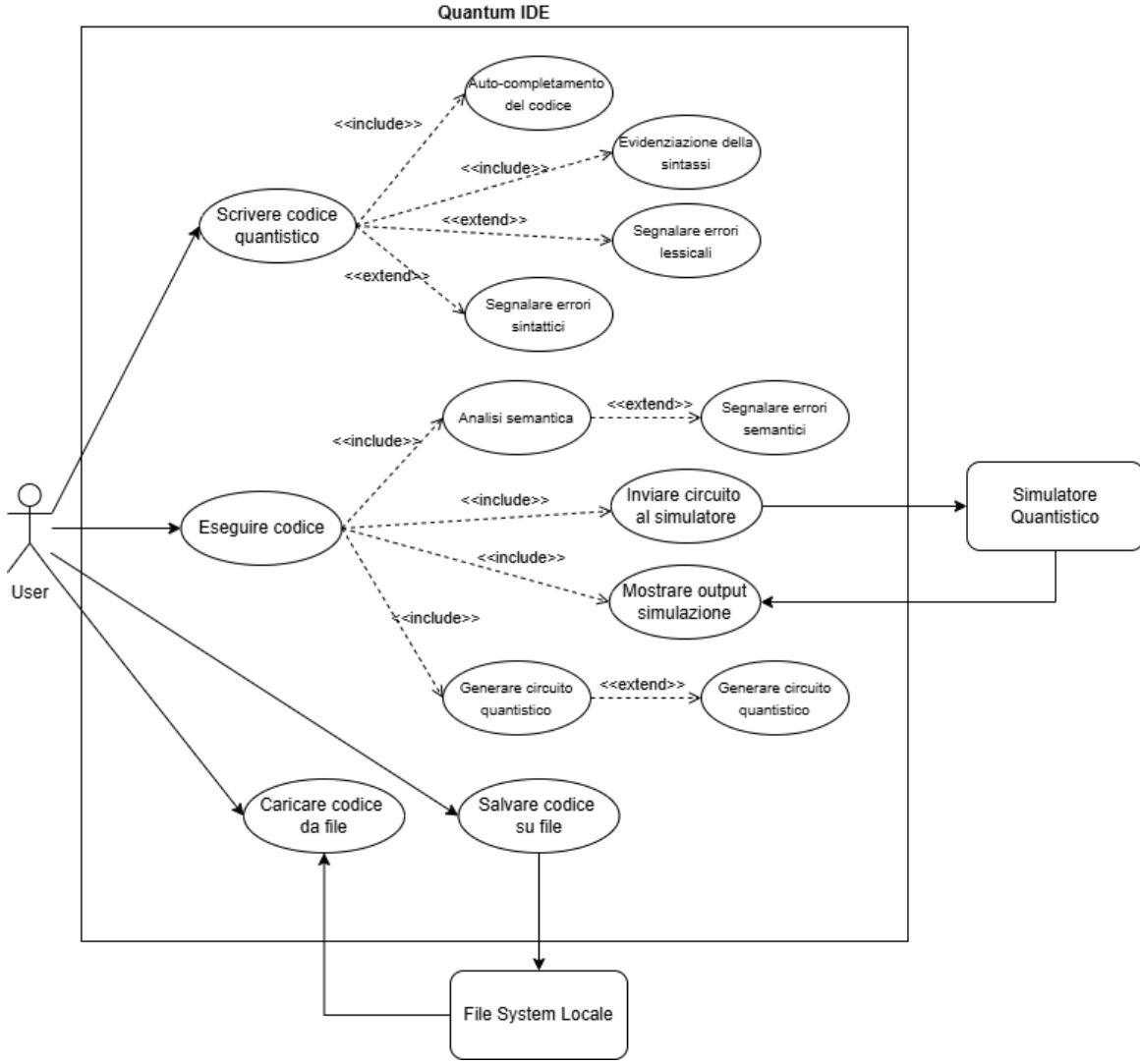


Figure 1: Diagramma dei casi d’uso del sistema

### 3.2 Diagramma di deploy

Il diagramma di deploy illustra la distribuzione fisica e logica dei principali componenti del sistema, basato su un’architettura *client-server*. L’**Utente** interagisce con l’applicazione tramite un’interfaccia web sviluppata con **Vite** e le tecnologie HTML, CSS e JavaScript, che fornisce un editor di codice, la visualizzazione del circuito quantistico e l’area di output.

Il **Server Backend**, realizzato in **Spring Boot**, gestisce la logica applicativa del compilatore: analizza il codice sorgente, genera il circuito quantistico corrispondente e ne avvia l’esecuzione. Il backend comunica con il client tramite API REST e scambio di dati in formato JSON.

Il **Quantum Simulator Node** rappresenta il modulo esterno dedicato all’esecuzione del circuito quantistico, collegato tramite *Local API / Simulation Call*.

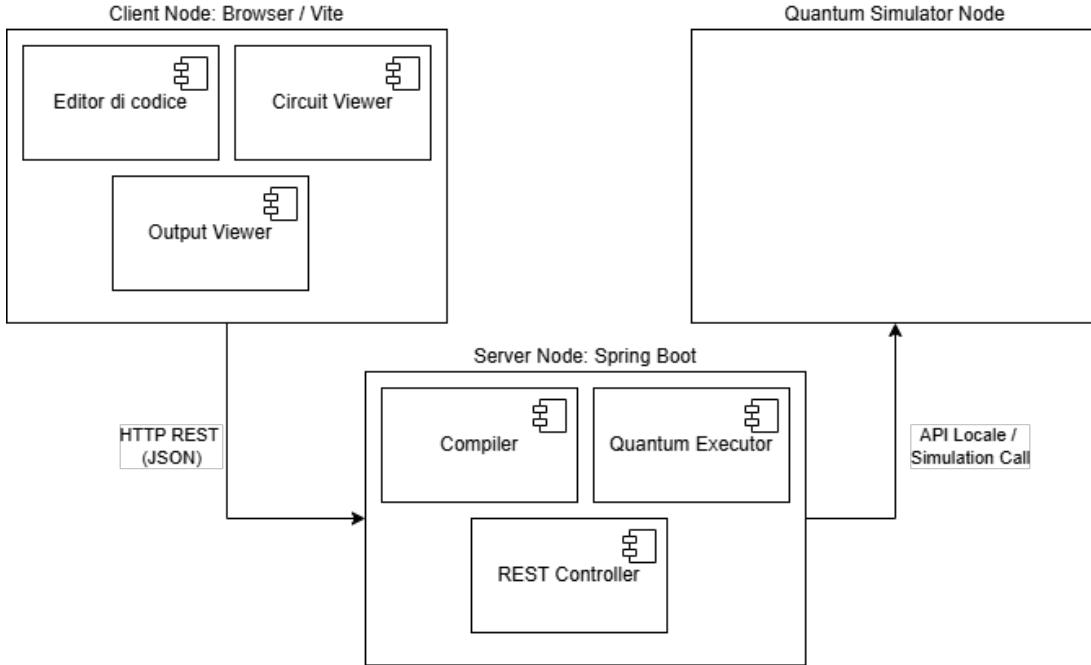


Figure 2: Diagramma di deploy del sistema

### 3.3 Diagramma delle classi

La Figura 3 mostra il diagramma delle classi del progetto, articolato in due package principali: `quantumCompilerPackage` e `testPackage`.

- **quantumCompilerPackage**: contiene le componenti principali del compilatore. In particolare:
  - `QuantumLexer` e `QuantumParser`, classi generate automaticamente da ANTLR a partire dalle rispettive grammatiche lessicale e sintattica. Poiché queste classi contengono numerosi metodi e attributi generati automaticamente, tali dettagli sono stati omessi nel diagramma per una maggiore chiarezza.
  - `QuantumSemanticAnalyzer`, classe implementata manualmente, che si occupa dell’analisi semantica dell’albero sintattico astratto (AST) prodotto dal parser. Include i metodi necessari per il controllo delle dichiarazioni, l’applicazione dei gate e la coerenza del programma quantistico.
- **testPackage**: raccoglie le classi di test utilizzate per verificare il corretto funzionamento dei moduli del compilatore.
  - `printTokens`, che consente di stampare i token generati a partire da un file di input.
  - `ScannerTester`, `ParserTester` e `SemanticTester`, classi JUnit che testano rispettivamente il lexer, il parser e l’analizzatore semantico, verificando che i file corretti non producano errori e che gli errori lessicali, sintattici e semantici vengano rilevati correttamente.

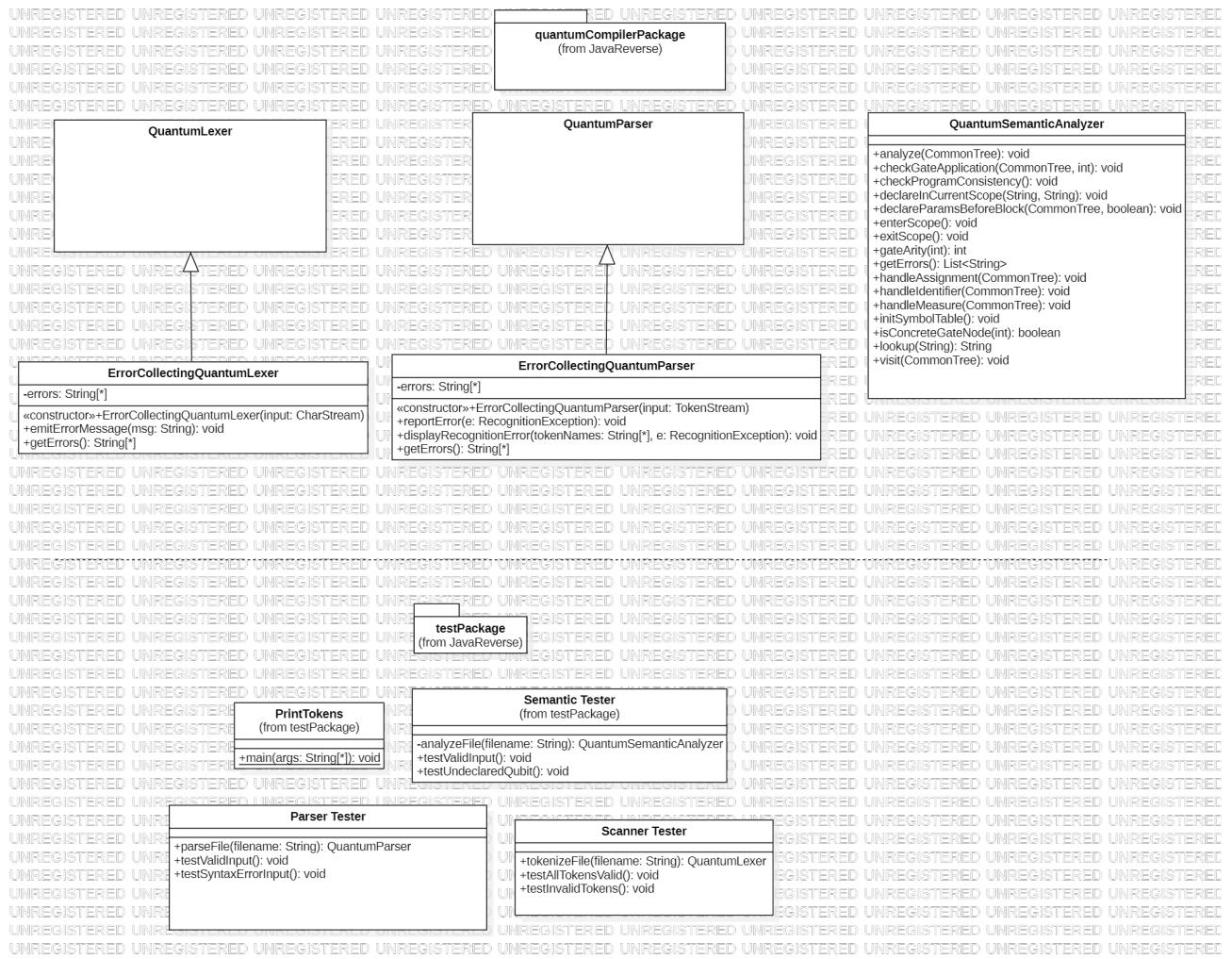


Figure 3: Diagramma delle classi del progetto

## 4 Grammatica non decorata

### 4.1 Regole sintattiche

La seguente grammatica descrive la struttura sintattica del linguaggio progettato. Ogni non terminale rappresenta una categoria sintattica principale del linguaggio, mentre i simboli terminali corrispondono ai token prodotti dal lexer.

$\langle \text{program} \rangle ::= \langle \text{statement} \rangle^* \text{EOF}$

▷ Un programma è costituito da una sequenza di istruzioni, seguita dal simbolo di fine file.

```

 $\langle \text{statement} \rangle ::= \langle \text{qubitDecl} \rangle ;$ 
|  $\langle \text{bitDecl} \rangle ;$ 
|  $\langle \text{gateDecl} \rangle$ 
|  $\langle \text{funcDecl} \rangle$ 
|  $\langle \text{gateApply} \rangle ;$ 
|  $\langle \text{measureStmt} \rangle ;$ 
|  $\langle \text{resetStmt} \rangle ;$ 

```

|  $\langle controlFlow \rangle$   
|  $\langle expr \rangle ;$

▷ Ogni istruzione può essere una dichiarazione, un'applicazione di gate, una misura, un reset, un costrutto di controllo o un'espressione.

---

$\langle qubitDecl \rangle ::= \text{QUBIT } \langle ID \rangle$   
|  $\text{REGISTER } \langle ID \rangle [ \langle INT \rangle ]$

▷ Dichiarazione di un singolo qubit o di un registro quantistico di dimensione specificata.

---

$\langle bitDecl \rangle ::= \text{BIT } \langle ID \rangle$   
|  $\text{REGISTER } \langle ID \rangle [ \langle INT \rangle ]$

▷ Dichiarazione di un bit classico o di un registro di bit.

---

$\langle gateDecl \rangle ::= \text{GATE } \langle ID \rangle ( \langle paramList \rangle ? ) \{ \langle statement \rangle^* \}$

▷ Definizione di una nuova porta quantistica, con parametri opzionali e corpo composto da istruzioni.

---

$\langle funcDecl \rangle ::= \text{FUNC } \langle ID \rangle ( \langle paramList \rangle ? ) \{ \langle statement \rangle^* \}$

▷ Definizione di una funzione classica o quantistica.

---

$\langle gateApply \rangle ::= \langle ID \rangle \langle argList \rangle$

▷ Applicazione di una porta a uno o più argomenti (qubit o registri).

---

$\langle measureStmt \rangle ::= \text{MEASURE } \langle ID \rangle \rightarrow \langle ID \rangle$

▷ Operazione di misura che associa il risultato di un qubit a un bit classico.

---

$\langle resetStmt \rangle ::= \text{RESET } \langle ID \rangle$

▷ Reimposta il valore di un qubit allo stato iniziale.

---

$\langle controlFlow \rangle ::= \langle ifStmt \rangle$   
|  $\langle whileStmt \rangle$

▷ Costrutti di controllo del flusso: condizionali e cicli.

---

$\langle ifStmt \rangle ::= \text{IF} ( \langle expr \rangle ) \{ \langle statement \rangle^* \} (\text{ELSE} \{ \langle statement \rangle^* \ })?$

▷ Istruzione condizionale con blocco opzionale **else**.

---

$\langle \text{whileStmt} \rangle ::= \text{WHILE } (\langle \text{expr} \rangle) \{ \langle \text{statement} \rangle^* \}$

▷ Ciclo che ripete il corpo finché la condizione rimane vera.

---

$\langle \text{expr} \rangle ::= \langle \text{logicalExpr} \rangle$

▷ Punto di ingresso per le espressioni, che possono includere operatori logici, relazionali e aritmetici.

---

$\langle \text{logicalExpr} \rangle ::= \langle \text{relationalExpr} \rangle ((\text{AND}^* | \text{OR}) \langle \text{relationalExpr} \rangle)^*$

▷ Espressioni logiche composte da relazioni collegate da operatori AND e OR.

---

$\langle \text{relationalExpr} \rangle ::= \langle \text{addExpr} \rangle ( < | \leq | > | \geq ) \langle \text{addExpr} \rangle^*$

▷ Espressioni relazionali basate sul confronto tra termini aritmetici.

---

$\langle \text{addExpr} \rangle ::= \langle \text{mulExpr} \rangle ((+ | -) \langle \text{mulExpr} \rangle)^*$

▷ Espressioni aritmetiche additive composte da termini separati da + o -.

---

$\langle \text{mulExpr} \rangle ::= \langle \text{unaryExpr} \rangle ((* | /) \langle \text{unaryExpr} \rangle)^*$

▷ Espressioni moltiplicative composte da fattori separati da \* o /.

---

$\langle \text{unaryExpr} \rangle ::= (! | -)? \langle \text{primaryExpr} \rangle$

▷ Espressioni unarie che possono includere negazione logica o segno negativo.

---

$\langle \text{primaryExpr} \rangle ::= \langle ID \rangle$   
|  $\langle \text{NUMBER} \rangle$   
|  $( \langle \text{expr} \rangle )$

▷ Espressioni primarie costituite da identificatori, numeri o sotto-espressioni racchiuse tra parentesi.

$\langle functionCall \rangle ::= (\text{SIN}$

|  $\text{COS}$

|  $\text{TAN}$

|  $\text{LOG}$

|  $\text{SQRT}$

|  $\text{EXP}) (\langle expr \rangle)$

▷ Chiamata a una funzione matematica predefinita con un singolo argomento numerico.

---

$\langle literal \rangle ::= \langle INT \rangle$

|  $\langle FLOAT \rangle$

|  $0$

|  $1$

|  $\text{true}$

|  $\text{false}$

|  $\text{PI}$

|  $\text{E}$

|  $\langle STRING \rangle$

|  $\langle CHAR \rangle$

▷ Rappresenta un valore costante del linguaggio: numerico, logico, simbolico o testuale.

---

## 4.2 Regole lessicali(QuantumLexer.g)

```
lexer grammar QuantumLexer;

options { language = Java; }

// === Parole chiave ===
QUBIT : 'QUBIT';
REGISTER : 'REGISTER';
MEASURE : 'MEASURE';
RESET : 'RESET';

// === Gate base ===
H : 'H'; X : 'X'; Y : 'Y'; Z : 'Z';
CNOT : 'CNOT'; CX : 'CX'; CY : 'CY'; CZ : 'CZ';
RX : 'RX'; RY : 'RY'; RZ : 'RZ';
S : 'S'; T : 'T';
SWAP : 'SWAP'; ISWAP : 'ISWAP';
CCX : 'CCX'; CSWAP : 'CSWAP';
U1 : 'U1'; U2 : 'U2'; U3 : 'U3';

// === Costanti matematiche ===
PI : 'PI'; EULER : 'E'; SQRT2 : 'SQRT2';

// === Valori binari ===
ZERO : '0'; ONE : '1';

// === Operatori matematici ===
```

```

PLUS : '+'; MINUS : '-'; TIMES : '*'; DIV : '/';
POW : '^'; MOD : '%';

// === Operatori logici e booleani ===
AND : '&&'; OR : '||'; NOT : '!';
EQ : '==' ; NEQ : '!='; LT : '<'; GT : '>';
LE : '<='; GE : '>=';
TRUE : 'true'; FALSE : 'false';

// === Funzioni matematiche ===
SIN : 'sin'; COS : 'cos'; TAN : 'tan';
LOG : 'log'; SQRT : 'sqrt'; EXP : 'exp';

// === Controllo di flusso ===
IF : 'if'; ELSE : 'else'; WHILE : 'while'; FOR : 'for';
RETURN : 'return'; BREAK : 'break'; CONTINUE : 'continue';

// === Tipologie ===
GATE : 'gate'; FUNC : 'func'; BIT : 'BIT';
INTTYPE : 'INT'; FLOATTYPE : 'FLOAT';

// === Simboli ===
SEMI : ';' ; COMMA : ',' ; LPAREN : '(' ; RPAREN : ')' ;
LBRACE : '{' ; RBRACE : '}' ; LBRACK : '[' ; RBRACK : ']' ;
ASSIGN : '=' ; ARROW : '->' ;

// === Identificatori
ID : [a-zA-Z_][a-zA-Z0-9_]* ;

// === Numeri ===
INT : [0-9]+ ;
FLOAT : [0-9]+ ('.' [0-9]+)? ([eE] [+ -]? [0-9]+)?;

// === Spazi ===
WS : [ \t\r\n]+ -> skip ;

// === Commenti ===
COMMENT
: '//' ~('\'n'|'\'r')* '\'r'? '\'n' {$channel=HIDDEN;}
| '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
;

// === Stringhe ===
STRING : '"' (~['\\'] | '\\'.)* '"';
CHAR : '\'' (~['\\'] | '\\'.) '\'';

// === Frammenti ===
fragment EXPONENT : ('e' | 'E') ('+' | '-')? ('0' .. '9')+ ;
fragment HEX_DIGIT : ('0' .. '9' | 'a' .. 'f' | 'A' .. 'F') ;
fragment ESC_SEQ
: '\\' ('b' | 't' | 'n' | 'f' | 'r' | '\"' | '\'' | '\\')
| UNICODE_ESC
;
```

```

| OCTAL_ESC
|
;
fragment OCTAL_ESC
: '\\\\ ('0'..'3') ('0'..'7') ('0'..'7')
| '\\\\ ('0'..'7') ('0'..'7')
| '\\\\ ('0'..'7')
;
fragment UNICODE_ESC
: '\\\\ 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
;

```

## 5 Grammatica decorata

La grammatica decorata del parser `QuantumParser` definisce la struttura sintattica arricchita per la costruzione dell'albero sintattico astratto (AST). I simboli marcati con `^` indicano i nodi promossi nell'AST, mentre i simboli seguiti da `!` sono soppressi, ovvero non appaiono nella struttura finale. Le azioni semantiche, racchiuse tra parentesi graffe, specificano i controlli e le operazioni eseguite durante l'analisi semantica, come la gestione della tabella dei simboli o la verifica dei tipi.

---

```

⟨program⟩ ::= ⟨statement⟩+ EOF { initSymbolTable(); visit(ast);
    checkProgramConsistency(); }

```

▷ Il nodo radice inizializza la tabella dei simboli, visita l'AST e verifica la coerenza del programma.

---

```

⟨statement⟩ ::= ⟨qubitDecl⟩ ;
| ⟨bitDecl⟩ ;
| ⟨gateDecl⟩
| ⟨funcDecl⟩
| ⟨gateApply⟩ ;
| ⟨measureStmt⟩ ;
| ⟨resetStmt⟩ ;
| ⟨controlFlow⟩
| ⟨expr⟩ ;

```

▷ Ogni istruzione produce un nodo AST specifico che viene analizzato tramite `visit()`.

---

```

⟨qubitDecl⟩ ::= QUBIT^ ⟨ID⟩ { declareInCurrentScope(ID, qubit); }
| REGISTER^ ⟨ID⟩ [ ⟨INT⟩ ] { declareInCurrentScope(ID, register); }

```

▷ Dichiara un qubit o un registro nella tabella dei simboli.

---

```

⟨bitDecl⟩ ::= BIT^ ⟨ID⟩ { declareInCurrentScope(ID, bit); }
| REGISTER^ ⟨ID⟩ [ ⟨INT⟩ ] { declareInCurrentScope(ID, register); }

```

▷ Dichiara un bit o un registro classico.

```
 $\langle gateDecl \rangle ::= \text{GATE}^{\wedge} \langle ID \rangle ( \langle paramList \rangle ? ) \langle block \rangle \{ \text{declareInCurrentScope}(ID, \text{gate}); \\ \text{declareParamsBeforeBlock}(node, \text{true}); \}$ 
```

▷ Definisce un nuovo gate e crea uno scope locale per i parametri.

---

```
 $\langle funcDecl \rangle ::= \text{FUNC}^{\wedge} \langle ID \rangle ( \langle paramList \rangle ? ) \langle block \rangle \{ \text{declareInCurrentScope}(ID, \text{func}); \\ \text{declareParamsBeforeBlock}(node, \text{false}); \}$ 
```

▷ Definisce una funzione e registra i parametri nello scope.

---

```
 $\langle gateApply \rangle ::= (\text{H} \\ | \text{X} \\ | \text{Y} \\ | \text{Z} \\ | \text{S} \\ | \text{T}^{\wedge} \langle ID \rangle \\ | (\text{RX} \\ | \text{RY} \\ | \text{RZ})^{\wedge} ( \langle expr \rangle ) \langle ID \rangle \\ | (\text{SWAP} \\ | \text{ISWAP} \\ | \text{CNOT})^{\wedge} \langle ID \rangle , \langle ID \rangle \\ | (\text{CCX} \\ | \text{CSWAP})^{\wedge} \langle ID \rangle , \langle ID \rangle , \langle ID \rangle \\ | \text{U1}^{\wedge} ( \langle expr \rangle ) \langle ID \rangle \\ | \text{U2}^{\wedge} ( \langle expr \rangle , \langle expr \rangle ) \langle ID \rangle \\ | \text{U3}^{\wedge} ( \langle expr \rangle , \langle expr \rangle , \langle expr \rangle ) \langle ID \rangle \{ \text{checkGateApplication}(node, \text{expectedQubits}); \\ \})$ 
```

▷ Controlla il numero e il tipo dei qubit nell'applicazione di un gate.

---

```
 $\langle measureStmt \rangle ::= \text{MEASURE}^{\wedge} \langle ID \rangle \rightarrow \langle ID \rangle \{ \text{handleMeasure}(node); \}$ 
```

▷ Controlla la compatibilità tra qubit e bit nella misura.

---

```
 $\langle resetStmt \rangle ::= \text{RESET}^{\wedge} \langle ID \rangle \{ \text{lookup}(ID); \}$ 
```

▷ Reinizializza il qubit allo stato  $|0\rangle$ .

---

```
 $\langle controlFlow \rangle ::= \langle ifStmt \rangle \\ | \langle whileStmt \rangle \\ | \langle forStmt \rangle \\ | \langle breakStmt \rangle \\ | \langle continueStmt \rangle$ 
```

▷ Strutture di controllo che generano blocchi e nuovi scope.

---

```
 $\langle ifStmt \rangle ::= \text{IF}^\wedge (\langle expr \rangle) \langle block \rangle (\text{ELSE! } \langle block \rangle)? \{ \text{enterScope}(); \text{visitBranches}(node); \text{exitScope}(); \}$ 
```

▷ Gestisce la condizione e gli scope dei blocchi `if/else`.

---

```
 $\langle whileStmt \rangle ::= \text{WHILE}^\wedge (\langle expr \rangle) \langle block \rangle \{ \text{enterScope}(); \text{visit}(node.getChildAt(1)); \text{exitScope}(); \}$ 
```

▷ Apre uno scope per il corpo del ciclo `while`.

---

```
 $\langle forStmt \rangle ::= \text{FOR}^\wedge ((\langle expr \rangle)? \text{SEMI! } \langle expr \rangle? \text{SEMI! } \langle expr \rangle?) \langle block \rangle \{ \text{enterScope}(); \text{visit}(node.getChildAt(3)); \text{exitScope}(); \}$ 
```

▷ Analizza il corpo del ciclo `for` in uno scope locale.

---

```
 $\langle breakStmt \rangle ::= \text{BREAK}\hat{\text{SEMI}}!$ 
```

▷ L'istruzione `break` è gestita sintatticamente ma non richiede azione semantica dedicata.

---

```
 $\langle continueStmt \rangle ::= \text{CONTINUE}\hat{\text{SEMI}}!$ 
```

▷ L'istruzione `continue` non ha controllo semantico esplicito.

---

```
 $\langle block \rangle ::= \text{LBRACE! } \langle statement \rangle^* \text{RBRACE!} \{ \text{enterScope}(); \text{visitChildren}(node); \text{exitScope}(); \}$ 
```

▷ Ogni blocco apre e chiude uno scope semantico.

---

```
 $\langle assignExpr \rangle ::= \langle logicalOrExpr \rangle (=^\wedge \langle assignExpr \rangle)? \{ \text{handleAssignment}(node); \}$ 
```

▷ Gestisce le assegnazioni e la dichiarazione implicita di variabili.

---

```
 $\langle ID \rangle ::= [\text{A-Za-z}_][\text{A-Za-z0-9}_]* \{ \text{handleIdentifier}(node); \}$ 
```

▷ Ogni identificatore viene verificato rispetto alla tabella dei simboli corrente.

---

## 6 Controlli semantici

Il modulo di analisi semantica ha il compito di verificare la correttezza logica e la coerenza del programma quantistico dopo la fase sintattica. L'analizzatore semantico, implementato nella classe `QuantumSemanticAnalyzer`, effettua una visita ricorsiva dell'albero di derivazione (AST) costruito dal parser, mantenendo una struttura a *scope* annidati per gestire correttamente la visibilità dei simboli.

## 6.1 Tabella dei simboli e gestione degli scope

L'analizzatore utilizza una pila di tabelle dei simboli (`Deque<Map<String, String>>`) per rappresentare gli *scope* correnti. All'avvio dell'analisi viene creato lo *scope globale*, che contiene:

- i **gate built-in** (H, X, Y, Z, S, T, RX, RY, RZ, U1, U2, U3, CNOT, SWAP, ISWAP, CCX, CSWAP);
- le **costanti matematiche** PI ed EULER.

Ogni volta che viene incontrato un nuovo blocco sintattico (funzione, gate definito dall'utente o blocco delimitato da parentesi graffe), viene creato un nuovo *scope* tramite il metodo `enterScope()`, che viene successivamente eliminato con `exitScope()`.

## 6.2 Dichiarazioni e riferimenti

Durante la visita dell'AST, l'analizzatore registra ogni nuova dichiarazione di variabile (qubit, bit, registro, funzione o gate) nella tabella dei simboli corrente. Ogni riferimento a un identificatore viene controllato tramite `lookup()`: se il simbolo non è presente in alcuno scope visibile, viene segnalato un errore del tipo:

Variabile non dichiarata: <nome>

In caso di assegnazioni, se il target non è ancora dichiarato, l'analizzatore effettua una *dichiarazione implicita* con tipo iniziale `unknown`, utile per gestire l'inferenza dei tipi.

## 6.3 Controllo dei gate

Per ogni gate quantistico concreto, viene verificata la correttezza del numero e del tipo degli argomenti. Ogni gate possiede un'arità attesa (H è unario, CNOT è binario, CCX è ternario, ecc.), determinata dal metodo `gateArity()`.

L'analizzatore verifica che:

1. il numero di argomenti sia sufficiente;
2. ciascun argomento sia di tipo `qubit` o `register`;
3. eventuali parametri di tipo `unknown` vengano inferiti come `qubit`.

In caso di violazioni, vengono prodotti messaggi come:

Il gate CNOT richiede un qubit, ma <id> è di tipo <tipo>.

## 6.4 Misura e inferenza dei tipi

Le istruzioni di misura vengono analizzate con il metodo `handleMeasure()`, che impone i seguenti vincoli:

- la misura deve avere esattamente due argomenti (`qubit` e `bit`);
- il primo argomento deve essere di tipo `qubit` o `param`;
- il secondo argomento deve essere o dichiarato come `bit` oppure viene inferito come tale.

Se un identificatore non è dichiarato, viene creato dinamicamente con il tipo appropriato (`qubit` o `bit`).

## 6.5 Coerenza globale del programma

Al termine della visita, il metodo `checkProgramConsistency()` verifica la presenza di almeno un qubit nel programma. In caso contrario viene generato un errore:

Il programma non dichiara alcun qubit.

## 6.6 Tipologia di errori rilevati

I principali errori semantici rilevabili sono:

- variabili dichiarate più volte nello stesso scope;
- uso di variabili non dichiarate;
- applicazioni di gate con numero errato di argomenti;
- uso di tipi incompatibili in misura o gate;
- assenza di dichiarazioni quantistiche nel programma.

## 6.7 Inferenza dinamica dei tipi

Un aspetto rilevante dell’analizzatore è la capacità di inferire dinamicamente il tipo di variabili inizialmente sconosciute (`unknown`), sulla base del loro contesto d’uso: per esempio, se una variabile `unknown` viene utilizzata come argomento di un gate, diventa un `qubit` e se viene utilizzata come destinazione di una misura, diventa un `bit`.

Questa strategia permette una maggiore flessibilità nella scrittura del codice, pur mantenendo la coerenza semantica globale del programma.

# 7 Verifica e validazione del compilatore

## 7.1 Strategia di testing

Per verificare il corretto funzionamento dei tre moduli principali del compilatore — analizzatore lessicale, sintattico e semantico — è stata progettata una piccola suite di test basata su file di input rappresentativi. L’obiettivo è stato quello di assicurare che ciascun modulo sia in grado di:

1. riconoscere correttamente un programma valido;
2. segnalare in modo appropriato le varie tipologie di errore (lessicale, sintattico e semantico);
3. garantire la coerenza dei risultati lungo l’intera pipeline di compilazione.

## 7.2 File di test

Sono stati definiti quattro file di input, uno per ciascuna categoria di verifica. Il file `valid_input.file` è stato utilizzato come riferimento per verificare che nessun errore venga segnalato in presenza di un programma corretto, mentre i tre file di errore sono serviti a testare la robustezza dei moduli di analisi.

- `valid_input.file`: programma completo e corretto, che include dichiarazioni di qubit, applicazioni di gate e operazioni di misura. Nessun errore deve essere segnalato durante l’esecuzione dei test.

- `scan_error.file`: contiene caratteri non riconosciuti (@, #, ecc.) o token malformati, per verificare che il lexer identifichi correttamente errori lessicali.
- `syntax_error.file`: include errori grammaticali come dichiarazioni incomplete, parentesi mancanti o simboli errati (`Hadamard q;` oppure `RX(PI q;)`). Serve a testare la capacità del parser di localizzare e segnalare gli errori sintattici.
- `semantic_error.file`: presenta variabili non dichiarate, tipi incompatibili o gate applicati a target errati. È usato per verificare la correttezza dei controlli semanticci implementati.

### 7.3 Implementazione dei test

Ciascun modulo è stato testato indipendentemente tramite classi JUnit 5 dedicate. Ogni test carica i file di input, invoca la fase corrispondente del compilatore e verifica che:

- il file corretto non produca alcun errore;
- il file errato generi il messaggio di errore previsto.

| Classe di test                         | Funzione  |
|--|---|
| <code>LexerTest.java</code>            | Verifica della tokenizzazione e degli errori lessicali        |
| <code>ParserTest.java</code>           | Verifica della costruzione dell'AST e degli errori sintattici |
| <code>SemanticAnalyzerTest.java</code> | Verifica dei controlli di tipo e degli errori semanticci      |

I test sono stati implementati in modo modulare, mantenendo un file di input condiviso (`valid_input.file`) per il caso positivo e file distinti per i casi di errore. Questo approccio garantisce la separazione delle responsabilità e facilita l'individuazione delle cause di fallimento.

### 7.4 Risultati

Tutti i test JUnit hanno restituito esito positivo. Il compilatore è in grado di distinguere correttamente tra programmi validi e programmi contenenti errori di diversa natura. L'approccio modulare adottato consente inoltre di estendere facilmente la suite di test con nuovi costrutti grammaticali o regole semantiche.

## 8 Quantum IDE

L'applicazione **Quantum IDE** costituisce l'ambiente di sviluppo integrato che utilizza il compilatore descritto nella sezione precedente per la generazione e la simulazione di circuiti quantistici. L'obiettivo principale è fornire un'interfaccia interattiva che consenta all'utente di scrivere, validare, visualizzare ed eseguire codice quantistico in modo intuitivo e immediato.

### 8.1 Architettura generale

L'applicazione è basata su un'architettura **client-server**, composta da tre moduli principali:

- un **back-end** sviluppato in Java con Spring Boot, che integra il compilatore costruito con ANTLR;
- un **front-end** realizzato in React e Vite, responsabile della gestione dell'interfaccia grafica e dell'interazione con l'utente;

- un modulo **Python**, che sfrutta la libreria **Qiskit** per la costruzione e la simulazione dei circuiti.

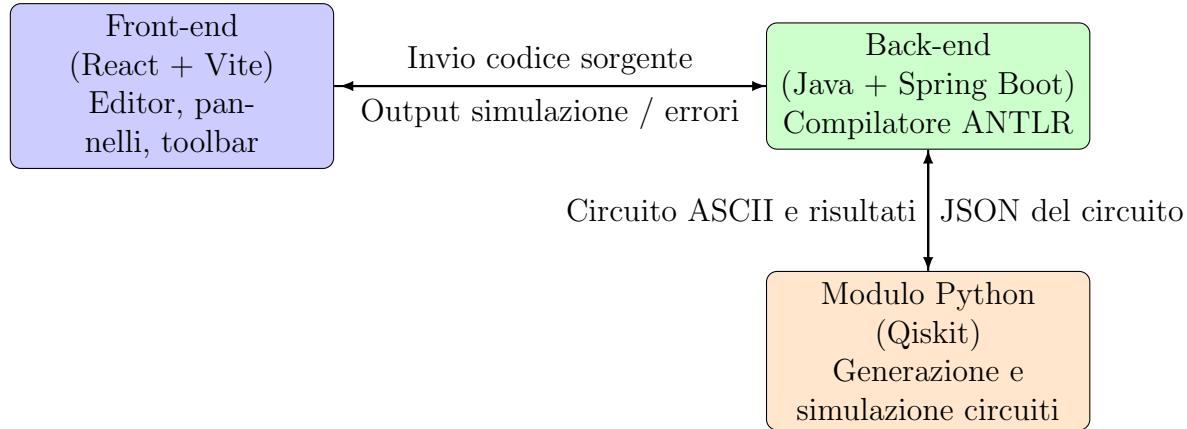


Figure 4: Architettura generale di Quantum IDE e flusso dati tra front-end, back-end e modulo Python.

Il ciclo di elaborazione segue la pipeline illustrata di seguito:

1. l'utente scrive o importa il codice sorgente nel pannello dell'editor;
2. il front-end invia il codice al back-end tramite una richiesta REST;
3. il compilatore analizza il codice e genera una rappresentazione intermedia in formato JSON;
4. il modulo Python converte tale JSON in un oggetto `QuantumCircuit` di Qiskit e ne esegue la simulazione;
5. i risultati (rappresentazione ASCII del circuito e distribuzione delle misure) vengono restituiti al front-end per la visualizzazione.

## 8.2 Collegamento front-end e back-end

La comunicazione tra front-end e back-end avviene tramite **HTTP REST API**. Il front-end invia il codice sorgente in formato **string** a un endpoint dedicato di Spring Boot che:

1. invoca il compilatore ANTLR per l'analisi sintattica e semantica;
2. in caso di successo, produce la descrizione JSON del circuito;
3. restituisce l'output compilato o eventuali errori al client.

Il front-end gestisce la risposta attraverso un sistema di pannelli reattivi:

- **Editor Panel** – area di testo per la scrittura o l'importazione del codice sorgente;
- **Circuit Panel** – visualizza il circuito quantistico in formato ASCII;
- **Simulation Panel** – mostra i risultati della simulazione (conteggi, probabilità e stati finali).

L’interfaccia consente inoltre di aggiornare dinamicamente il codice e visualizzare in tempo reale le modifiche prodotte dal compilatore e dal simulatore. La toolbar superiore integra pulsanti per:

- eseguire la compilazione e la simulazione;
- salvare il codice su file locale (`.qasm` o `.txt`);
- importare codice sorgente preesistente;
- aprire una finestra di **help**, che fornisce le istruzioni d’uso e la sintassi supportata dal linguaggio.

### 8.3 Integrazione con Qiskit

La generazione e la simulazione dei circuiti sono delegate al modulo `quantum_backend.py`, eseguito come processo esterno dal back-end Java. Il modulo riceve il JSON generato dal compilatore e costruisce l’equivalente oggetto `QuantumCircuit` di Qiskit, mappando ogni istruzione sulle corrispondenti porte (es. H, X, CX, RY, ecc.)

Il circuito viene quindi *transpileato* e simulato tramite il backend `AerSimulator`, che produce:

1. la rappresentazione ASCII del circuito;
2. i risultati della misura sotto forma di distribuzione di probabilità.

Il back-end riceve tali dati e li inoltra al front-end per la visualizzazione finale. **Quantum IDE** integra così l’intero flusso di sviluppo quantistico — dalla scrittura del codice alla simulazione — in un ambiente unico, coerente e interattivo.

## 9 Conclusioni

La presente documentazione ha illustrato l’architettura e il funzionamento di **Quantum IDE**, dall’analisi del compilatore basato su ANTLR fino all’integrazione front-end, back-end e modulo Python per la simulazione dei circuiti quantistici.

L’ambiente integrato consente all’utente di scrivere, validare, visualizzare ed eseguire codice quantistico in maniera interattiva, semplificando il processo di sviluppo e simulazione rispetto all’uso diretto di librerie come Qiskit.

### 9.1 Sviluppi futuri

In futuro, Quantum IDE potrebbe essere esteso con:

- supporto per ulteriori linguaggi o dialetti quantistici;
- integrazione con backend hardware reali per eseguire circuiti su computer quantistici;
- funzionalità avanzate di debug e visualizzazione grafica dei circuiti;
- strumenti di collaborazione e versionamento del codice direttamente nell’IDE.