TEJAS Journal of Technologies and Humanitarian Science
ISSN-2583-5599
Vol.05, I.02 (2026)
https://www.tejasjournals.com/
https://doi.org/10.63920/tjths.52015

QCLang: A Quantum Programming Language with Affine Type Enforcement

Abhay Singh[1], Aayushmaan Saksena[2] , Pratham Mohan[3], Dr. Homa Rizvi[4]

[1,2,3]Scholar (B. Tech) Department of Computer Science & Engineering, Shri Ramswaroop Memorial University, Deva Road, Lucknow

[4]Assistant Professor, Department of Computer Science & Engineering, Shri Ramswaroop Memorial University, Deva Road, Lucknow

singhabhay3145@gmail.com, ayushman2908@gmail.com, prathammohan3@gmail.com, homarizvi731@gmail.com

KEYWORDS

Quantum Programming Language, Affine Type System, No-Cloning Theorem, Compiler Design, OpenQASM, Substructural Types, Qubit Resource Management

ABSTRACT

Quantum computing is rapidly transitioning from theoretical research to practical hardware. Yet the software tools used to program quantum computers remain dangerously low-level, offering little to no protection against violations of fundamental physical laws such as the No-Cloning Theorem. This paper presents QCLang, a high-level imperative quantum programming language that enforces quantum mechanical constraints directly at compile time through an affine type system. By treating qubits as affine resources—values that may be used at most once—QCLang statically prevents illegal operations such as qubit cloning and use-after-move, ensuring that any program that successfully compiles is physically valid. We describe the theoretical foundations of affine logic, the language specification, the compiler architecture, and the translation pipeline from QCLang source code to OpenQASM. We evaluate the language against existing quantum programming frameworks and demonstrate its expressiveness through canonical quantum algorithm implementations including Quantum Teleportation and the Deutsch-Jozsa algorithm. QCLang represents a meaningful step toward safer, more developer-friendly quantum software tooling.

I. Introduction

The past decade has witnessed remarkable progress in quantum hardware. Devices from IBM, Google, and IonQ now routinely expose tens to hundreds of physical qubits to researchers and developers. Yet the software ecosystem surrounding these machines has not kept pace. The dominant programming paradigm remains largely circuitlevel: developers construct quantum programs gate-bygate using tools like OpenQASM or Python libraries such as Qiskit, reasoning at a level of abstraction that is closer to assembly language than to modern high-level programming. This low-level approach carries a fundamental problem. Classical programming intuitions actively mislead quantum developers. In classical code, copying a variable is trivially cheap, discarding an unused value is harmless, and reading a variable does not change its state. In quantum computing, none of these assumptions hold. The No-Cloning Theorem [1] forbids the duplication of an arbitrary unknown quantum state. Discarding a qubit without measurement collapses entanglement and destroys coherence in the surrounding system. Measurement is irreversible—once a qubit is observed, its superposition is destroyed permanently.

Current tools provide no compile-time protection
against these violations. A developer can write q2 = q1 in Qiskit and inadvertently copy a Python object reference

Corresponding Author: Abhay Singh, Scholar (B.Tech) Department of Computer Science & Engineering, Shri Ramswaroop Memorial University, Deva Road, Lucknow
Email: singhabhay3145@gmail.com

rather than the quantum state, producing behavior that only fails silently at runtime or yields incorrect results on hardware. Debugging such errors on real quantum devices is expensive, time-consuming, and often physically irreversible.

QCLang addresses this problem at its root by building quantum-physical constraints into the type system of the language itself. Drawing from substructural logic, QCLang classifies qubits as affine resources: values that may be used at most once. The compiler statically tracks every qubit through the program's control flow. If a qubit is used twice—whether by being passed to two functions, assigned to two variables, or applied to both sides of a gate—the program is rejected at compile time with a clear diagnostic. No physically invalid quantum program can reach hardware.

The key contributions of this paper are:

1) A formal language specification for QCLang, including its type system, grammar, and operational semantics for classical and quantum constructs.
2) A complete compiler architecture covering lexing, parsing, affine semantic analysis, and OpenQASM code generation with register allocation.
3) A theoretical grounding in affine logic and its direct correspondence to the physical laws of quantum mechanics.
4) Demonstrations of Quantum Teleportation and the Deutsch-Jozsa algorithm in QCLang, showcasing both safety guarantees and expressive power.

II. BACKGROUND AND RELATED WORK

A. The Quantum Software Landscape

The current generation of quantum programming tools largely falls into two categories: circuit construction libraries and gate-level assembly languages. Qiskit [14] and Cirq are the most widely adopted representatives of the former. They embed quantum circuit construction into Python, benefiting from its ecosystem but inheriting its dynamic type system, which offers no compile-time safety guarantees. OpenQASM [4] serves as a hardware-facing intermediate representation but was never designed as a developer-facing language.

Higher-level efforts have made inroads. Microsoft's Q# provides a statically typed, functional-imperative language with quantum-aware constructs, though its integration with the .NET ecosystem and departure from C-style syntax creates a steep learning curve [5]. Silq, developed at ETH Zurich, introduced automatic uncomputation— the ability to automatically reverse temporary quantum computations to clean up ancilla qubits—using a linearinspired type system [6]. These advances are significant, but neither Silq nor Q# offers the combination of C-style accessibility and strict affine enforcement that QCLang targets.

B. Substructural Type Systems

The theoretical foundation for QCLang lies in substructural logic, specifically the work of Jean-Yves Girard on Linear Logic [3]. Classical type systems permit two structural rules: Weakening (a variable may be declared and never used) and Contraction (a variable may be used multiple times). Linear logic removes both rules, requiring every resource to be used exactly once. Affine logic relaxes this by allowing Weakening while still forbidding Contraction—a resource may be used at most once, but may also be silently discarded [10].

Affine type systems have been successfully deployed in systems programming. Rust's ownership model is affine: values may be moved or dropped but never cloned without explicit instruction [13]. QCLang applies this same discipline to quantum types, where cloning corresponds to a physical impossibility and dropping corresponds to a valid but consequential quantum operation (resetting or tracing out the qubit).

C. The No-Cloning Theorem

The No-Cloning Theorem, independently proven by Wootters & Zurek and by Dieks in 1982, establishes that no unitary operation can produce two identical copies of an arbitrary unknown quantum state [1, 2]. The proof follows directly from the linearity of quantum mechanics.

Suppose a cloning operator U exists such that:
$U(|\psi\rangle \otimes |0\rangle) = |\psi\rangle \otimes |\psi\rangle$
for all states $|\psi\rangle$. For basis states $|0\rangle$ and $|1\rangle$ this is consistent. But for a superposition $|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$, linearity of U forces:
$U(|\psi\rangle \otimes |0\rangle) = \alpha(|0\rangle \otimes |0\rangle) + \beta(|1\rangle \otimes |1\rangle)$
whereas true cloning would require:

$|\psi\rangle \otimes |\psi\rangle = \alpha 2\ |00\rangle + \alpha\beta\ |01\rangle + \beta\alpha\ |10\rangle + \beta 2\ |11\rangle$

These two expressions are equal only when $\alpha\beta = 0$, meaning the state is already a basis state. For a general superposition, cloning is impossible. In a programming language, this theorem implies that the assignment operator cannot function as a copy constructor for quantum types. QCLang enforces this by treating every qubit assignment as a move, invalidating the source variable.

III. THE QCLANG LANGUAGE

A. Design Philosophy

QCLang is designed around one guiding principle: if it compiles, it is physically valid. This requires the language to encode the laws of quantum mechanics into its type system rather than relying on programmer discipline or runtime checks. Alongside this safety requirement, QCLang prioritizes accessibility by adopting a C-style imperative syntax familiar to most programmers. Figure 1 shows the QCLang compiler CLI with all available commands, and Figure 2 shows the interactive REPL mode.

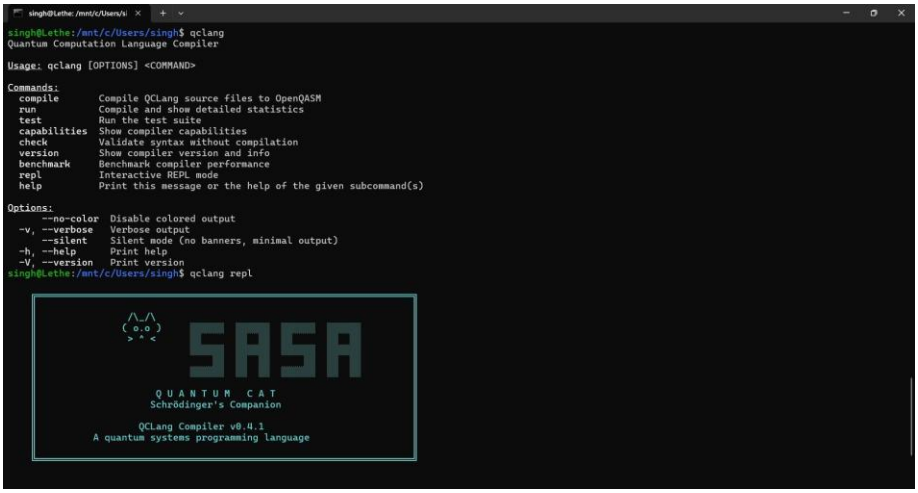

Figure 1: QCLang compiler CLI showing available commands and the REPL startup banner (v0.4.1).

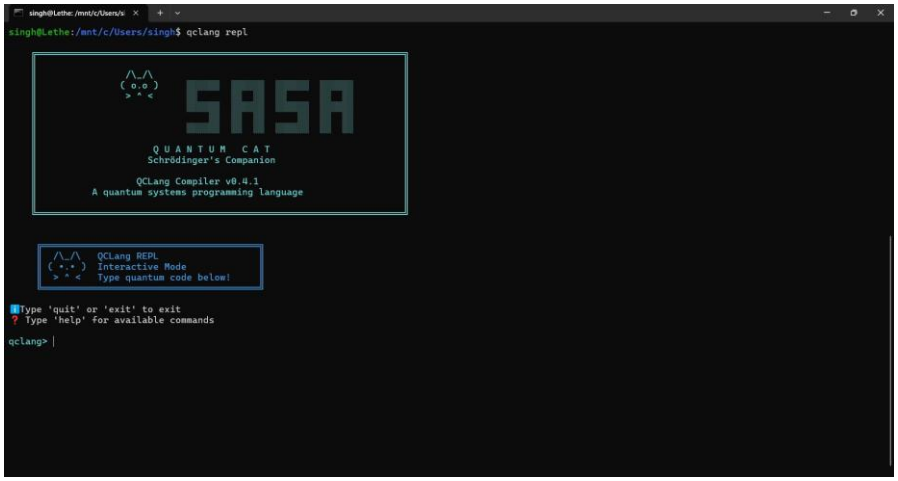

Figure 2: QCLang interactive REPL mode — Schr¨odinger's Companion prompt ready to accept quantum code.

B. Type System

QCLang distinguishes between quantum and classical types at the language level.

Quantum types (affine):

. qubit — A handle to a physical or logical qubit. May be used at most once. Cannot be copied or aliased.

. qubit[N] — A fixed-size array of N qubits. Inherits affine behavior.

https://doi.org/10.63920/tjths.52015

Classical types (unrestricted): . bit — The classical result of a quantum measurement.
Freely copyable.
. int — 32-bit signed integer for loop indices and counts.
. float — 32-bit floating-point number for rotation gate angles.
. bool — Boolean for classical control flow.
The separation between qubit and bit reflects the physical distinction between a quantum state in superposition and a classical measurement outcome. The measure() operation serves as the sole boundary between the two worlds, consuming a qubit and returning a bit.

C. Syntax and Core Constructs

Variable Declaration:

```
qubit q;
int i = 0;
float theta = 3.14159 / 2.0;
```

Quantum Gate Application:

```
hadamard(q);
CX(control, target);
U(theta, phi, lambda, q);
swap(a, b);
```

Measurement:

```
bit b = measure(q);
```

The qubit q is consumed and b holds the classical result. Loop Constructs (compile-time unrolled):

```
for (i = 0; i < 5; i = i + 1) {
    hadamard(q[i]);
}
```

All loops over quantum variables are unrolled at compile time. OpenQASM 2.0 requires a fully static circuit specification, so the compiler evaluates the loop body at compile time and emits a flat gate sequence.

Classical Conditionals:

```
if (b == 1) {
    X(q);
}
```

When the condition is a compile-time constant, the branch is resolved statically. When the condition is a runtime measurement result, the compiler emits a classicallycontrolled quantum operation in QASM.

D. Affine Enforcement: The Semantic Rules
The defining feature of QCLang is its affine semantic checker. The compiler maintains a symbol table mapping each variable to a type, scope, and usage state.

Qubit variables carry one of three states:
. VALID — declared and available for use
. CONSUMED — used in a gate, measured, or moved
. ERROR — used again after being consumed (compilation fails)
State transitions are defined as follows:
Declaration → Valid
Valid + gate/measure/move → Consumed
Consumed + any use → Error

TEJAS Journal of Technologies and Humanitarian Science
ISSN-2583-5599
Vol.05, I.02 (2026)
https://www.tejasjournals.com/
https://doi.org/10.63920/tjths.52015

Example 1 — Illegal Cloning:

```
qubit q1;
hadamard(q1);
qubit q2 = q1; // q1 -> CONSUMED, q2 -> VALID
CX(q1, q2);   // COMPILE ERROR: q1 CONSUMED
```

Example 2 — Self-Control Error:

```
CX(q, q); // COMPILE ERROR: q consumed on
          // first arg, CONSUMED on second
```

Both constructs correspond to physical impossibilities. The first attempts to clone a quantum state in violation of the No-Cloning Theorem. The second attempts a gate where control and target are the same qubit. QCLang rejects both before code generation begins.

E. Borrowing

Many quantum operations transform a qubit without logically consuming it—for example, applying a Hadamard gate and continuing to use the qubit afterward. QCLang supports a borrowing mechanism for this purpose:

```
hadamard(&mut q);
```

While the borrow is active, the original binding q is temporarily suspended. Once the function returns, q resumes Valid status. This aligns naturally with the sequential nature of quantum hardware: a single qubit can only participate in one gate at a time.

IV. COMPILER ARCHITECTURE

The QCLang compiler is structured as a classical multistage pipeline, with the affine checker occupying the semantic analysis stage between parsing and code generation.

A. Frontend: Lexer and Parser

The lexer tokenizes the source stream into a flat sequence of tokens, recognizing keywords (qubit, measure, hadamard), identifiers, numeric literals, and operators. The parser constructs an Abstract Syntax Tree (AST) using a recursive descent strategy.

Key AST node types include:

. VarDecl — variable declaration with type and optional initializer
. QuantumOp — gate application with argument list
. MeasureOp — measurement, consuming a qubit and returning a bit
. IfStmt — conditional with classical or measurementderived condition
. ForLoop — bounded loop with compile-time evaluable bounds
. FuncDef — function definition with typed parameter List

B. Semantic Analysis: The Affine Checker

The affine checker performs a single-pass traversal of the AST, maintaining a scoped symbol table that tracks the type and usage state of every declared variable. For qubit-typed variables, the checker enforces the state machine described in Section III-D. For function calls involving qubit arguments, the checker verifies that each argument is Valid before the call and marks it Consumed afterward (or restores it to Valid for borrow-annotated parameters). Scope boundaries trigger a cleanup pass: any qubit variable in Valid state at the end of its scope is automatically emitted as a reset instruction in the output. This corresponds to the Weakening rule of affine logic—the compiler handles the physical cleanup of unused qubits on behalf of the programmer.

C. Backend: Code Generation and Register Allocation

The backend translates the type-checked AST into OpenQASM 2.0. The primary challenge is register allocation: QCLang allows descriptive variable names (qubit alice, qubit bob) while OpenQASM uses indexed register arrays (qreg q[N]). The register allocator maintains a mapping from logical variable names to physical register indices. When a qubit variable goes out of scope or is consumed, its index becomes available for reuse by subsequently declared qubits. This compact allocation minimizes the number of physical qubits required by the compiled circuit—a critical

optimization for NISQ devices with limited qubit counts and coherence times. Loop unrolling and conditional lowering are performed prior to register allocation. Runtime conditionals (branching on measurement results) are translated to QASM conditional instructions of the form if(creg==1) gate q;.

D. Example Translation

QCLang Source:

```
qubit q;
hadamard(q);
bit b = measure(q);
```

Generated OpenQASM 2.0:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[1];
creg b[1];
h q[0];
measure q[0] -> b[0];
```

Figure 3: shows the compiler output for a teleportation program, including the generated OpenQASM and compilation summary.

E. Implementation Stack

The compiler frontend uses Python with the PLY (Python Lex-Yacc) library for lexing and parsing. The AST and semantic analysis layers are implemented as Python class hierarchies. The backend emits OpenQASM strings compatible with IBM Qiskit and any OpenQASMcompliant simulator. A Rust-based prototype of the affine checker was also developed, exploiting the observation that Rust's own ownership system is itself affine— enabling the host language's borrow checker to assist in verifying the guest language's constraints during development.

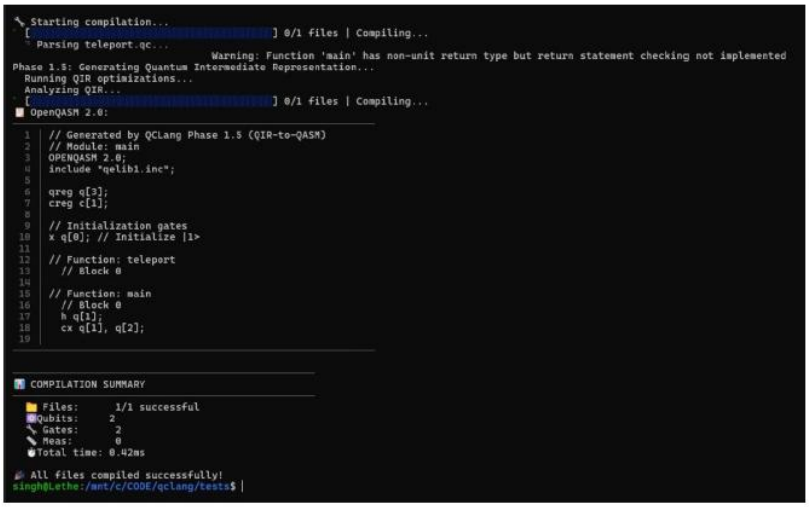

Figure 3: QCLang compiler output for a quantum teleportation program, showing the generated OpenQASM 2.0 and compilation summary (2 qubits, 2 gates, 0.42 ms).

V. ALGORITHMIC DEMONSTRATIONS

A. Quantum Teleportation

Quantum Teleportation transfers an unknown quantum state $|\psi\rangle$ from one party (Alice) to another (Bob) using a shared entangled pair and two classical bits of communication [11]. It exercises all major QCLang features simultaneously: entanglement creation, measurement, and classically-controlled correction gates.

TEJAS Journal of Technologies and Humanitarian Science
ISSN-2583-5599
Vol.05, I.02 (2026)
https://www.tejasjournals.com/
https://doi.org/10.63920/tjths.52015

```
qubit src;
qubit alice_ent;
qubit bob_ent;

hadamard(alice_ent);
CX(alice_ent, bob_ent);

CX(src, alice_ent);
hadamard(src);
bit m1 = measure(src);
bit m2 = measure(alice_ent);

if (m2 == 1) { X(bob_ent); }
if (m1 == 1) { Z(bob_ent); }
```

Once src and alice ent are measured, their qubit handles become Consumed. The type system prevents any subsequent attempt to use them as quantum variables, correctly reflecting that these states have been irreversibly collapsed. The quantum information has been transferred to bob ent; the affine type system encodes this chain of custody explicitly.

B. Deutsch-Jozsa Algorithm

The Deutsch-Jozsa algorithm determines whether a boolean function $f : \{0, 1\}_n \rightarrow \{0, 1\}$ is constant or balanced using a single quantum query, demonstrating exponential speedup over classical deterministic approaches [12].

```
qubit q[n];
qubit ancilla;

X(ancilla);
```

```
hadamard(ancilla);
hadamard(q);

oracle(q, ancilla);

hadamard(q);
bit result[n] = measure(q);
```

The single line hadamard(q) broadcasts the Hadamard gate across the entire qubit array. The compiler expands this into n individual gate applications during compilation, abstracting away index management and eliminating the off-by-one errors that commonly occur in low-level quantum circuit code.

VI. COMPARATIVE ANALYSIS

Table 1 summarizes QCLang's features against the most widely used quantum programming tools.

Table 1: Feature Comparison of Quantum Programming Approaches

Feature	QCLang	Qiskit	Q#	Silq
Paradigm	Imperative	Python DSL	Func/Imp.	Imperative
Type System	Static, Affine	Dynamic	Static	Static, Linear
Compile Safety	Affine Check	None	Partial	Uncompute
No-Clone Enforced	Yes	No	Partial	Yes
Syntax Style	C-style	Python	Novel	Novel
Hardware Target	OpenQASM	Multi	QIR/.NET	Simulators
Auto-Cleanup	Yes (drop)	N/A	No	Yes (uncomp.)

Versus Qiskit: Qiskit is a library embedded in Python, not a language. An assignment like q2 = q1 in Qiskit copies a Python object reference—it does not constitute a quantum move. The error only manifests at runtime when the same qubit is submitted to two gate slots. QCLang catches this entire class of errors before any code is generated or executed.

Versus Q#: Q# provides strong static typing and some compile-time checks, but its enforcement of the No- Cloning constraint is partial. Programmers must still exercise care about how qubits are passed between operations. Q#'s functional style also presents a learning curve for developers from C or Java backgrounds.

Versus Silq: Silq is the closest conceptual relative to QCLang, using a substructural type system to enforce physical laws. The key architectural difference lies in cleanup semantics. Silq performs automatic uncomputation—reversing temporary quantum operations to restore ancilla qubits to $|0\rangle$. QCLang uses implicit reset (drop) for discarded qubits. Neither approach is universally superior; the right choice depends on algorithm structure and hardware target. QCLang's distinctive contribution is the combination of C-style accessibility, strict affine enforcement, and direct OpenQASM emission, making it particularly well-suited as both a teaching tool and a safe abstraction layer over NISQ hardware

VII. CONCLUSION AND FUTURE WORK

This paper has presented QCLang, an affine quantum programming language that enforces the laws of quantum mechanics at compile time. By classifying qubits as affine resources—values that may be used at most once—the language statically prevents the entire class of quantum programming errors arising from illegal state cloning, useafter-move, and double-application. Any program that successfully compiles is guaranteed to be physically valid with respect to the No-Cloning Theorem. The compiler pipeline translates QCLang source code through affine semantic analysis into OpenQASM, handling loop unrolling, register allocation, and classical conditional lowering. Demonstrations on Quantum Teleportation and Deutsch-Jozsa confirm both the safety guarantees and the expressive power of the language. As quantum hardware matures, the cost of debugging incorrect programs on real devices will only increase. Languages like QCLang—which shift correctness guarantees from the programmer to the compiler—are not merely academic exercises. They represent the architectural direction that the quantum software stack must eventually adopt. Future development will focus on:

1) Gate Optimization: Peephole optimization passes to fuse consecutive gates and cancel adjacent inverse pairs (e.g., $H \cdot H = I$), reducing circuit depth for NISQ devices.

2) Simulation Backend: Integrating a lightweight state-vector simulator directly into the compiler toolchain, enabling testing without access to physical quantum hardware.

3) OpenQASM 3.0 Support: Extending the backend to target QASM 3.0, which supports real-time classical computation during quantum execution—essential for quantum error correction.

4) Entanglement Tracking: Exploring type system extensions that track entanglement relationships between qubits, enabling the compiler to warn about operations that would unexpectedly collapse entangled pairs.

5) Algorithm Library: Expanding the standard library to include common quantum subroutines such as the Quantum Fourier Transform, Grover's diffusion operator, and phase estimation as first-class language primitives. QCLang represents a first step toward a future where writing correct quantum programs is no harder—and no more error-prone—than writing correct classical ones.

References

[1] W. K.Wootters and W. H. Zurek, "A single quantum cannot be cloned," Nature, vol. 299, pp. 802–803, 1982.
[2] D. Dieks, "Communication by EPR devices," Physics Letters A, vol. 92, no. 6, pp. 271–272, 1982.
[3] J.-Y. Girard, "Linear logic," Theoretical Computer Science, vol. 50, no. 1, pp. 1–101, 1987.
[4] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," arXiv preprint arXiv:1707.03429, 2017.
[5] K. Svore et al., "Q#: Enabling scalable quantum computing and development with a high-level DSL," in Proc. RWDSL, ACM, 2018.
[6] B. Bichsel, M. Baader, T. Gehr, and M. Vechev, "Silq: A high-level quantum language with safe uncomputation and intuitive semantics," in Proc. 41st ACM SIGPLAN PLDI, pp. 286–300, 2020.
[7] K. Qinami and C. Abbott, "QCLang: A quantum computing language," Undergraduate Research Report, Columbia University, 2018.
[8] J. Preskill, "Quantum computing in the NISQ era and beyond," Quantum, vol. 2, p. 79, 2018.
[9] B. C. Pierce, Types and Programming Languages. Cambridge, MA: MIT Press, 2002.
[10] D. Walker, "Substructural type systems," in Advanced Topics in Types and Programming Languages, MIT Press, 2005, pp. 3–43.

[11] M. A. Nielsen and I. L. Chuang, Quantum Computation and Quantum Information, 10th Ed. Cambridge: Cambridge University Press, 2010.
[12] D. Deutsch and R. Jozsa, "Rapid solution of problems by quantum computation," Proc. Royal Society of London A, vol. 439, pp. 553–558, 1992.
[13] N. D. Matsakis and F. S. Klock, "The Rust programming language," ACM SIGADA Ada Letters, vol. 34, no. 3, pp. 103–104, 2014.
[14] IBM Quantum, "Qiskit: An open-source framework for quantum computing," 2023. [Online]. Available: https://qiskit.org
[15] N. Akhtar, S. Rabbani, H. Rabbani, Saurav Kumar, Y. Perwej, "AI-Driven Intelligent Resume Recommendation Engine", International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET), Print ISSN: 2395-1990, Online ISSN: 2394-4099, Volume 12, No. 3, Pages 1141–1155, June 2025, DOI: 10.32628/IJSRSET2512145.
[16] Prof. Kameswara Rao Poranki, Y. Perwej, Nikhat Akhtar," Integration of SCM and ERP for Competitive Advantage", TIJ's Research Journal of Science & IT Management – RJSITM, International Journal's-Research Journal of Science & IT Management of Singapore, ISSN:2251-1563, Singapore, in www.theinternationaljournal.org as RJSSM, Volume 04, Number 05, Pages 17-24, 2015
[17] Y. Perwej, "An Evaluation of Deep Learning Miniature Concerning in Soft Computing", International Journal of Advanced Research in Computer and Communication Engineering (IJARCCE), ISSN (Online): 2278-1021, ISSN (Print): 2319-5940, Volume 4, Issue 2,Pages 10 - 16, February 2015, DOI: 10.17148/IJARCCE.2015.4203
[18] Y. Perwej, Firoj Parwej, "A Neuroplasticity (Brain Plasticity) Approach to Use in Artificial Neural Network", International Journal of Scientific & Engineering Research (IJSER), France, ISSN 2229 – 5518, Volume 3, Issue 6, Pages 1- 9, 2012, DOI: 10.13140/2.1.1693.2808
[19] Y. Perwej, "The Bidirectional Long-Short-Term Memory Neural Network based Word Retrieval for Arabic Documents", Transactions on Machine Learning and Artificial Intelligence (TMLAI), which is published by Society for Science and Education, United Kingdom (UK), ISSN 2054-7390, Volume 3, Issue 1, Pages 16 - 27, 2015, DOI: 10.14738/tmlai.31.863
[20] Y. Perwej, "Unsupervised Feature Learning for Text Pattern Analysis with Emotional Data Collection: A Novel System for Big Data Analytics", IEEE International Conference on Advanced computing Technologies & Applications (ICACTA'22), SCOPUS, IEEE No: #54488 ISBN No Xplore: 978-1-6654-9515-8, Coimbatore, India, 2022, DOI: 10.1109/ICACTA54488.2022.9753501