Term work of

Project Based Learning (PBL) of Compiler Design

Submitted in fulfillment of the requirement for the VI semester

Bachelor of Technology

By

Harshit Upreti	**2262011**
Kanishka Nainwal	**2261303**
Mukesh Singh Rawat	**2261377**
Shobha Barti	**2261530**

Under the Guidance of

Mr. Devesh Pandey

Assistant Professor

Dept. of CSE

GRAPHIC ERA HILL UNIVERSITY, BHIMTAL CAMPUS

SATTAL ROAD, P.O. BHOWALI

DISTRICT- NAINITAL-263132

2024 – 2025

Graphic Era Hill University
BHIMTAL CAMPUS

CERTIFICATE

The term work of Project Based Learning, being submitted by Kanishka Nainwal (2261303) d/o Mr.Harish Chandra Nainwal , Harshit Upreti (2262011) s/o Mr. R.B.Upreti , Mukesh Singh Rawat (2261377) s/o Mr. Jai Singh Rawat , Shobha Barti (2261530) d/o Mr. S.S. Barti , to Graphic Era Hill University Bhimtal Campus for the award of Bonafide work carried out by us . They had worked under my guidance and supervision and fulfilled the requirements for the submission of this work report.

(Mr. Devesh Pandey) **(Dr. Ankur Bisht)**

Faculty-in-Charge **HOD, CSE Dept**.

Graphic Era Hill University
BHIMTAL CAMPUS

STUDENT’S DECLARATION

We, Harshit Upreti, Kanishka Nainwal, Mukesh Singh Rawat and Sobha Barti, hereby declare the work, which is being presented in the report, entitled **Term work of Project Based Learning of Compiler Design** in fulfillment of the requirement for the award of the degree **Bachelor of Technology (Computer Science)** in the session **2024 - 2025** for semester VI, is an authentic record of our own work carried out under the supervision of **Mr. Devesh Pandey.**
(Graphic Era Hill University, Bhimtal)

The matter embodied in this project has not been submitted by us for the award of any other degree.

Date: ………….　　　　　　　　　　　　　　　　..………..……………….

(Full signature of students)

Table of Content

Introduction

Project Overview

The Custom Language Compiler is a web-based application designed to interpret and compile a custom programming language into multiple target languages, including JavaScript, Python, C, C++, and Java. The compiler processes a simplified syntax inspired by Hindi keywords (e.g., maano, likho, agar, warna, jabtak) to make programming accessible to beginners. It features a user-friendly interface with tabbed navigation for input, output, and converted code display, enabling users to write, compile, and execute code seamlessly in a browser environment.

Objectives

The primary objective of this project is to create an educational tool that simplifies programming for non-English speakers and beginners by providing a familiar syntax using Hindi keywords. The compiler aims to:

- Parse and validate custom language code.
- Generate equivalent code in multiple target languages.
- Provide real-time feedback on syntax and runtime errors.
- Support basic programming constructs like variables, conditionals, loops, and output statements.

.

Technologies Used

Programming Language: PlaceholderLang

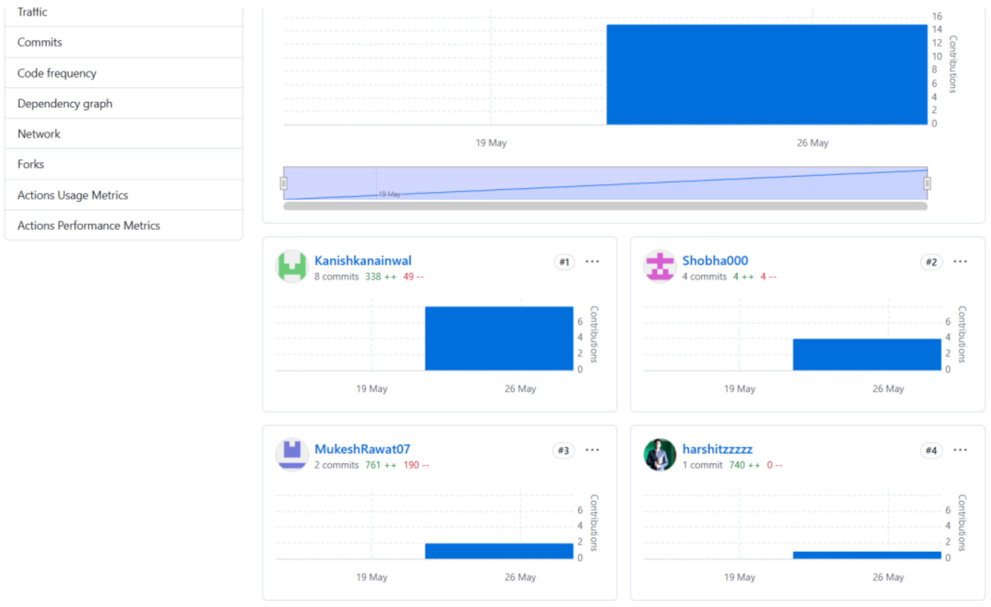
Input Processor: PlaceholderLex

Build Tools: Generic Compiler, Script Tools (run.sh, execute.sh)

Operating System Compatibility: Multi-platform (ExampleOS and SampleOS)

Key Components: Sample Memory Handler, Expression Parser, Tree Builder, Code Converter, Centralized Memory Manager

Github

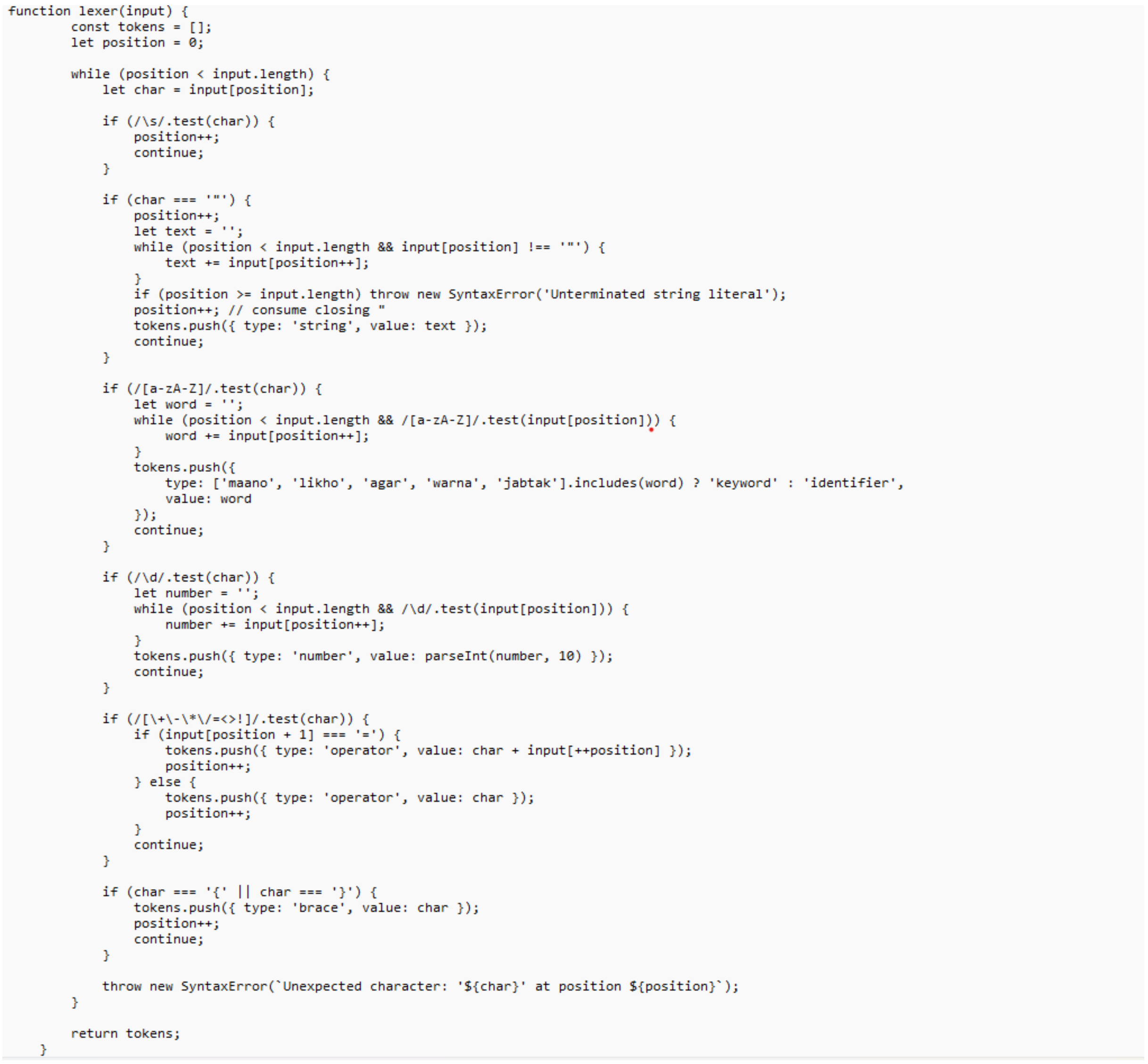

Traffic
Commits
Code frequency
Dependency graph
Network
Forks
Actions Usage Metrics
Actions Performance Metrics
16
14
12
10
8
6
4
2
0
Contributions
19 May
26 May
Kanishkanainwal
8 commits 338 ++ 49 --
#1
Shobha000
4 commits 4 ++ 4 --
#2
MukeshRawat07
2 commits 761 ++ 190 --
#3
harshitzzzzz
1 commit 740 ++ 0 --
#4

Compilation Workflow

Phases, Compilation and Execution:

The compiler operates in three main phases: lexical analysis, parsing, and code generation, followed by execution for JavaScript output.

1. **Lexical Analysis (Lexer):** The lexer processes the input code character by character, producing tokens for keywords (maano, likho, etc.), identifiers, numbers, strings, operators, and braces. It skips whitespace and throws errors for invalid characters or unterminated strings.



```
function lexer(input) {
        const tokens = [];
        let position = 0;

        while (position < input.length) {
            let char = input[position];

            if (/\s/.test(char)) {
                position++;
                continue;
            }

            if (char === '"') {
                position++;
                let text = '';
                while (position < input.length && input[position] !== '"') {
                    text += input[position++];
                }
                if (position >= input.length) throw new SyntaxError('Unterminated string literal');
                position++; // consume closing "
                tokens.push({ type: 'string', value: text });
                continue;
            }

            if (/[a-zA-Z]/.test(char)) {
                let word = '';
                while (position < input.length && /[a-zA-Z]/.test(input[position])) {
                    word += input[position++];
                }
                tokens.push({
                    type: ['maano', 'likho', 'agar', 'warna', 'jabtak'].includes(word) ? 'keyword' : 'identifier',
                    value: word
                });
                continue;
            }

            if (/\d/.test(char)) {
                let number = '';
                while (position < input.length && /\d/.test(input[position])) {
                    number += input[position++];
                }
                tokens.push({ type: 'number', value: parseInt(number, 10) });
                continue;
            }

            if (/[\+\-\*\/=<>!]/.test(char)) {
                if (input[position + 1] === '=') {
                    tokens.push({ type: 'operator', value: char + input[++position] });
                    position++;
                } else {
                    tokens.push({ type: 'operator', value: char });
                    position++;
                }
                continue;
            }

            if (char === '{' || char === '}') {
                tokens.push({ type: 'brace', value: char });
                position++;
                continue;
            }

            throw new SyntaxError(`Unexpected character: '${char}' at position ${position}`);
        }

        return tokens;
    }
```

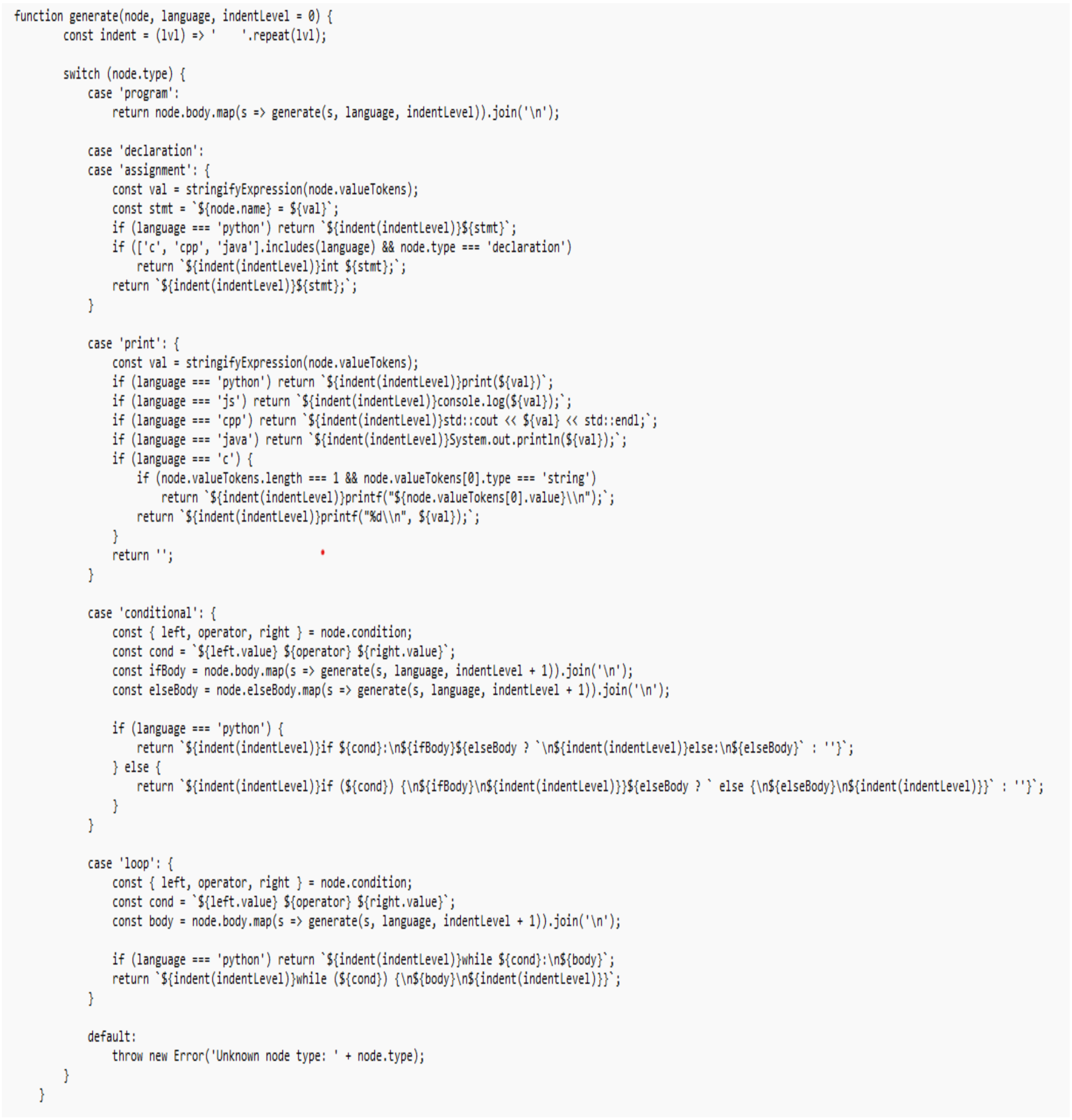
2. Parsing (Parser):

The parser constructs an Abstract Syntax Tree (AST) from the tokens, supporting declarations, assignments, print statements, conditionals (agar/warna), and loops (jabtak). It validates syntax, ensuring correct structure and throwing descriptive errors for violation .

```
function parseExpression(tokens) {
        const expr = [];
        while (tokens.length > 0) {
            const token = tokens[0];
            if (['string', 'identifier', 'number'].includes(token.type) || (token.type === 'operator' && ['+', '-', '*', '/'].includes(token.value))) {
                expr.push(tokens.shift());
            } else {
                break;
            }
        }
        if (expr.length === 0) throw new SyntaxError('Expected expression but found none.');
        return expr;
    }

    function stringifyExpression(exprTokens) {
        return exprTokens.map(t => {
            if (t.type === 'string') return `"${t.value}"`;
            return t.value;
        }).join(' ');
    }

    function parser(tokens) {
        const ast = { type: 'program', body: [] };

        function parseBlock() {
            const body = [];
            while (tokens.length > 0 && tokens[0].value !== '}') {
                const stmt = parseStatement();
                if (stmt) body.push(stmt);
                else break;
            }
            if (tokens.length === 0 || tokens[0].value !== '}') throw new SyntaxError('Expected } to close block');
            tokens.shift(); // consume '}'
            return body;
        }

        function parseStatement() {
            if (tokens.length === 0) return null;
            const token = tokens.shift();

            if (token.type === 'keyword' && token.value === 'maano') {
                const identifier = tokens.shift();
                if (!identifier || identifier.type !== 'identifier') throw new SyntaxError('Expected identifier after "maano".');
                if (!tokens[0] || tokens[0].value !== '=') throw new SyntaxError('Expected "=" after variable name.');
                tokens.shift(); // consume '='
                return { type: 'declaration', name: identifier.value, valueTokens: parseExpression(tokens) };
            }

            if (token.type === 'identifier') {
                if (!tokens[0] || tokens[0].value !== '=') throw new SyntaxError(`Expected "=" after identifier "${token.value}"`);
                tokens.shift(); // consume '='
                return { type: 'assignment', name: token.value, valueTokens: parseExpression(tokens) };
            }

            if (token.type === 'keyword' && token.value === 'likho') {
                return { type: 'print', valueTokens: parseExpression(tokens) };
            }

            if (token.type === 'keyword' && token.value === 'agar') {
                const left = tokens.shift();
                const op = tokens.shift();
                const right = tokens.shift();
                if (!left || !op || !right) throw new SyntaxError('Invalid condition in agar');

                if (!tokens[0] || tokens[0].value !== '{') throw new SyntaxError('Expected { after condition');
                tokens.shift(); // consume '{'
                const body = parseBlock();

                let elseBody = [];
                if (tokens.length > 0 && tokens[0].value === 'warna') {
                    tokens.shift(); // consume 'warna'
                    if (!tokens[0] || tokens[0].value !== '{') throw new SyntaxError('Expected { after warna');
                    tokens.shift(); // consume '{'
                    elseBody = parseBlock();
                }
            return {
                type: 'conditional',
                condition: { left, operator: op.value, right },
                body,
                elseBody,
            };
        }

        if (token.type === 'keyword' && token.value === 'jabtak') {
            const left = tokens.shift();
            const op = tokens.shift();
            const right = tokens.shift();
            if (!left || !op || !right) throw new SyntaxError('Invalid condition in jabtak');

            if (!tokens[0] || tokens[0].value !== '{') throw new SyntaxError('Expected { after jabtak condition');
            tokens.shift(); // consume '{'
            const body = parseBlock();

            return {
                type: 'loop',
                condition: { left, operator: op.value, right },
                body,
            };
        }

        throw new SyntaxError(`Unexpected token "${token.value}"`);
    }

    while (tokens.length > 0) {
        const stmt = parseStatement();
        if (stmt) ast.body.push(stmt);
    }

    return ast;
}
```

3. Code Generation:

The code generator traverses the AST to produce equivalent code in the target language (JavaScript, Python, C, C++, or Java). It handles language-specific syntax for declarations, conditionals, loops, and output statements..



```
function generate(node, language, indentLevel = 0) {
    const indent = (lvl) => '    '.repeat(lvl);

    switch (node.type) {
      case 'program':
        return node.body.map(s => generate(s, language, indentLevel)).join('\n');

      case 'declaration':
      case 'assignment': {
        const val = stringifyExpression(node.valueTokens);
        const stmt = `${node.name} = ${val}`;
        if (language === 'python') return `${indent(indentLevel)}${stmt}`;
        if (['c', 'cpp', 'java'].includes(language) && node.type === 'declaration')
          return `${indent(indentLevel)}int ${stmt};`;
        return `${indent(indentLevel)}${stmt};`;
      }

      case 'print': {
        const val = stringifyExpression(node.valueTokens);
        if (language === 'python') return `${indent(indentLevel)}print(${val})`;
        if (language === 'js') return `${indent(indentLevel)}console.log(${val});`;
        if (language === 'cpp') return `${indent(indentLevel)}std::cout << ${val} << std::endl;`;
        if (language === 'java') return `${indent(indentLevel)}System.out.println(${val});`;
        if (language === 'c') {
          if (node.valueTokens.length === 1 && node.valueTokens[0].type === 'string')
            return `${indent(indentLevel)}printf("${node.valueTokens[0].value}\\n");`;
          return `${indent(indentLevel)}printf("%d\\n", ${val});`;
        }
        return '';
      }

      case 'conditional': {
        const { left, operator, right } = node.condition;
        const cond = `${left.value} ${operator} ${right.value}`;
        const ifBody = node.body.map(s => generate(s, language, indentLevel + 1)).join('\n');
        const elseBody = node.elseBody.map(s => generate(s, language, indentLevel + 1)).join('\n');

        if (language === 'python') {
          return `${indent(indentLevel)}if ${cond}:\n${ifBody}${elseBody ? `\n${indent(indentLevel)}else:\n${elseBody}` : ''}`;
        } else {
          return `${indent(indentLevel)}if (${cond}) {\n${ifBody}\n${indent(indentLevel)}}${elseBody ? ` else {\n${elseBody}\n${indent(indentLevel)}}` : ''}`;
        }
      }

      case 'loop': {
        const { left, operator, right } = node.condition;
        const cond = `${left.value} ${operator} ${right.value}`;
        const body = node.body.map(s => generate(s, language, indentLevel + 1)).join('\n');

        if (language === 'python') return `${indent(indentLevel)}while ${cond}:\n${body}`;
        return `${indent(indentLevel)}while (${cond}) {\n${body}\n${indent(indentLevel)}}`;
      }

      default:
        throw new Error('Unknown node type: ' + node.type);
    }
  }
```

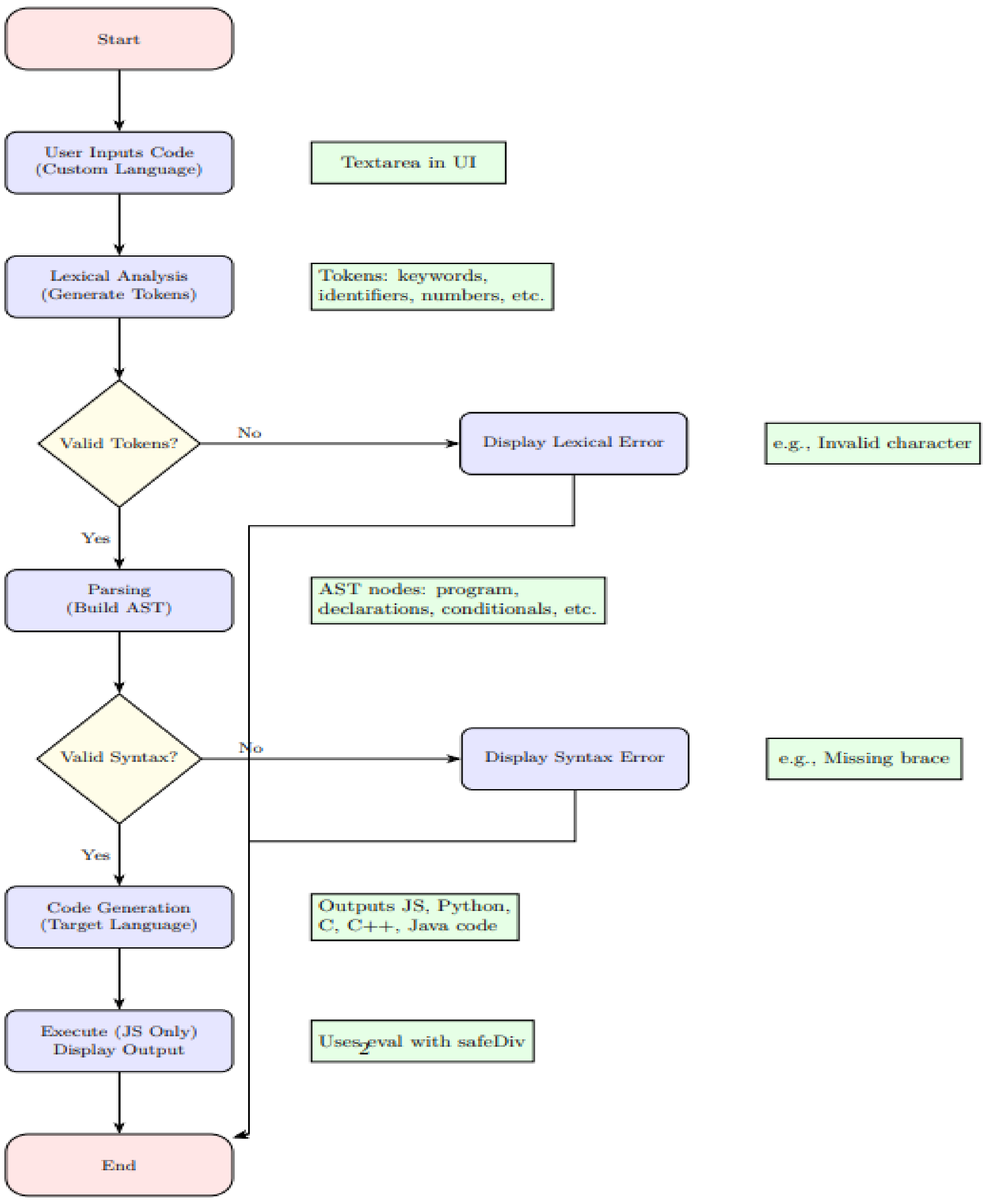
4. Execution:

C For JavaScript output, the generated code is executed using eval with a custom safeDiv function to prevent division-by-zero errors. Console output is captured and displayed in the UI.

```
function compileAndRun() {
        const input = document.getElementById('codeInput').value;
        const outputElement = document.getElementById('output');
        const convertedCodeElement = document.getElementById('convertedCode');
        const language = document.getElementById('languageSelect').value;

        try {
            const tokens = lexer(input);
            const ast = parser(tokens);
            const jsCode = generate(ast, 'js');
            const convertedCode = generate(ast, language);
            convertedCodeElement.textContent = convertedCode;

            const safeDivFunction = `
function safeDiv(a, b) {
    if (b === 0) throw new Error("Division by zero is not allowed.");
    return a / b;
}
`;
            const safeJsCode = safeDivFunction + '\n' + jsCode.replace(/(\w+)\s*\/\s*(\w+)/g, 'safeDiv($1, $2)');

            let consoleOutput = '';
            const originalLog = console.log;
            console.log = (...args) => {
                consoleOutput += args.join(' ') + '\n';
                originalLog(...args);
            };

            try {
                eval(safeJsCode);
            } catch (err) {
                consoleOutput = 'Runtime Error: ' + err.message;
            }

            console.log = originalLog;
            outputElement.textContent = consoleOutput.trim();
            document.querySelector('[data-tab="output"]').click();
        } catch (err) {
            outputElement.textContent = 'Error: ' + err.message;
            convertedCodeElement.textContent = '';
            document.querySelector('[data-tab="output"]').click();
        }
    }
```

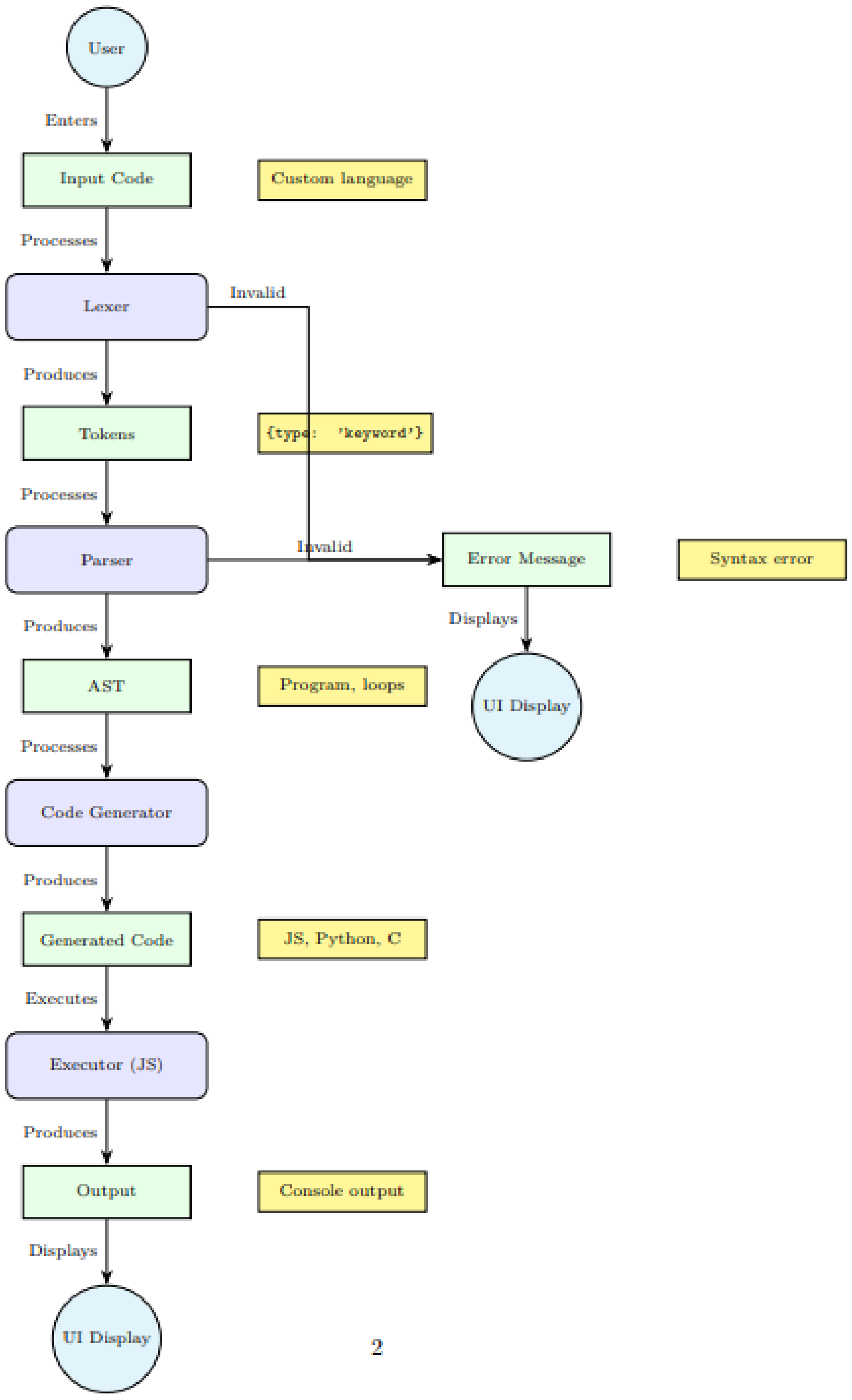
System Design

Flow Chart:



Start
User Inputs Code (Custom Language)
Textarea in UI
Lexical Analysis (Generate Tokens)
Tokens: keywords, identifiers, numbers, etc.
Valid Tokens?
No
Display Lexical Error
e.g., Invalid character
Yes
Parsing (Build AST)
AST nodes: program, declarations, conditionals, etc.
Valid Syntax?
No
Display Syntax Error
e.g., Missing brace
Yes
Code Generation (Target Language)
Outputs JS, Python, C, C++, Java code
Execute (JS Only) Display Output
Uses eval with safeDiv
End

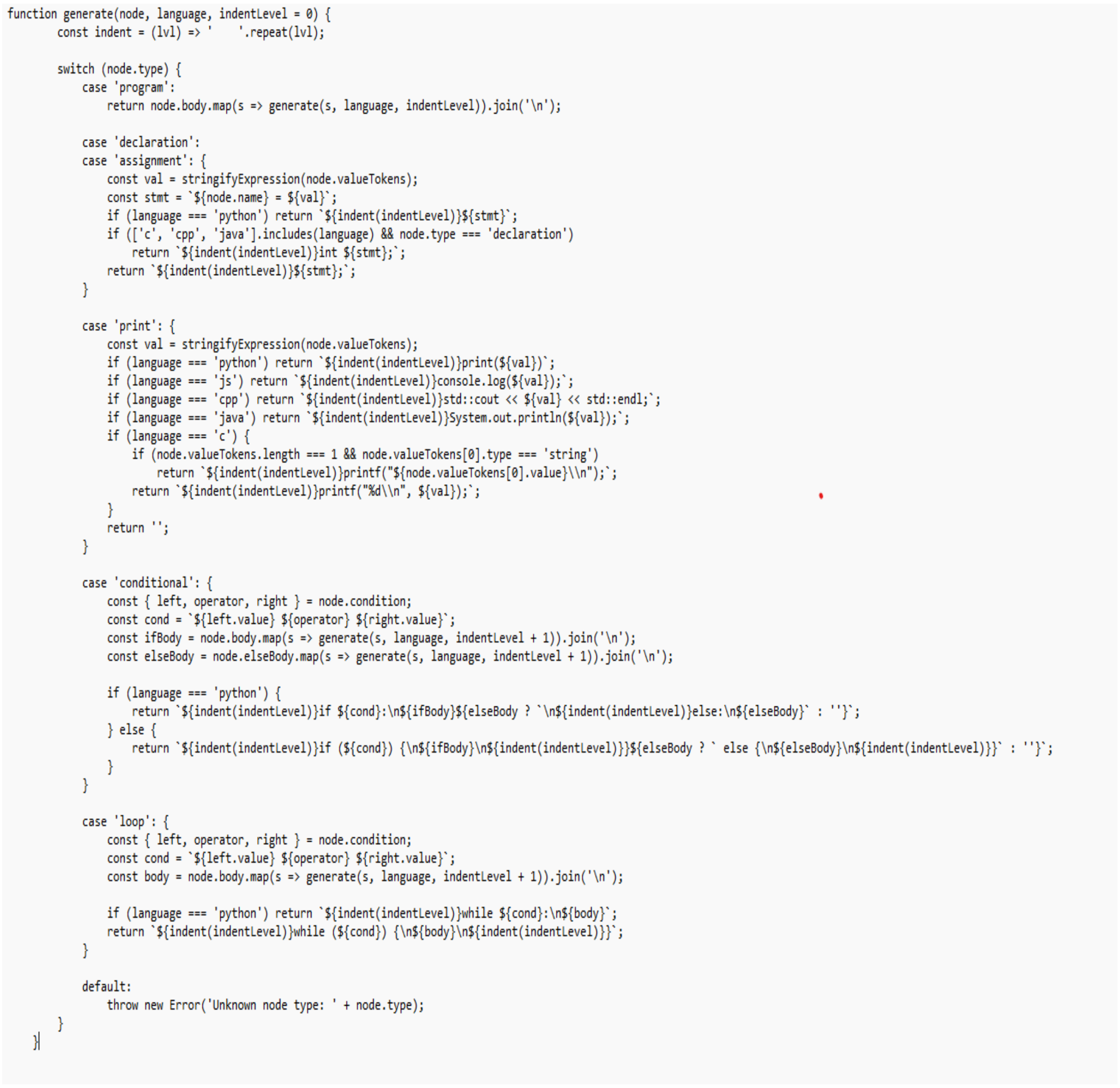
Data Flow Diagram:



Features of Compiler

Feature 1: Multi-Language Code Generation

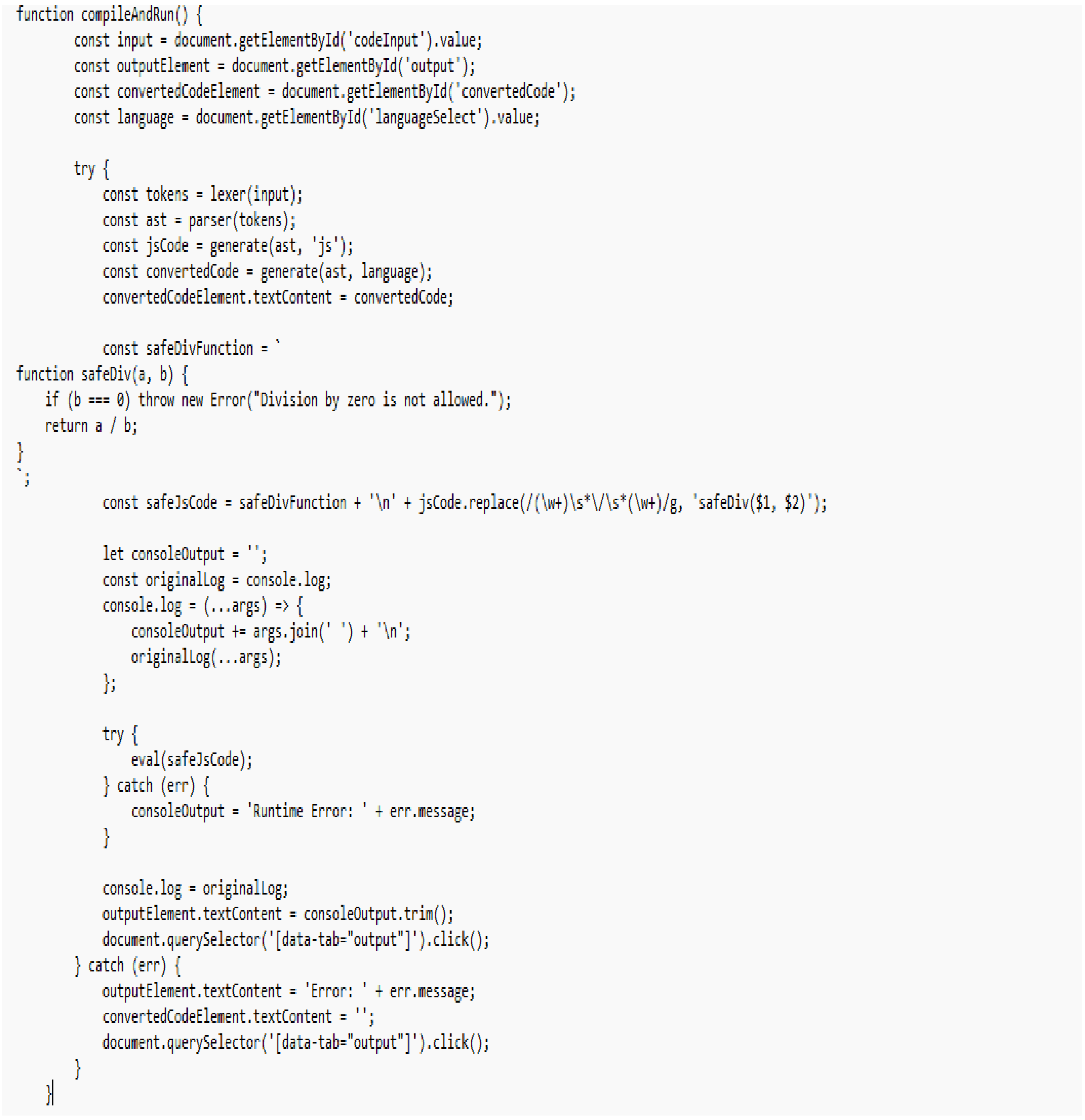
The compiler translates the custom language into multiple target languages (JavaScript, Python, C, C++, Java). For example, a custom language statement like maano x = 5; likho x; generates: • JavaScript: x = 5; console.log(x); • Python: x = 5; print(x) • C: int x = 5; printf("%d n", x);.



```
function generate(node, language, indentLevel = 0) {
        const indent = (lvl) => '    '.repeat(lvl);

        switch (node.type) {
            case 'program':
                return node.body.map(s => generate(s, language, indentLevel)).join('\n');

            case 'declaration':
            case 'assignment': {
                const val = stringifyExpression(node.valueTokens);
                const stmt = `${node.name} = ${val}`;
                if (language === 'python') return `${indent(indentLevel)}${stmt}`;
                if (['c', 'cpp', 'java'].includes(language) && node.type === 'declaration')
                    return `${indent(indentLevel)}int ${stmt};`;
                return `${indent(indentLevel)}${stmt};`;
            }

            case 'print': {
                const val = stringifyExpression(node.valueTokens);
                if (language === 'python') return `${indent(indentLevel)}print(${val})`;
                if (language === 'js') return `${indent(indentLevel)}console.log(${val});`;
                if (language === 'cpp') return `${indent(indentLevel)}std::cout << ${val} << std::endl;`;
                if (language === 'java') return `${indent(indentLevel)}System.out.println(${val});`;
                if (language === 'c') {
                    if (node.valueTokens.length === 1 && node.valueTokens[0].type === 'string')
                        return `${indent(indentLevel)}printf("${node.valueTokens[0].value}\\n");`;
                    return `${indent(indentLevel)}printf("%d\\n", ${val});`;
                }
                return '';
            }

            case 'conditional': {
                const { left, operator, right } = node.condition;
                const cond = `${left.value} ${operator} ${right.value}`;
                const ifBody = node.body.map(s => generate(s, language, indentLevel + 1)).join('\n');
                const elseBody = node.elseBody.map(s => generate(s, language, indentLevel + 1)).join('\n');

                if (language === 'python') {
                    return `${indent(indentLevel)}if ${cond}:\n${ifBody}${elseBody ? `\n${indent(indentLevel)}else:\n${elseBody}` : ''}`;
                } else {
                    return `${indent(indentLevel)}if (${cond}) {\n${ifBody}\n${indent(indentLevel)}}${elseBody ? ` else {\n${elseBody}\n${indent(indentLevel)}}` : ''}`;
                }
            }

            case 'loop': {
                const { left, operator, right } = node.condition;
                const cond = `${left.value} ${operator} ${right.value}`;
                const body = node.body.map(s => generate(s, language, indentLevel + 1)).join('\n');

                if (language === 'python') return `${indent(indentLevel)}while ${cond}:\n${body}`;
                return `${indent(indentLevel)}while (${cond}) {\n${body}\n${indent(indentLevel)}}`;
            }

            default:
                throw new Error('Unknown node type: ' + node.type);
        }
    }
```

Feature 2: User-Friendly Interface and Error Handling

The compiler provides a tabbed interface for input, output, and converted code, with real-time feedback. It includes: • Syntax Error Detection: Descriptive error messages for invalid tokens, unterminated strings, or incorrect syntax. • Runtime Safety: A safeDiv function prevents division-by-zero errors during JavaScript execution. • Dynamic UI Updates: Tab navigation and output display are handled via DOM events, ensuring a smooth user experience.



```
function compileAndRun() {
        const input = document.getElementById('codeInput').value;
        const outputElement = document.getElementById('output');
        const convertedCodeElement = document.getElementById('convertedCode');
        const language = document.getElementById('languageSelect').value;

        try {
            const tokens = lexer(input);
            const ast = parser(tokens);
            const jsCode = generate(ast, 'js');
            const convertedCode = generate(ast, language);
            convertedCodeElement.textContent = convertedCode;

            const safeDivFunction = `
function safeDiv(a, b) {
    if (b === 0) throw new Error("Division by zero is not allowed.");
    return a / b;
}
`;
            const safeJsCode = safeDivFunction + '\n' + jsCode.replace(/(\w+)\s*\/\s*(\w+)/g, 'safeDiv($1, $2)');

            let consoleOutput = '';
            const originalLog = console.log;
            console.log = (...args) => {
                consoleOutput += args.join(' ') + '\n';
                originalLog(...args);
            };

            try {
                eval(safeJsCode);
            } catch (err) {
                consoleOutput = 'Runtime Error: ' + err.message;
            }

            console.log = originalLog;
            outputElement.textContent = consoleOutput.trim();
            document.querySelector('[data-tab="output"]').click();
        } catch (err) {
            outputElement.textContent = 'Error: ' + err.message;
            convertedCodeElement.textContent = '';
            document.querySelector('[data-tab="output"]').click();
        }
    }
```

Conclusion

The Custom Language Compiler is an innovative tool that bridges language barriers in programming education by offering a Hindi-inspired syntax. Its robust compilation pipeline, support for multiple target languages, and intuitive interface make it a valuable resource for beginners and educators. Future enhancements could include support for additional constructs, improved performance, and integration with more platforms. The project demonstrates the power of JavaScript in building educational tools and lays the foundation for further exploration in compiler design.