MANUAL FILE OF NATURAL LANGUAGE TO CODE COMPILER

CONCEPT NOTE

Writing code from high-level instructions or descriptions can take a lot of time and might lead to mistakes.The goal of this project is to develop a tool known as a Natural English to Code Compiler that can automatically convert plain English or pseudocode input into functional C++ code.

The natural language to code compiler will convert plain text into a functional C++ code by following a structured approach.
First, it will read the user's input and break it into parts, like instructions for declaring variables or creating loops. Then, it will organize these parts into a tree-like structure called an Abstract Syntax Tree (AST), where each part of the program (like a variable or a loop) is a separate branch. This tree helps the compiler understand the input. Finally, the compiler will use the tree to generate C++ code.

A functional C++ program created from plain language instructions is the Natural Language to Code Compiler's output. The compiler understands the user's request, for instance, if they say, "Find the square of a number," generates a structure (AST), and then generates C++ code that computes the square of the input number and shows the outcome. This makes coding easier by automatically converting natural language into useful code.

Natural Language to Code Compiler - User Manual

Introduction

The **Natural Language to Code Compiler** is an intuitive tool that converts human-readable instructions into functional C++ code. This tool is designed to assist developers by automatically generating code based on natural language inputs, making programming more accessible and efficient.

Features

- Converts natural language commands into C++ code
- Supports various programming constructs such as loops, conditionals, functions, and classes
- Displays parse tree and abstract syntax tree (AST) structure
- Provides a manual upload option for user reference
- Offers an interactive UI with real-time code generation

User Interface Overview

The UI of the NL2C Compiler consists of:

- **Input Panel (Left Side):** Users enter natural language commands to describe their desired code.
- **Output Panel (Right Side):** Displays parsed tree structures, Abstract Syntax Tree (AST), and the generated C++ code.
- **Compile Button:** Processes the input and converts it into executable C++ code.
- **Manual Button:** Provides access to documentation for reference.

Integrated Functions

The compiler integrates multiple functional layers to ensure accurate code conversion.

1. Input Handling and Preprocessing

- `getUserInput()`: Captures the user's high-level instruction.
- `tokenizeInput(const std::string &input)`: Splits input into tokens for analysis.

- `normalizeInput(const std::string &input)`: Converts input to a standard format (e.g., lowercase, removing extra spaces).
- `detectLanguageIntent(const std::string &input)`: Determines the intent (e.g., "create loop", "define variable").
- `validateSyntax(const std::string &input)`: Checks if the input is grammatically correct.

2. Parsing Functions

- `parseVariables(const std::string &input)`: Extracts variable names and types.
- `parseLoops(const std::string &input)`: Identifies and parses loop structures.
- `parseConditionals(const std::string &input)`: Parses "if", "else", or "switch" statements.
- `parseFunctions(const std::string &input)`: Extracts function definitions or calls.
- `parseOperators(const std::string &input)`: Identifies operators (e.g., arithmetic, logical).

3. Mapping to C++ Constructs

- `mapToCppVariable(const std::string &input)`: Maps variable instructions to C++ syntax.
- `mapToCppLoop(const std::string &input)`: Maps loop instructions to for, while, or do-while.
- `mapToCppConditional(const std::string &input)`: Maps conditionals to if or switch syntax.
- `mapToCppFunction(const std::string &input)`: Converts instructions into C++ function prototypes or definitions.
- `mapToCppLibrary(const std::string &input)`: Suggests or includes necessary C++ libraries.

4. Code Generation

- `generateCppVariable(const std::string &type, const std::string &name, const std::string &value)`: Creates C++ variable code.

- `generateCppLoop(const std::string &type, const std::string &condition)`: Creates a loop based on type and condition.
- `generateCppConditional(const std::string &condition, const std::string &body)`: Creates an if or switch block.
- `generateCppFunction(const std::string &returnType, const std::string &name, const std::string &params, const std::string &body)`: Generates a C++ function.
- `generateCppHeader()`: Adds #include directives automatically.

5. Compiler Backend

- `compileCode(const std::string &code)`: Compiles the generated C++ code.
- `executeCode()`: Runs the compiled code and captures output.
- `saveCodeToFile(const std::string &fileName, const std::string &code)`: Saves the generated code to a .cpp file.
- `loadCodeFromFile(const std::string &fileName)`: Loads code for further processing.
- `formatCode(const std::string &code)`: Formats the generated code for readability.

6. Debugging and Error Handling

- `checkCompilationErrors(const std::string &output)`: Analyzes compiler error messages.
- `handleUnknownIntent(const std::string &input)`: Provides feedback for unsupported instructions.
- `suggestCorrections(const std::string &input)`: Suggests possible corrections for invalid input.
- `logError(const std::string &error)`: Logs errors for debugging.
- `logUserActivity(const std::string &activity)`: Tracks user interactions for analytics.

7. Utility Functions

- `getType(const std::string &input)`: Determines the data type from user input (e.g., int, float).

- `getCondition(const std::string &input)`: Extracts the condition for loops or if statements.
- `getFunctionSignature(const std::string &input)`: Extracts function return type and parameters.
- `generateIndentation(int level)`: Adds proper indentation to the generated code.
- `getLibrariesNeeded(const std::string &code)`: Lists libraries needed for the generated code.

8. Predefined Patterns

- `createHelloWorldProgram()`: Generates a simple "Hello, World!" program.
- `createBasicMathProgram()`: Generates code for basic arithmetic operations.
- `createFileHandlingProgram()`: Generates a file reading/writing program.
- `createSortingProgram(const std::string &algorithm)`: Generates code for sorting (e.g., bubble, quicksort).
- `createMatrixMultiplicationProgram()`: Generates code for matrix operations.

9. Interactive Features

- `provideExamples()`: Displays examples of input for users.
- `provideHelp()`: Offers detailed help and guidance.
- `addCommentsToCode(const std::string &code, const std::string &comments)`: Adds comments to generated code.
- `convertCodeToPseudoCode(const std::string &code)`: Translates C++ back to pseudo-code for learning.
- `generateDocumentation(const std::string &code)`: Automatically creates documentation for the generated code.

10. Advanced Features

- `supportLanguageExtensions()`: Allows additional languages in the future.
- `handleAdvancedDataStructures(const std::string &input)`: Maps instructions for structs, classes, or templates.
- `implementErrorHandlingInCode(const std::string &code)`: Adds try-catch blocks.

- `analyzePerformance(const std::string &code)`: Checks for potential inefficiencies.
- `suggestImprovements(const std::string &code)`: Recommends optimizations for the generated code.

Getting Started

1. Input Command

- Type a natural language instruction in the **Input** box. Example:
  `find the fibonacci upto 10`

2. Compile

- Click the **Compile** button to process the input.
- The system generates a parse tree, constructs an AST, and produces the equivalent C++ code.
- The output is displayed on the **Output** section.

3. Manual Upload

- Click the **Manual** button to upload a PDF guide or documentation for additional reference.
- The uploaded document can be accessed anytime for further information.

Understanding the Output

1. **Parse Tree Structure**
   - Shows how the input is broken down into meaningful components.
   - Example:
     `Action: fibonacci`
   - `├── code: 10`
2. **AST (Abstract Syntax Tree)**
   - Represents the hierarchical structure of the generated code.
   - Example:
     `#include <iostream>`
   - `void generateFibonacci(int n) {`
   - `    // Fibonacci logic`
   - `}`
3. **Generated C++ Code**

  - Final compiled output in C++.

Troubleshooting

- **No output generated?** Ensure that the input follows a clear programming-related instruction.
- **Incorrect code output?** Try rephrasing the input for better accuracy.
- **UI issues?** Refresh the page or restart the compiler.

Contact Support

For any issues, contact the development team at **aryanbansal182004@gmail.com** or refer to the uploaded manual for additional guidance.

Conclusion

The **Natural Language to Code Compiler** is designed to bridge the gap between human instructions and functional C++ programming. With a structured parsing system, intelligent mapping to C++ constructs, and a robust backend, it ensures seamless code generation. Whether you're a beginner learning C++ or an advanced user looking for quick conversions, this tool simplifies the process significantly.

For further assistance, refer to the **Manual** button in the UI or consult the documentation.