README-A1

Lexical Analyzer for Ada Subset

Welcome to the Lexical Analyzer project! This project is part of a compiler construction course and is designed to scan a subset of the Ada programming language, breaking the source code into meaningful tokens. The project uses several modules to keep the design modular, robust, and maintainable.

Overview

The Lexical Analyzer reads an Ada source file and produces a stream of tokens. It handles:

- **Reserved Keywords:** e.g., `PROCEDURE`, `MODULE`, etc.
- **Identifiers:** Variable names and procedure names. (Identifiers longer than 17 characters are rejected.)
- **Numeric Literals:** Both integer and real number tokens.
- **String Literals:** Enclosed in double quotes with support for escaped quotes.
- **Character Literals:** Enclosed in single quotes with support for escaped single quotes.
- **Operators & Punctuation:** Assignment (`:=`), arithmetic operators, relational operators, and more.

Detailed debugging and logging are provided via a custom Logger, and file operations are handled by the FileHandler module.

Project Structure

- **Definitions.py:**
  Defines the token types (using an `Enum`), reserved words, and regular expression patterns for matching tokens.
  - **Regular Expressions:**
    - **LITERAL:** Matches string literals using the pattern
      `r'"(?:[^"\n]|"")*(?:"|$)'`
      which starts with a double quote, accepts any character that is not a double quote or a newline (or allows two consecutive double quotes as an escape), and then ends with a double quote (or end-of-input if unterminated).
    - **CHAR_LITERAL:** Matches character literals with the pattern
      `r"'(?:[^'\n]|'')(?:"+"'|$)"`

that handles a single quoted character (or an escaped quote).
    - **ID:** Matches identifiers with the pattern `r"[a-zA-Z][a-zA-Z0-9_]*"` while additional logic ensures identifiers are no longer than 17 characters.
    - Other patterns exist for numbers, operators, and punctuation.
- **Token.py:**
  Implements the `Token` class. Each token stores its type, lexeme, line number, column number, and any associated values (for numbers or literals). The class provides methods for debugging and user-friendly output.
- **LexicalAnalyzer.py:**
  Contains the `LexicalAnalyzer` class, which is responsible for:
    - Reading source code.
    - Skipping whitespace and comments.
    - Matching tokens using the regex patterns from Definitions.
    - Logging errors (e.g., identifiers longer than 17 characters are logged and skipped).
- **FileHandler.py:**
  Provides file-related operations such as finding, reading, writing, and appending to files. It supports both command-line arguments and interactive input (including a GUI file explorer via Tkinter if available).
- **Logger.py:**
  Implements a singleton logger that supports colored output, caller filtering, and configuration options for both console and file logging. This is used throughout the project to provide consistent debug information.
- **JohnA1.py:**
  Serves as the main driver. It uses FileHandler to read a source file, invokes the LexicalAnalyzer to tokenize the file, and then prints (and optionally writes) the token table.

How to Run

Prerequisites

- **Python 3.6+**
- Tkinter (optional): For the file explorer functionality. If not installed, the program will prompt you to enter file paths manually.

Installation

1. **Clone the Repository:**

```
git clone https://github.com/jakujobi/Ada_Compiler_Construction.git
```

2. Navigate to the Directory:**

```
cd Ada_Compiler_Construction
cd A1 - lexical Analyzer
```

3. **Set Up a Virtual Environment (Optional):**

```
python3 -m venv venv
source venv/bin/activate  # For Windows: venv\Scripts\activate
```

Running the Analyzer

To run the lexical analyzer, use the following command:

```
python3 JohnA1.py <input_file.ada> [output_file.txt]
```

- **input_file.ada:**
  The Ada source file to analyze.
- **output_file.txt (Optional):**
  File where the token table will be saved.

Example

If you have a file named `example.ada`, run:

```
python3 JohnA1.py example.ada tokens.txt
```

This will:

1. Read the source code from `example.ada`.
2. Tokenize the source code.
3. Print the source code and a formatted token table to the console.
4. Save the token table to `tokens.txt`.

Design Decisions

- **Modular Design:**
  The project is split into multiple modules (Definitions, Token, LexicalAnalyzer, FileHandler, Logger, JohnA1) to enhance code reuse and maintainability.
- **Regular Expressions:**
  Each token type is matched using specific regex patterns. For instance:
  - **LITERAL:**
    Matches string literals and supports escaped quotes.
  - **CHAR_LITERAL:**
    Matches character literals and handles escape sequences.
  - **ID:**
    Matches identifiers while enforcing a maximum length of 17 characters via additional logic.
- **Error Handling:**
  The lexer logs errors (for example, when an identifier is too long) and skips such tokens to prevent further processing issues.
- **Logging:**
  A custom Logger is used throughout the project to provide detailed debugging information. This helps in development and troubleshooting.
- **File Operations:**
  The FileHandler module abstracts all file-related operations, making it easy to handle files across various parts of the project.

Acknowledgements and Ethical Use of AI

Ethical Considerations

This project was developed with ethical practices in mind, especially concerning the use of artificial intelligence. Here are the key points regarding ethical use:

- **Local, Self-Run AI LLM Models**: *DeepSeek-R1-Distill-Qwen-14B*
  - was used to to provide assistance in code debugging, and documentation including parts of this one you are currently reading * *hi.
- **IDE Integrated AI**: Cody AI
  - helped in suggesting improvements, generating documentation, and offering coding examples for small snippets of code.
    - For example: Helped suggest regex patterns based on he descriptions and prompts that i gave it
  - I reviewed all generated suggestions before integrating them into the project
- **Research:**
  I used Perplexity, Microsoft Copilot to search for and learn about new concepts.

    - An example of this is `singleton`, a class that maintains the same single instance across every where it is initiated. This information helped me when making the `Logger` module as it allowed me to collate all logs into the same instance and configuration.
- **Purpose of AI Use:**
  The AI assistance in this project was intended to:
  - Help with code documentation and commenting.
  - Provide suggestions on design and error handling.
  - Aid in the generation of additional documentation and support materials.
  The tools were not used to replace my human judgment or to make decisions on my behalf but to serve as a helpful tool in my learning, and the software development process.

Contributing

Contributions are welcome! If you have suggestions or improvements:

- Fork the repository.
- Make your changes.
- Submit a pull request.
- Ensure your changes are well-documented and tested.

License

This project is licensed under the Hippocratic License. See the LICENSE file for details.

I prefer this licence because it allows my code to be used by other people on the condition of upholding ethical standards as written.
For example:

- Licensee shall not, whether indirectly or indirectly through agents or assignee: infringe on human rights as outlined by the United Nations Universal Declaration of Human Rights
- Not for use by or for any Military Activities, Mass Surveillance, Law Enforcement, Ecocide, Removal of Indigenous peoples from their lands.
  True, it sounds a bit extreme, but I have heard of open source projects being used for "special military operations", "exercising the rights of a nation to defend itself through genocide of thousands of families, women and children included". (see events from 2022 to 2025 -at moment of writing)