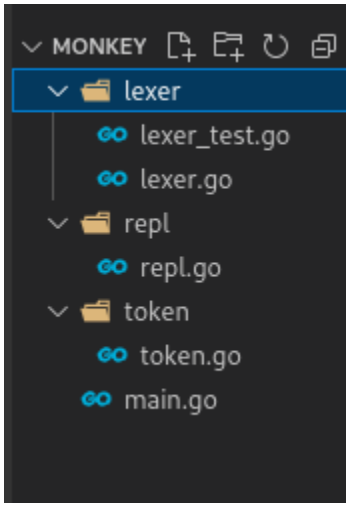
In lexical Analyzer:

- [ ] **Scans the pure HLL code line by line.**
- [ ] **Takes Lexemes as i/p and produces Tokens.**

Lexemes —->Scanning eliminate non- token elements—>Analyzing —---->Tokens



From the above In the lexical Analyzer folder.

Token/token.go code.

This code defines a package called "token" that provides functionality related to lexical tokens in a programming language. Here's a breakdown of its components:

1. `TokenType` **is a custom type defined as a string.**
    - **It represents the type of a token, such as "IDENT" for identifiers, "INT" for integers, and various keywords and operators.**
2. **Constants are defined for different token types:**
    - `ILLEGAL` **and** `EOF` **represent illegal and end-of-file tokens, respectively.**
    - `IDENT` **represents identifiers.**
    - `INT` **represents integer literals.**
    - **Various operators and delimiters are also defined as constants, such as** `ASSIGN`, `PLUS`, `MINUS`, `COMMA`, `SEMICOLON`, **etc.**
    - **Keywords like** `FUNCTION`, `LET`, `TRUE`, `FALSE`, `IF`, `ELSE`, **and** `RETURN` **are also defined.**
3. **The** `Token` **struct represents a token and has two fields:**
    - `Type` **of** `TokenType` **represents the type of the token.**

    - `Literal` of `string` represents the literal value of the token.
4. A map named `keywords` is defined, which maps keyword strings to their respective `TokenType`.
    - For example, the keyword "fn" is mapped to the `FUNCTION` token type.
5. The `LookupIdent` function takes an identifier as input and returns its corresponding `TokenType`.
    - It checks if the identifier exists in the `keywords` map and returns the associated token type if found.
    - If the identifier is not a keyword, it returns `IDENT` as the token type, indicating it is a regular identifier.

Overall, this code provides a way to represent different types of tokens, associate keywords with their token types, and perform a lookup to determine the token type of an identifier. It serves as a foundation for building a lexer or tokenizer for a programming language.

lexer/lexer.go

This code defines a lexer package that implements a lexer for the Monkey programming language. Here's a breakdown of its components:

1. The `Lexer` struct represents the lexer and contains the following fields:
    - `input` is the input string that the lexer will tokenize.
    - `position` is the current position in the input string.
    - `readPosition` is the current reading position in the input string (after the current character).
    - `ch` is the current character under examination.
2. The `New` function is a constructor for the `Lexer` struct. It initializes a new lexer with the provided input string, sets the initial positions, and reads the first character from the input.
3. The `NextToken` method is responsible for tokenizing the input string and returning the next token.
    - It skips any whitespace characters at the current position.
    - It switches on the current character `l.ch` to determine the token type.
    - For example, if `l.ch` is '=', it checks if the next character is also '=', and if so, creates an `EQ` token for "==".
    - If the next character is not '=', it creates an `ASSIGN` token for "=".
    - Similar logic is applied for other operators, delimiters, and special characters.
    - If the character is a letter, it reads the identifier and looks up its token type using `LookupIdent`.

    - If the character is a digit, it reads the number and assigns the `INT` token type.
    - If the character is not recognized, it creates an `ILLEGAL` token.
    - After processing the token, it reads the next character.
4. The `skipWhitespace` method is used to advance the lexer's position past any whitespace characters (spaces, tabs, newlines, etc.).
5. The `readChar` method reads the next character from the input string and updates the lexer's position accordingly.
6. The `peekChar` method returns the next character in the input string without advancing the lexer's position.
7. The `readIdentifier` method reads an identifier (a sequence of letters) from the input string, starting from the current position.
8. The `readNumber` method reads a number (a sequence of digits) from the input string, starting from the current position.
9. The `isLetter` function checks if a given character is a letter (a-z, A-Z, or underscore).
10. The `isDigit` function checks if a given character is a digit (0-9).
11. The `newToken` function creates a new `Token` struct with the given token type and the literal value of the character.

Overall, this code defines a lexer that takes an input string, tokenizes it, and produces a stream of tokens representing the different components of the Monkey programming language, such as identifiers, keywords, operators, and literals.

repl/repl.go

This code defines a package called "repl" (short for "read-eval-print loop") that provides functionality for running a simple interactive shell for the Monkey programming language. Here's a breakdown of its components:

1. The `Start` function is the entry point for the REPL. It takes two arguments: `in` (an `io.Reader` interface) for input and `out` (an `io.Writer` interface) for output.
2. Inside the `Start` function, a `bufio.Scanner` is created to read input from the provided `in` reader.
3. The REPL enters a loop where it repeatedly prompts for user input and processes it.
4. The `fmt.Printf(PROMPT)` statement prints the prompt string ">> " to the output to indicate that the REPL is ready to accept input.

5. `scanner.Scan()` is called to read the next line of input. If `scanner.Scan()` returns `false`, it means there is no more input, and the function returns, effectively ending the REPL.
6. The line of input is obtained from `scanner.Text()`.
7. A new lexer (`l := lexer.New(line)`) is created for the input line.
8. The lexer is used to iterate over the tokens in the input line by repeatedly calling `l.NextToken()`.
9. Inside the loop, each token is printed using `fmt.Printf("%+v\n", tok)` in a formatted way.
10. The loop continues until the lexer returns an `EOF` token, indicating the end of the input line.

Overall, this code sets up a basic REPL that reads input lines, tokenizes them using the lexer from the Monkey language, and prints the resulting tokens. It provides a simple way to interactively experiment with the lexer component of the Monkey programming language.

Main.go

This code is the main entry point for the Monkey programming language interpreter. Here's a breakdown of its components:

1. **The `main` function is the entry point of the program.**
2. **It first attempts to retrieve the current user's information using `user.Current()`. If an error occurs, it panics, terminating the program.**
3. **The username is obtained from the `user` struct and printed as a welcome message.**
4. **A greeting message is printed, indicating that the user is using the Monkey programming language and is invited to type in commands.**
5. **Finally, the REPL (read-eval-print loop) is started by calling `repl.Start()` and passing `os.Stdin` (standard input) and `os.Stdout` (standard output) as the input and output sources, respectively. This sets up an interactive session where the user can input Monkey programming language statements and see the corresponding output.**

Overall, this code initializes the Monkey interpreter, displays a welcome message to the user, and starts the REPL to interact with the Monkey programming language. It serves as the starting point for executing Monkey code and provides a user-friendly interface for working with the language.