PROJECT REPORT

Compiler Construction Lab (CSL-323)

PROJECT TITLE: REPL Compiler

BS(CS) – 5B

Group Members

Name	Enrollment
1. AYESHA	02-134231-026
2. MAHA	02-134231-096
3. NIDA	02-134231-086

Submitted to:

Ma’am Mehwish Saleem

BAHRIA UNIVERSITY KARACHI CAMPUS

Department of Computer Science

ABSTRACT

This project presents the design and implementation of a **Custom Python REPL (Read-Eval-Print Loop)** as part of a Compiler Construction course. The REPL environment facilitates interactive Python code execution while incorporating key compiler construction concepts such as **tokenization, lexical analysis, error detection, and state management**.

The system tokenizes user input to identify and classify lexical tokens like keywords, operators, identifiers, and numbers, storing both valid and erroneous tokens separately for further inspection. It supports multi-line inputs, maintains an execution environment, and tracks command history with timestamps. The REPL also includes utility commands for viewing environment variables, token logs, command history, and usage statistics, enhancing the user's debugging and learning experience.

Additionally, the REPL implements a debug mode for stepwise code execution, providing detailed token information and environment snapshots after each command. Error handling captures exceptions gracefully and records tokens from erroneous code, improving the robustness and diagnostic capabilities of the interpreter.

This project demonstrates foundational principles of compiler design such as lexical analysis, symbol table management, and error reporting, packaged within a practical and user-friendly Python shell environment.

INTRODUCTION

The REPL (Read-Eval-Print Loop) project is designed to create a dynamic and interactive programming environment where users can write and execute Python code seamlessly. It bridges the gap between coding and seeing immediate results, making it an ideal tool for learning, experimentation, and rapid prototyping.

Implemented entirely in Python, this custom REPL offers features such as multi-line input handling, real-time code evaluation, command history management, and a comprehensive token analyzer. The token analyzer classifies lexical tokens into categories like keywords, operators, identifiers, numbers, and symbols. It also maintains separate token logs for successfully executed code and erroneous code, assisting users in understanding Python's lexical structure and debugging their programs more effectively.

This project caters to a wide audience—from beginners who want to grasp Python programming fundamentals interactively, to experienced developers seeking a lightweight environment to test snippets or debug issues. The REPL's user-friendly interface and useful commands such as environment inspection, debug mode, and statistics display provide an enriching coding experience.

Moreover, the REPL incorporates robust error handling, enabling users to identify, log, and learn from their mistakes in real-time. By combining these features, the project makes programming more accessible, engaging, and educational for users of all skill levels.

PROBLEM STATEMENT:

While Python is renowned for its simplicity and accessibility, beginners and even experienced developers often face hurdles in experimenting with and debugging code interactively. Key challenges include:

1. **Lack of Real-Time Feedback:** Without immediate and clear feedback, beginners struggle to identify and understand coding errors as they write code.
2. **Limited Accessibility:** Traditional IDEs and complex setups may overwhelm new users, hindering exploration and learning.
3. **Token Analysis Gap:** Users lack tools to break down and analyze code tokens (keywords, identifiers, operators, symbols) that are essential for understanding and debugging.
4. **Environment Management Difficulties:** Resetting or managing the coding environment in existing tools can be cumbersome, making iterative learning inefficient.

The REPL project addresses these issues by providing a lightweight, interactive Python environment that delivers instant feedback, token analysis, and seamless environment management. It logs tokens from both successful and erroneous code executions into separate files, assisting users in debugging and understanding their code more effectively.

METHODOLOGY:

The REPL tool is developed with a focus on usability, educational value, and robust functionality through the following key components:

1. **Interactive Shell Design:**
   A Python-based REPL accepts multi-line user input and dynamically executes code using exec() for statements and eval() for expressions, maintaining a global environment to preserve state across commands.
2. **Token Analysis and Classification:**
   Using regular expressions, the REPL tokenizes user input, classifying tokens as keywords, identifiers, operators, numbers, or symbols. Tokens from error-causing code snippets are saved separately to help users identify problematic code parts.
3. **Command History Management:**
   Commands entered during sessions are timestamped, stored in-memory, and persisted to a history file. Users can view and reuse past commands to improve efficiency.

4. **Environment Control:**
   Users can clear the global execution environment to reset all variables and definitions, facilitating clean testing of new code snippets without residual state interference.
5. **Error Handling and Feedback:**
   The system captures and displays detailed error messages with color-coded emphasis. It saves tokens related to erroneous code separately to support error analysis.
6. **Additional Utilities:**
   Commands for displaying tokens, error tokens, environment variables, and usage statistics provide transparency. A debug mode allows step-by-step code execution with detailed token and environment snapshots.
7. **Persistence and Usability Enhancements:**
   History persistence across sessions, colored and formatted output using colorama, and an intuitive command set contribute to a beginner-friendly coding experience.
8. **Testing and Iterative Feedback:**
   Continuous testing ensures graceful handling of diverse inputs. User feedback guides iterative improvements to functionality and usability.

PROJECT SCOPE

1. **Core REPL Functionality:**
   An interactive loop to accept, execute, and display results of Python code with support for multi-line inputs.
2. **Token Analysis System:**
   Robust tokenization and classification of user code, with separate logs for successful and error executions to facilitate debugging and learning.
3. **Persistent Command History:**
   Time-stamped commands are saved and can be reviewed or reused in future sessions.
4. **Environment Management:**
   Ability to clear the current execution context, allowing fresh starts and reducing confusion from lingering variables.
5. **User-Friendly Interface:**
   Color-coded, clear feedback for outputs, errors, and commands, plus accessible help and stats commands to support users at all skill levels.
6. **Error-Specific Debugging Tools:**
   Dedicated logs and immediate error messages to help users quickly identify and fix problems.
7. **Educational Support Features:**
   Debug mode with stepwise execution, token display, and environment snapshots to deepen understanding of code flow and structure.

By integrating these features, the REPL project empowers users—especially beginners—to experiment with Python interactively and effectively, bridging gaps in traditional learning environments and IDEs.

CODE

```
import os
import re
from datetime import datetime
from colorama import Fore, Style, init

# Initialize colorama
init(autoreset=True)

# === Global Variables ===
global_env = {}
command_history = []
execution_count = 0

# === File Constants ===
HISTORY_FILE = "repl_history.txt"
TOKENS_FILE = "tokens.txt"
ERROR_TOKENS_FILE = "error_tokens.txt"

# === Input Handler ===
def get_user_input(prompt=">>> "):
    lines = []
    while True:
        line = input(prompt)
        if not line.strip():
            break
        lines.append(line)
    return "\n".join(lines)

# === Tokenization ===
def tokenize_code(code, is_error=False):
    pattern = r"\b\w+\b|[\+\-\*/=<>!&|;:(){}\[\],.]"
    tokens = re.findall(pattern, code)
    token_data = [(token, classify_token(token)) for token in tokens]

    append_tokens_to_file(token_data, ERROR_TOKENS_FILE if is_error else
TOKENS_FILE)
    return token_data

def classify_token(token):
    keywords = {"def", "class", "return", "if", "else", "for", "while",
"import", "from", "as", "with"}
    operators = {"+", "-", "*", "/", "=", "<", ">", "!", "&", "|"}
    if token in keywords:
        return "KEYWORD"
    elif token in operators:
        return "OPERATOR"
    elif re.match(r"^\d+$", token):
        return "NUMBER"
    elif re.match(r"^[a-zA-Z_]\w*$", token):
        return "IDENTIFIER"
    else:
        return "SYMBOL"
```

```
def append_tokens_to_file(token_data, file_name):
    try:
        with open(file_name, "a") as f:
            for token, token_type in token_data:
                f.write(f"{token}: {token_type}\n")
        # Removed noisy print here for cleaner REPL output
    except IOError as e:
        print(f"{Fore.RED}❌ Error saving tokens: {e}")

def show_tokens(file_name):
    if not os.path.exists(file_name):
        print(f"{Fore.RED}No tokens found in {file_name}.")
        return
    print(f"{Fore.MAGENTA}{Style.BRIGHT}📦 Tokens in {file_name}:")
    try:
        with open(file_name, "r") as f:
            for line in f:
                print(f"{Fore.CYAN}{line.strip()}")
    except IOError as e:
        print(f"{Fore.RED}❌ Error reading tokens: {e}")

# === Evaluation ===
def evaluate_code(code):
    global execution_count
    try:
        # If code contains assignment, function or class definition, use exec
        if any(kw in code for kw in ["=", "def ", "class "]):
            exec(code, global_env)
            execution_count += 1
            return None
        else:
            result = eval(code, global_env)
            execution_count += 1
            return result
    except Exception as e:
        # Save tokens from erroneous code
        tokenize_code(code, is_error=True)
        return f"{Fore.RED}{Style.BRIGHT}⚠ Error: {Style.RESET_ALL}{e}"

# === Output ===
def display_output(result):
    if result is not None:
        print(f"{Fore.GREEN}{Style.BRIGHT}🎯 Output:\n{Fore.CYAN}{result}")

# === History Handling ===
def add_to_history(command):
    timestamp = datetime.now().strftime("%H:%M:%S")
    command_history.append(f"[{timestamp}] {command}")

def show_history():
    if not command_history:
        print(f"{Fore.YELLOW}No commands in history yet.")
        return
    print(f"{Fore.MAGENTA}{Style.BRIGHT}🕓 Command History:")
    for idx, cmd in enumerate(command_history):
```

```
        print(f"{Fore.YELLOW}{idx + 1}: {Fore.RESET}{cmd}")

def save_history_to_file():
    try:
        with open(HISTORY_FILE, "w") as f:
            f.write("\n".join(command_history) + "\n")
        print(f"{Fore.GREEN}💾 History saved.")
    except IOError as e:
        print(f"{Fore.RED}❌ Error saving history: {e}")

def load_history_from_file():
    if not os.path.exists(HISTORY_FILE):
        print(f"{Fore.YELLOW}ℹ️ No history file found.")
        return
    try:
        with open(HISTORY_FILE, "r") as f:
            for line in f:
                stripped = line.strip()
                if stripped:
                    command_history.append(stripped)
        print(f"{Fore.GREEN}📂 History loaded.")
    except IOError as e:
        print(f"{Fore.RED}❌ Error loading history: {e}")

# === Utility Commands ===
def show_help():
    print(f"""{Fore.CYAN}{Style.BRIGHT}
📜 Available Commands:
──────────────────────────────
 ◇ help          → Show this help menu
 ◇ history       → Show command history
 ◇ clear         → Clear the environment
 ◇ tokens        → Show valid tokens
 ◇ error_tokens → Show error tokens
 ◇ env           → Show environment variables
 ◇ stats         → Display usage statistics
 ◇ debug         → Step-by-step code execution
 ◇ exit()        → Exit the REPL
""")

def show_environment():
    if not global_env:
        print(f"{Fore.YELLOW}Environment is empty.")
        return
    print(f"{Fore.MAGENTA}{Style.BRIGHT}🌍 Current Environment:")
    for key, val in global_env.items():
        # Skip builtins to keep output clean
        if key.startswith("__") and key.endswith("__"):
            continue
        print(f"{Fore.YELLOW}{key}: {Fore.CYAN}{val}")

def show_stats():
    print(f"""{Fore.BLUE}{Style.BRIGHT}
```

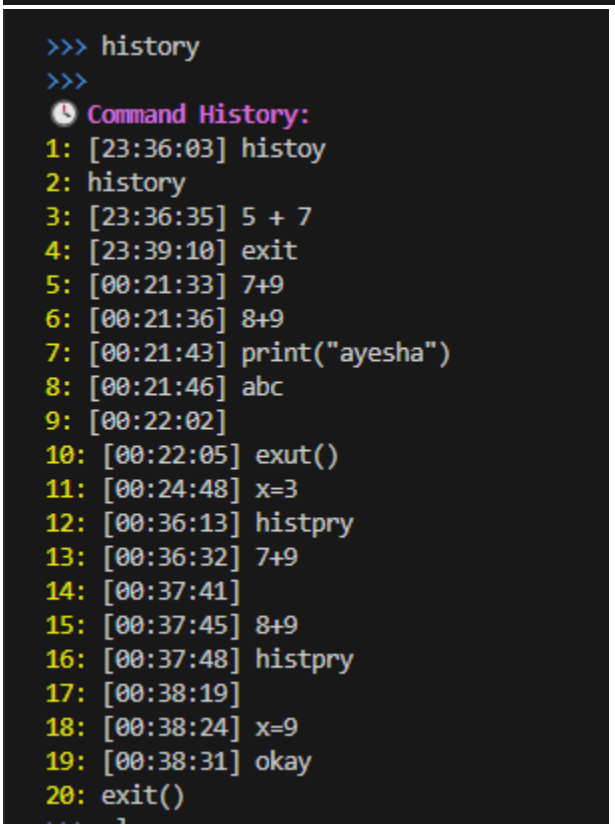
```
📊 REPL Stats:
──────────────
{Fore.YELLOW}• Commands Executed: {execution_count}
• History Entries : {len(command_history)}""")

def debug_mode():
    print(f"{Fore.YELLOW}🐞 Debug mode: Enter lines one by one. Type 'done'
to exit.\n")
    step = 1
    while True:
        line = input(f"{Fore.YELLOW}→ {Style.RESET_ALL}")
        if line.strip().lower() == "done":
            break

        # Show entered code
        print(f"{Fore.YELLOW}Debug Step {step}:")
        print(f">>> {line}")

        # Tokenize and show tokens (but do NOT save error tokens here to keep
debug output clean)
        tokens = tokenize_code(line, is_error=False)
        if tokens:
            print("Tokens:")
            for token, token_type in tokens:
                print(f"  {token}: {token_type}")
        else:
            print("Tokens: (none)")

        # Evaluate code and show result or error
        result = evaluate_code(line)
        if result is None:
            print("Result: None")
        elif isinstance(result, str) and result.startswith(f"{Fore.RED}"):
            # This is an error message from evaluate_code
            print(result)
        else:
            print(f"Result:\n{Fore.CYAN}{result}")

        # Show environment snapshot (only user-defined keys, skip builtins
and modules)
        print("Environment snapshot:")
        user_vars = {k: v for k, v in global_env.items() if not
k.startswith("__") and not callable(v) and not isinstance(v, type(os))}
        if user_vars:
            for k, v in user_vars.items():
                print(f"  {k}: {v}")
        else:
            print("  (empty or builtins only)")

        print(f"{'-'*40}")
        step += 1

def clear_environment():
    global global_env
    global_env.clear()
    print(f"{Fore.YELLOW}🧹 Environment cleared.")
```

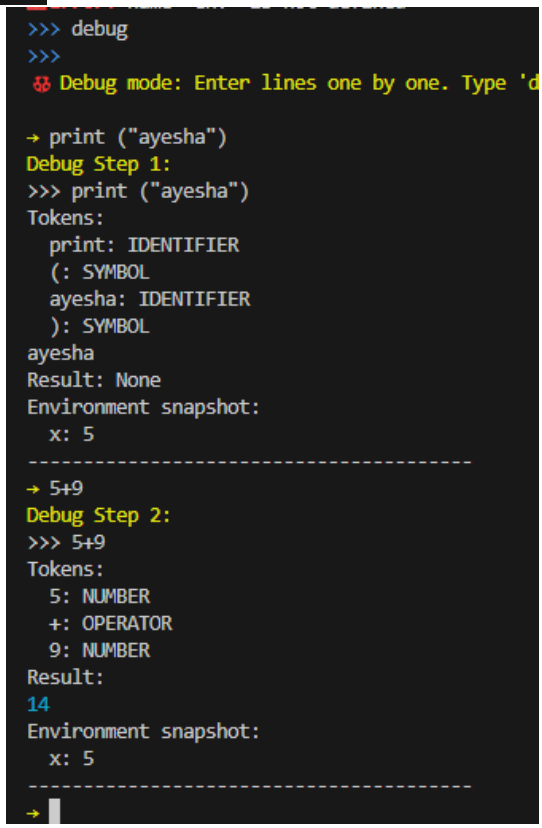
```
# === Main REPL Loop ===
def repl_loop():
    print(f"""{Fore.GREEN}{Style.BRIGHT}
```


```
{Fore.YELLOW}Type 'help' to view available commands.
""")
    load_history_from_file()

    while True:
        try:
            code = get_user_input(prompt_style)
            cmd = code.strip().lower()

            if cmd == "exit()":
                save_history_to_file()
                print(f"{Fore.CYAN}👋 Goodbye!")
                break
            elif cmd in commands:
                commands[cmd]()
            else:
                add_to_history(code)
                result = evaluate_code(code)
                # Tokenize on success or failure separately to avoid double
writing
                if result is None or (isinstance(result, str) and not
result.startswith(Fore.RED)):
                    tokenize_code(code)
                display_output(result)
        except KeyboardInterrupt:
            print(f"\n{Fore.RED}⛔ KeyboardInterrupt. Type 'exit()' to
quit.")
            save_history_to_file()  # Save history before continuing or exit
        except EOFError:
            print(f"\n{Fore.RED}⛔ EOF received. Exiting...")
            save_history_to_file()  # Save history before exiting
            break

# === Command Dispatcher ===
commands = {
    "help": show_help,
    "history": show_history,
    "clear": clear_environment,
    "tokens": lambda: show_tokens(TOKENS_FILE),
    "error_tokens": lambda: show_tokens(ERROR_TOKENS_FILE),
    "env": show_environment,
    "stats": show_stats,
    "debug": debug_mode,
}

# === Prompt Style ===
prompt_style = f"{Fore.BLUE}>>> {Style.RESET_ALL}"

# === Entry Point ===
```

```
if __name__ == "__main__":
    repl_loop()
```

OUTPUT



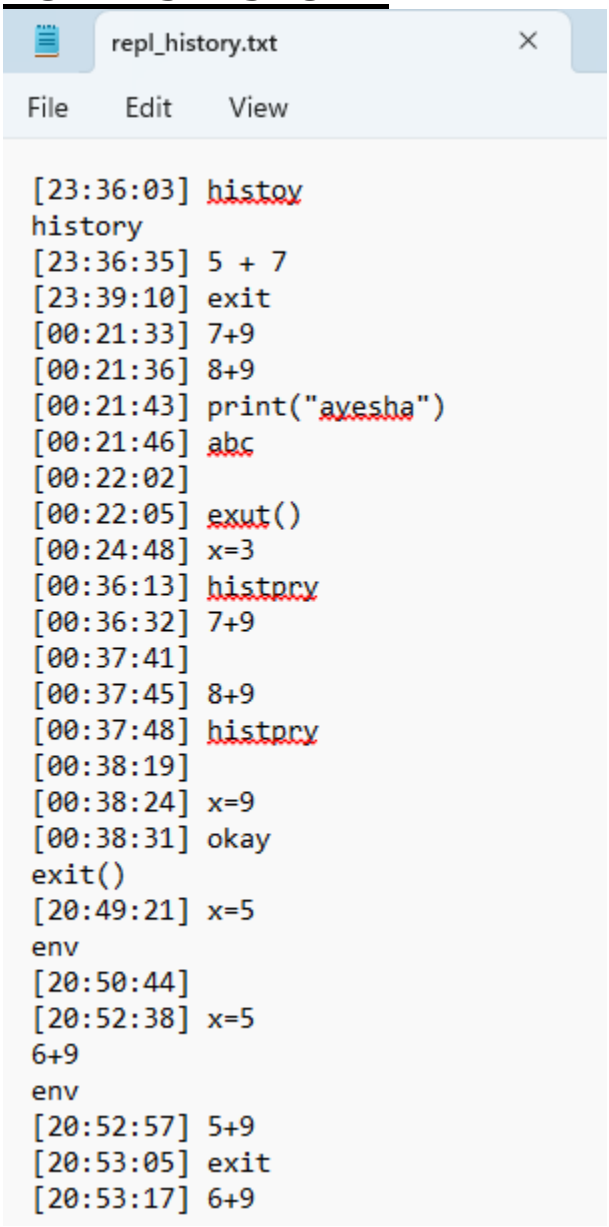


```
>>> tokens
>>>
📦 Tokens in tokens.txt:
print: IDENTIFIER
(: SYMBOL
Hello: IDENTIFIER
,: SYMBOL
REPL: IDENTIFIER
!: OPERATOR
): SYMBOL
5: NUMBER
+: OPERATOR
3: NUMBER
*: OPERATOR
2: NUMBER
def: KEYWORD
square: IDENTIFIER
(: SYMBOL
x: IDENTIFIER
): SYMBOL
:: SYMBOL
return: KEYWORD
x: IDENTIFIER
*: OPERATOR
x: IDENTIFIER
square: IDENTIFIER
(: SYMBOL
4: NUMBER
): SYMBOL
print: IDENTIFIER
(: SYMBOL
Hello: IDENTIFIER
,: SYMBOL
REPL: IDENTIFIER
!: OPERATOR
```

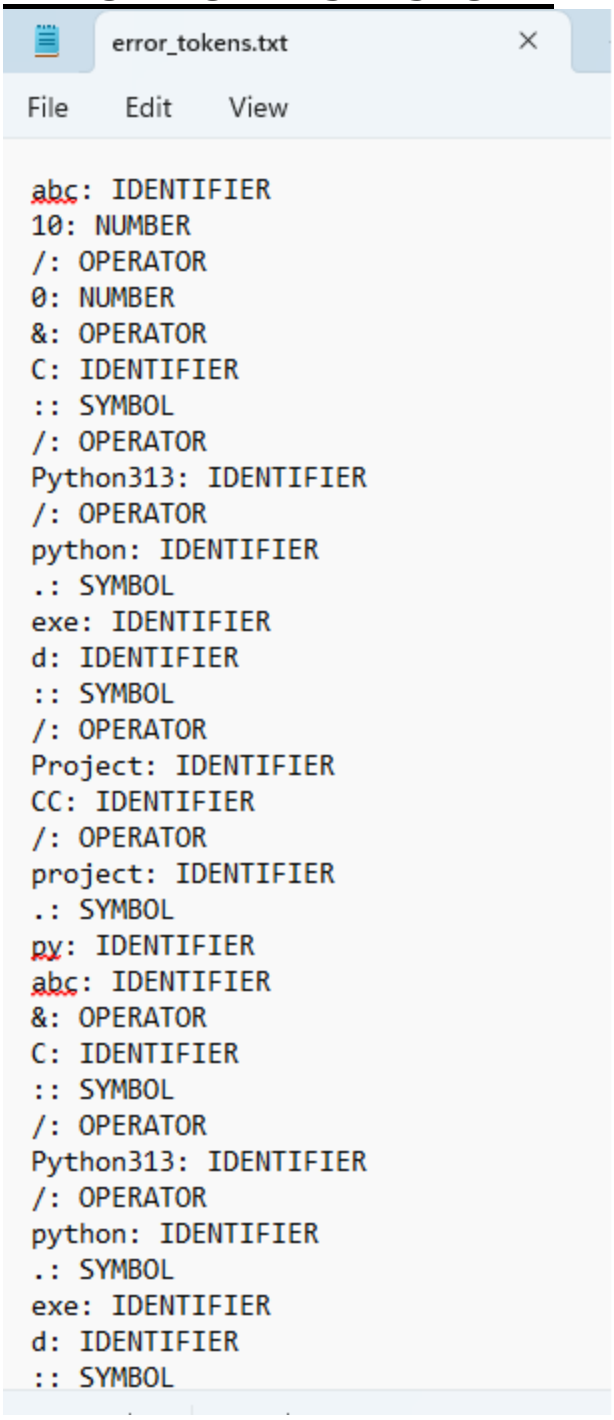
```
>>> error_tokens
>>>
📦 Tokens in error_tokens.txt:
abc: IDENTIFIER
10: NUMBER
/: OPERATOR
0: NUMBER
&: OPERATOR
C: IDENTIFIER
:: SYMBOL
/: OPERATOR
Python313: IDENTIFIER
/: OPERATOR
python: IDENTIFIER
.: SYMBOL
exe: IDENTIFIER
d: IDENTIFIER
:: SYMBOL
/: OPERATOR
Project: IDENTIFIER
CC: IDENTIFIER
/: OPERATOR
project: IDENTIFIER
.: SYMBOL
py: IDENTIFIER
abc: IDENTIFIER
&: OPERATOR
C: IDENTIFIER
```

```
>>> 6+9
>>>
🎯 Output:
15
>>> stats
>>>

📊 REPL Stats:
──────────────

• Commands Executed: 5
• History Entries : 26
>>> exit()
>>>
💾 History saved.
👋 Goodbye!
```

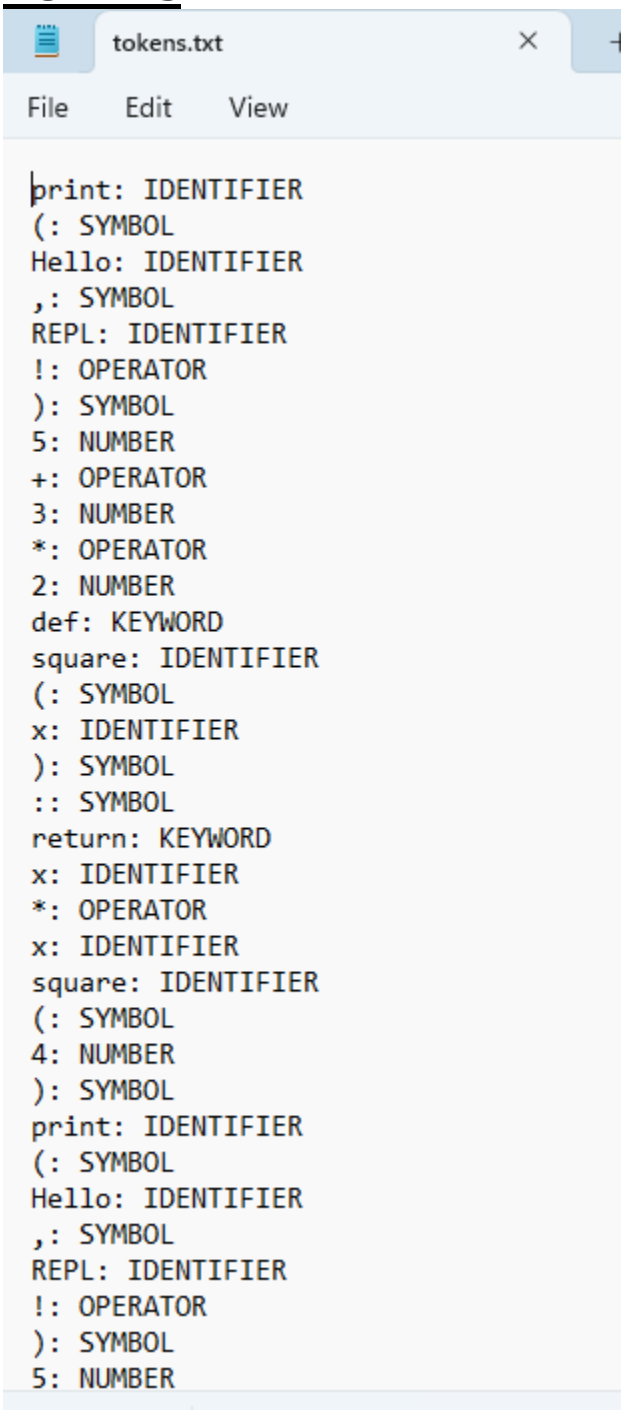
TOKENS HISTORY



ERROR TOKENS HISTORY



TOKENS


tokens.txt
File
Edit
View
print: IDENTIFIER
(: SYMBOL
Hello: IDENTIFIER
,: SYMBOL
REPL: IDENTIFIER
!: OPERATOR
): SYMBOL
5: NUMBER
+: OPERATOR
3: NUMBER
*: OPERATOR
2: NUMBER
def: KEYWORD
square: IDENTIFIER
(: SYMBOL
x: IDENTIFIER
): SYMBOL
:: SYMBOL
return: KEYWORD
x: IDENTIFIER
*: OPERATOR
x: IDENTIFIER
square: IDENTIFIER
(: SYMBOL
4: NUMBER
): SYMBOL
print: IDENTIFIER
(: SYMBOL
Hello: IDENTIFIER
,: SYMBOL
REPL: IDENTIFIER
!: OPERATOR
): SYMBOL
5: NUMBER

FUTURE DEVELOPMENT

The REPL project has promising potential for future enhancements. Some of the key areas for development include:

1. **GUI Integration:** Developing a graphical user interface to make the REPL more user-friendly and accessible, especially for non-technical users.
2. **Multi-Language Support:** Expanding the system to support additional programming languages such as JavaScript, Ruby, or Java, making the REPL more versatile and widely applicable.
3. **Advanced Debugging Tools:** Incorporating features like step-by-step code execution, real-time variable inspection, and runtime visualization to provide users with deeper insights into their code.
4. **Cloud-Based REPL:** Hosting the REPL on cloud platforms to enable remote access, real-time collaboration, and cloud-based session management.
5. **Interactive Tutorials:** Integrating guided tutorials directly within the REPL to help users learn Python and related concepts interactively, improving the learning curve for beginners.

Conclusion

The REPL project effectively demonstrates Python's power and flexibility in building interactive programming environments. By addressing common challenges in coding, learning, and debugging, it offers a practical and user-friendly tool that encourages experimentation and learning.

With features like token analysis, command history, and environment management, the REPL serves as a valuable resource for both novice and experienced programmers. Its design not only supports dynamic code execution but also fosters better understanding and productivity.

Looking ahead, the potential for enhancements such as multi-language support, GUI integration, and cloud-based access ensures that the REPL can continue to evolve and meet the growing needs of the programming community. These improvements will make programming more accessible, engaging, and effective for everyone.