北京郵電大學

大作业报告：基于领域特定脚本语言的客服机器人的设计与实现

学院：计算机学院（国家示范性软件学院）

专业：计算机科学与技术

班级：2022211305

学号：2022211683

姓名：张晨阳

2024 年 12 月 16 号

目录

1. 概述

1.1. 任务描述

领域特定语言（Domain Specific Language，DSL）可以提供一种相对简单的文法，用于特定领域的业务流程定制。本作业要求定义一个领域特定脚本语言，这个语言能够描述在线客服机器人（机器人客服是目前提升客服效率的重要技术，在银行、通信和商务等领域的复杂信息系统中有广泛的应用）的自动应答逻辑，并设计实现一个解释器解释执行这个脚本，可以根据用户的不同输入，根据脚本的逻辑设计给出相应的应答。

1.2. 基本要求

- 脚本语言的语法可以自由定义，只要语义上满足描述客服机器人自动应答逻辑的要求。
- 程序输入输出形式不限，可以简化为纯命令行界面。
- 应该给出几种不同的脚本范例，对不同脚本范例解释器执行之后会有不同的行为表现。

1.3. 作业环境

- Windows 11
- PyCharm 2024.1.4 (Professional Edition)
- Python 3.11.9
- git version 2.41.0

2. 脚本语言设计

2.1. 脚本语言示例

为了便于后续语法的讲解，首先给出一个基于我的脚本语言设计的银行客服脚本示例：

```
Step welcome
    Speak "欢迎来到银行客服中心，请问有什么可以帮您?"
    Listen 5, 30
    Case "查询余额"
        goto balanceProc
    Case "存款"
        goto depositProc
    Case "取款"
        goto withdrawProc
    Case "帮助"
        goto helpProc
    Case "退出"
        goto goodbye
    Default
Step balanceProc
    Speak "您的账户当前余额为" + $balance + "元。"
    goto welcome
Step depositProc
    Speak "请输入您要存入的金额："
    GetNum $depositAmount
    goto depositConfirm
Step depositConfirm
    Recharge $balance add $depositAmount
    Speak "存款成功！您的账户余额为" + $balance + "元。"
    goto welcome
Step withdrawProc
    Speak "请确认您是否要取出"
    Listen 5, 30
    Case "确认"
        goto withdrawConfirm
    Default
Step withdrawConfirm
    Consume $balance reduce 100
    Speak "取款成功！您的账户余额为" + $balance + "元。"
    goto welcome
```

```
Step helpProc
    Speak "如果您需要帮助，请告诉我具体问题，我会尽力帮助您。您可以选择存款、取款等"
    Listen 5, 30
    Case "存款"
        goto depositProc
    Case "取款"
        goto withdrawProc
    Default
Step goodbye
    Speak "感谢您使用我们的银行服务，祝您有愉快的一天！"
    Exit
Step wait
    Timeout 5
        goto goodbye
```

上述脚本实现了一个可以进行简短对话、存款、取款、处理用户简单需求的银行客服机器人，我们将在下一部分详细的讲解脚本语言的语法。

2.2. 脚本语言语法

简答地说，我们的每个 script 由多个 Step 组成，每个 Step 由多个 action 组成，每个 action 由多个小组件组成。

详细如下：

2.2.1.Step

每个 Step 是一个完整的流程步骤，表示一个状态，其中包含多个动作，如 Speak、Listen、Case 等。

Step 是脚本的基础单元。

语法：

```
Step <step_name>
    <action>
    <action>
    ...
```

- <step_name>：步骤的标识符，表示当前步骤的名称。
- <action>：在步骤中执行的操作，如 Speak、Listen、Case、Timeout 等。
- 每两个 step 之间不需要空行。
- 机器人在任意时刻只会处于一个 Step。
- 默认第一个初始状态为 welcome.

2.2.2.Speak

Speak 用于输出文本信息，可以是静态文本、动态表达式或变量。符号 + 用于将多个字符串连接起来。

语法：

```
Speak "<speak_text>"
```

- <speak_text>：输出的文本内容，可以是字符串、变量或包含加法等操作的表达式，支持字符串连接。

2.2.3.Listen

Listen 用于监听用户输入。

该操作会在指定的时间段内等待用户输入，若超时则执行 Default 操作。

语法：

```
Listen <listen_start>, <listen_stop>
```

- <listen_start>：开始监听的时间（单位：秒）。
- <listen_stop>：停止监听时间（单位：秒）。如果用户未在此时间内输入，将执行 Default 操作。

2.2.4.Goto

Goto 用于跳转到指定的步骤，通常是根据 Case 或 Timeout 等触发的。

语法：

```
goto <goto_step>
```

- <goto_step>：目标步骤的标识符。

2.2.5.Case

Case 用于判断用户输入并跳转到对应的步骤。

后接一个 goto 操作。

语法：

```
Case "<case_input>"
    goto <goto_step>
```

- <case_input>：用于匹配用户输入的步骤名。

2.2.6.Default

Default 用于在 Case 中未匹配到用户输入且 Listen 超时，则会跳转到默认步骤。

语法：

```
Default
```

- 根据不同的当前状态，实现不同的默认操作。
- 若处于 welcome 状态，则跳转到 wait 状态。
- 若处于非 welcome 状态，则跳转到 welcome 状态。

2.2.7.Timeout

Timeout 用于设置超时操作。

等待指定时间，超时后执行下一条操作。

语法：

```
Timeout <timeout_amount>
    <action>
```

- <timeout_amount>：超时的时间（单位：秒）。
- <action>：超时后的操作，通常是 Speak 或 goto。

2.2.8.GetNum

GetNum 用于获取用户输入的数字，并将其存储在变量中。

语法：

```
GetNum <variable>
```

- <variable>：要存储用户输入的变量名。例如$depositAmount

2.2.9.Recharge & Consume

Recharge 和 Consume 用于修改账户余额或进行数值计算。

新增 add 和 reduce 操作替代原来的加法和减法符号。

语法：

```
Recharge <variable> add <amount> | <variable>
Consume <variable> reduce <amount> | <variable>
```

- <variable>：要更新的变量名，例如 $balance。
- <amount>|<variable>：要增加或减少的数值，可以为常量，也可以为用户输入的变量。

2.2.10. 符号：+

在 Speak 操作中，符号 + 表示字符串的连接操作，用于将多个字符串或表达式连接成一个完整的输出。

语法：

```
"<string1>" + "<string2>" + <variable>
```

- <string1> 和 <string2>：可以是静态字符串、变量或表达式。
- 若为变量，则不需要双引号。

2.2.11. Exit

Exit 用于结束会话或对话，通常在流程的最后步骤使用。

语法：

```
Exit
```

3. 模块划分设计

3.1. 项目目录结构

为了清晰展示本项目的整体布局和各模块的功能，我进行了合理细致的模块划分与分模块设计，本节将详细介绍本项目的目录结构。

以下是本项目的目录树示意图：

```
DSL/
│
├── doc/
│   └── report.pdf
│
├── src/
│   ├── interpreter.py
│   ├── parser.py
│   └── lexer.py
│
├── scripts/
│   ├── script1.txt
│   ├── script2.txt
│   ├── script3.txt
│   ├── script4.txt
│   └── script5.txt
│
├── tests/
│   ├── test_lexer.py
│   ├── test_parser.py
│   └── test_interpreter.py
│
├── run.py
├── auto_run.py
└── README.md
```

目录说明

- **doc/**：该目录用于存放项目相关的文档资料，包含一个 report.pdf 文件，记录了项目的详细报告内容。
- **src/**：存放项目的源代码，是项目的核心部分。主要包含以下文件：
    - interpreter.py：解释器模块，负责根据解析生成的语法树执行脚本逻辑，

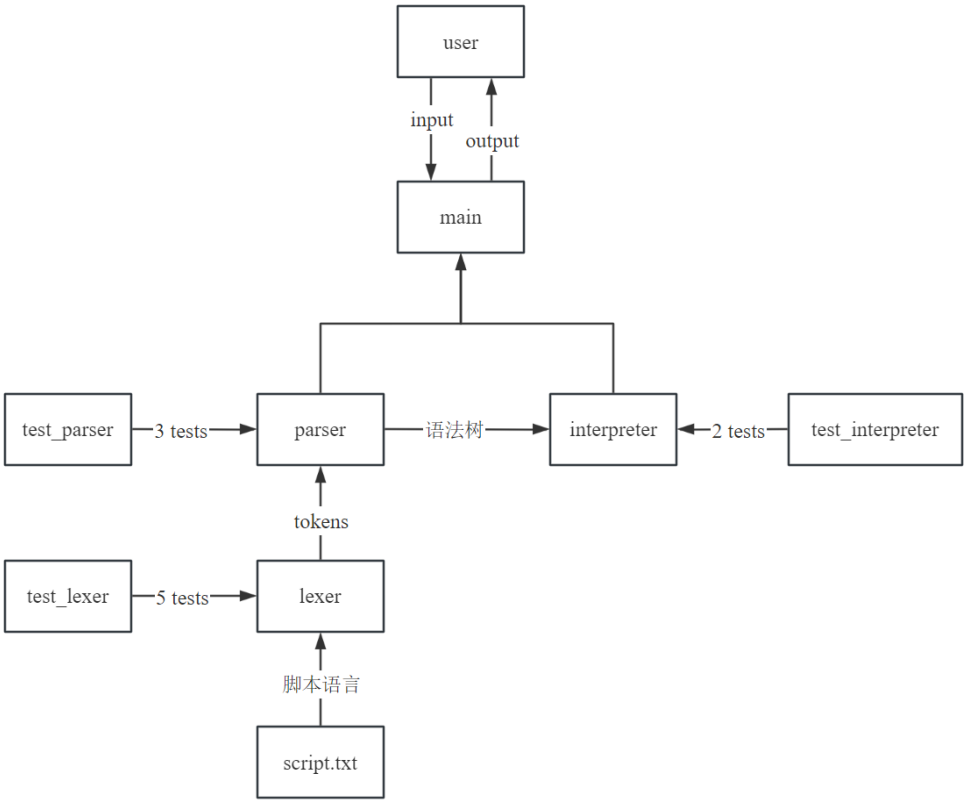
实现脚本语言的具体功能。

- parser.py：语法分析器模块，利用 pyparsing 库解析脚本语言的语法规则，将词法分析器生成的词法单元转换为语法树。
- lexer.py：词法分析器模块，使用 pyparsing 库，将输入的脚本分解为基本的词法单元，为语法分析器提供输入。

- **scripts/**：该目录用于存放不同的脚本文件，便于测试和演示脚本语言的功能。包含以下文件：
  - script1.txt：完整的脚本示例，模拟“冒险者协会”的业务流程。
  - script2.txt：完整的脚本示例，模拟“银行客服”的业务流程。
  - script3.txt：简易的脚本示例，包含“欢迎”和“退出”功能。
  - script4.txt：简易的脚本示例，包含“投诉”功能。
  - script5.txt：简易的脚本示例，允许用户自定义存钱金额。
- **tests/**：包含各模块的测试桩代码。具体包括：
  - test_lexer.py：针对词法分析器的测试桩，模拟正确的匹配过程，即优先匹配。验证词法单元的提取是否准确。
  - test_parser.py：针对语法分析器的测试桩，模拟正确的 lexer.py 的返回值，验证语法树的生成是否符合预期。
  - test_interpreter.py：针对解释器的测试桩，模拟变量替换功能的正确返回，确保脚本逻辑（特别是变量处理）的执行符合要求。
- **run.py**：项目的主程序入口，负责启动整个脚本语言的执行流程，从加载脚本、解析到执行。
- **auto_run.py**：自动运行脚本，自动进行用户输入，模拟完整的程序流程。
- **README.md**：项目说明文档，提供项目的基本介绍、安装步骤、使用方法及其他相关信息，便于用户快速上手和了解项目。

3.2. 整体架构设计

系统的整体架构如下：



对于一个给定的脚本文件 script.txt，首先由 lexer 分解为我们设置的基本的词法单元(keywords + tokens)，然后由 parser 子模块生成完整的语法树，该语法树传入 interpreter 模块，解析为可执行的脚本逻辑。

用户通过主程序入口进行输入，主程序接收用户输入并执行相关操作输出。

各子模块与其对应的测试桩进行不同的 test，均需通过 test。

同一个脚本文件可同时进行多个用户交互，各用户交互上下文独立。

下面我们将简要介绍子模块的划分。

3.3. 子模块设计

3.3.1.Lexer

在整个项目中，lexer 模块负责对输入的脚本文件进行词法分析，将其分解成基础的词法单元（tokens）。

这些词法单元包括关键词、标识符、变量、字符串、数字等，它们是后续语法分析（parser）和脚本执行（interpreter）的基础。lexer 模块的设计遵循了简洁、高效、可扩展的原则，确保能够支持灵活的脚本语言解析。

设计目标

- **识别关键词和符号**：lexer 通过定义一组关键词（如 Step、Speak、Listen 等）和符号（如数字、字符串、变量等）来识别脚本中的基本构件。这些关键词和符号是脚本语法的基础，lexer 会确保优先匹配这些关键词，从而为语法分析提供准确的词法单元。
- **支持变量和标识符**：脚本中可能包含用户定义的变量（如 $balance）和其他标识符（如步骤名称、标签等）。lexer 会正确区分这些用户定义的标识符与保留的关键词，确保脚本解析的正确性。

主要功能

- **关键词识别**：lexer 定义了一组关键词（如 Step、Speak 等），用于识别脚本中的控制结构。每个关键词在解析时会被转换为特定的 token，方便后续的语法分析和脚本逻辑执行。
- **标识符和变量识别**：通过定义标识符和变量的匹配规则，lexer 能够提取脚本中的动态元素（如步骤名、用户输入等）。其中，变量以 $ 开头，如 $balance，标识符则用于表示步骤名或其他自定义元素。
- **支持字符串和数字**：lexer 还支持对带引号的字符串和数字的识别，便于脚本中包含各种文字或数值数据，供后续解析和执行使用。
- **返回 token 流**：lexer 将输入文本转化为 token 流，这些 token 是脚本中的基本构件，包含了所有识别出的关键词、标识符、变量、字符串、数字等。后续的 parser 模块会依赖这些 token 生成语法树，进而执行脚本逻辑。

3.3.2.Parser

parser 模块在项目中承担着将词法分析器（lexer）生成的词法单元转换为语法树（AST）的关键角色。通过解析脚本语言的语法规则，parser 模块能够理解脚本的结构和逻辑，为解释器（interpreter）提供可执行的指令集。

以下是对 parser 子模块设计的简要介绍。

设计目标

- **构建准确的语法树**：parser 模块旨在将词法分析器输出的词法单元精确地组织成语法树，反映脚本的层次结构和逻辑关系。这一过程确保脚本的语法结构符合预定的语言规范。
- **支持多种脚本结构**：设计支持脚本中的各种语法结构，如步骤定义、动作指令（如 Speak、Listen、Goto 等）、条件分支（Case、Default）以及变量操作（Recharge、Consume）。模块应能够灵活处理不同复杂度的脚本内容。
- **错误检测与处理**：在解析过程中，parser 模块需要能够检测并报告语法错误，提供有意义的错误信息，帮助用户快速定位和修正脚本中的问题。
- **模块化与可扩展性**：设计遵循模块化原则，确保 parser 模块的各部分功能清晰分明，便于未来的扩展和维护。例如，新增新的动作指令或语法结构时，只需在现有框架内进行扩展，而不需大幅修改现有代码。

主要功能

- **步骤定义解析**：识别并解析脚本中的每个步骤（Step），提取步骤名称和相关的动作指令。每个步骤作为语法树中的一个节点，包含其执行的具体动作。
- **动作指令解析**：解析脚本中的各种动作指令，包括但不限于 Speak、Listen、Goto、Recharge、Consume、GetNum 等。每个指令根据其语法规则被解析为语法树中的子节点，反映其参数和操作。
- **表达式解析**：处理脚本中的表达式，如字符串拼接、变量操作等。通过解析这些表达式，构建语法树中相应的表达式节点，为解释器提供准确的执行逻辑。
- **语法错误报告**：在解析过程中，若遇到不符合语法规则的脚本内容，parser 模块能够捕捉并报告具体的错误信息，包括错误位置和原因，帮助用户快速修复脚本。

3.3.3.Interpreter

Interpreter 模块在项目中负责根据解析器（parser）生成的语法树执行脚本逻辑。该模块将脚本的语法结构转化为具体的操作指令，管理执行状态和变量，处理用户交互，确保脚本按照预定的流程正确运行。

以下是对 Interpreter 子模块设计的简要介绍。

设计目标

- **执行脚本逻辑**：Interpreter 模块的主要目标是根据解析器生成的语法树，逐步执行脚本中的各个步骤和动作，确保脚本逻辑的正确性和一致性。
- **变量管理**：管理脚本中的变量，包括变量的初始化、更新和引用，确保变量在不同步骤和动作中的正确使用。
- **步骤和动作调度**：处理脚本中的各个步骤（Step）及其包含的动作指令（如 Speak、Listen、Goto 等），实现步骤之间的跳转和动作的顺序执行。
- **用户交互处理**：模拟用户输入和响应，处理用户在脚本执行过程中可能的交互操作，确保脚本能够根据用户输入做出相应的反应。
- **并发用户交互**：支持同一个脚本文件中多个用户交互上下文的独立处理，确保不同用户的交互不会相互干扰，提高系统的灵活性和可扩展性。
- **错误处理与退出机制**：在脚本执行过程中，能够处理各种错误情况，并提供合理的退出机制，确保脚本能够安全终止或跳转到指定步骤。

主要功能

- **脚本执行循环**：run 方法作为脚本执行的主循环，负责按照步骤顺序执行脚本，直到所有步骤完成或脚本被显式退出。
- **步骤执行**：execute_step 方法负责执行当前步骤中的所有动作指令，包括输出文本、等待用户输入、变量操作和跳转步骤等。
- **动作指令处理**：execute_action 方法根据动作类型调用相应的执行方法，如 execute_speak、execute_listen、execute_goto、execute_recharge、execute_consume 等，确保每个动作按预定逻辑正确执行。
- **变量替换**：replace_variables 方法负责在输出文本中替换变量为其当前值，支持脚本中的动态内容显示。

4. 核心模块实现

4.1. Lexer 词法分析器

4.1.1.实现方法

lexer.py 模块使用 pyparsing 库来定义和解析词法单元。以下是词法分析器的主要实现步骤：

1. **关键词定义：**

```
keywords = {
    "Step": pp.Keyword("Step"),
    "Speak": pp.Keyword("Speak"),
    "Listen": pp.Keyword("Listen"),
    "Case": pp.Keyword("Case"),
    "Default": pp.Keyword("Default"),
    "Exit": pp.Keyword("Exit"),
    "add": pp.Keyword("add"),
    "reduce": pp.Keyword("reduce"),
    "goto": pp.Keyword("goto"),
    "Timeout": pp.Keyword("Timeout"),
    "Recharge": pp.Keyword("Recharge"),
    "Consume": pp.Keyword("Consume"),
    "GetNum": pp.Keyword("GetNum"),
}
```

定义了一组保留关键词，用于识别脚本中的控制结构和操作指令。

2. **标识符和变量定义：**

```
identifier = pp.Word(pp.alphas + "_", pp.alphas + pp.nums +
  "_").addParseAction(lambda t: None if t[0] in keyword_names else t[0])

variable = pp.Combine(pp.Literal('$') + pp.Word(pp.alphas + "_",
pp.alphas + pp.nums + "_"))
```

- identifier 用于匹配变量名、步骤名等标识符，排除与关键词重复的情况。
- variable 用于匹配以 $ 开头的变量名。

3. **符号定义：**

```
string = pp.QuotedString('"')
number = pp.Word(pp.nums)
```

```
3. plus_operator = pp.Literal('+')
4. comma = pp.Literal(',')
```

定义了字符串、数字、加号操作符和逗号等基本符号。

4. **词法单元组合：**

```
 1. tokens = {
 2.     "identifier": identifier,
 3.     "variable": variable,
 4.     "string": string,
 5.     "number": number,
 6.     "plus_operator": plus_operator,
 7.     "comma": comma,
 8. }
 9.
10. all_tokens = list(keywords.values()) + list(tokens.values())
```

将所有定义的词法单元（包括关键词和其他符号）组合在一起，形成完整的词法分析规则。

5. **词法分析器函数：**

```
1. def lex_input(input_text):
2.     result = []
3.     match_first = pp.MatchFirst(all_tokens)
4.     result.extend(match_first.scanString(input_text))
5.     return result
```

lex_input 函数使用 MatchFirst 确保关键词优先匹配，然后扫描输入文本，返回所有匹配的词法单元。

4.1.2.数据结构

词法分析器主要使用以下数据结构：

- **字典（Dictionary）：**
  - keywords：存储所有的保留关键词，便于快速查找和匹配。
  - tokens：存储其他词法单元，如标识符、变量、字符串等。
- **集合（Set）：**
  - keyword_names：存储关键词的名称，用于在标识符解析时排除关键词。
- **列表（List）：**
  - all_tokens：存储所有的词法单元，供 MatchFirst 使用。

4.1.3.词法分析流程

词法分析器的工作流程如下：

1. **输入文本**：接收用户输入的脚本文本。
2. **词法单元匹配**：使用预定义的词法规则，通过 pyparsing 库解析文本，将其分解为关键词、标识符、变量、字符串、数字等基本单元。
3. **生成词法单元流**：lex_input 函数返回一个词法单元流，供语法分析器（Parser）使用。

4.2. Parser 语法分析器

4.2.1.实现方法

parser.py 模块同样采用 pyparsing 库来定义和解析脚本语言的语法规则。以下是语法分析器的主要实现步骤：

1. **导入必要的组件：**

```
1. import pyparsing as pp
2. from src.lexer import identifier, variable, string, number,
keywords, comma, plus_operator
```

语法分析器依赖于词法分析器定义的词法单元，确保词法和语法的一致性。

2. **定义语法规则：**

- **步骤定义：**

```
1. step_name = keywords["Step"] + identifier("step_name")
```

每个步骤以 Step 关键词开头，后跟步骤名称。

- **表达式规则：**

```
1. expression = (string | variable)
```

处理脚本中的字符串和变量，支持后续的拼接操作。

- **动作指令定义：**

包括 Speak、Listen、Goto、Timeout、Recharge、Consume、GetNum、Case、Default 和 Exit 等动作指令的语法规则。

例如，Speak 动作的定义：

```
1. speak_action = keywords["Speak"] + pp.OneOrMore(expression +
pp.Optional(plus_operator))("speak_text")
```

- **动作组合：**

```
1. action = pp.Group(speak_action | listen_action | case_action |
default_action | goto_action | recharge_action | consume_action |
exit_action | timeout_action | getNum_action)
```

使用 Group 将不同类型的动作指令组合在一起，便于统一处理。

- **步骤定义组合：**

```
1. step = pp.Group(step_name + pp.ZeroOrMore(action) +
pp.Optional(pp.LineEnd()))("step")
```

每个步骤包含一个步骤名称和一系列动作指令。

- **完整脚本定义：**

```
script = pp.OneOrMore(step + pp.Optional(pp.LineEnd()))("script")
```

整个脚本由多个步骤组成。

3. **解析函数：**

```
def parse_script(input_text):
    try:
        parsed = script.parseString(input_text)
        return parsed
    except pp.ParseException as pe:
        print(f"Parsing error: {pe}")
        return None
```

parse_script 函数负责接收脚本文本并返回解析后的语法树。如果解析失败，将捕捉并输出错误信息。

4.2.2.数据结构

语法分析器主要使用以下数据结构：

- **树状结构（Syntax Tree）：**
  - 语法分析器生成的语法树（AST）以树状结构表示脚本的层次关系和逻辑流程。每个节点代表一个语法元素，如步骤、动作指令等。
- **字典（Dictionary）：**
  - 在解析过程中，使用字典来存储动作指令的属性。例如，step_name、speak_text、goto_step 等，便于后续处理和引用。

4.2.3.语法树结构

语法分析器生成的语法树（AST）具有以下结构特点：

1. **根节点：**
   - script：表示整个脚本，由多个 step 组成。
2. **步骤节点：**
   - step：每个 step 节点包含一个步骤名称和一系列动作指令。
3. **动作指令节点：**
   - 每个动作指令（如 Speak、Listen、Goto 等）作为子节点存在于对应的 step 节点下，包含其特定的属性和参数。
4. **属性节点：**
   - 动作指令的属性（如 speak_text、goto_step、timeout_amount 等）作为键值对存储在动作指令节点中，便于解释器后续的执行。

示例语法树：

以测试用例 1 为例：

```
Step welcome
    Speak "尊敬的勇者，欢迎来到冒险者协会，请问有什么可以帮您?"
    Listen 5, 30
    Case "投诉"
        goto complainProc
    Case "查询"
        goto billProc
    Case "完成冒险"
        goto rechargeProc
    Case "消费"
        goto consumeProc
    Case "离开"
        goto thanks
    Default
        goto wait
    Timeout 15
        Speak "哈喽您还活着吗？"
Step rechargeProc
    Recharge $balance add 100
    Speak "验证成功！您的账户当前余额是" + $balance + "枚金币"
```

解析后的语法树结构如下：

```
['Script',
          ['Step', 'welcome',    ['Speak', ['"尊敬的勇者，欢迎来到冒险者协会，请问有什么可以帮您?"']],
                    ['Listen', '5', '30'],
                    ['Case', '"投诉"', 'goto', 'complainProc'],
                    ['Case', '"查询"', 'goto', 'billProc'],
                    ['Case', '"完成冒险"', 'goto', 'rechargeProc'],
                    ['Case', '"消费"', 'goto', 'consumeProc'],
                    ['Case', '"离开"', 'goto', 'thanks'],
                    ['Default', 'goto', 'wait'],
                    ['Timeout', '15', 'Speak', ['"哈喽您还活着吗？"']]
          ],
          ['Step', 'rechargeProc',    ['Recharge', '$balance', 'add', '100'],
                    ['Speak', ['"验证成功！您的账户当前余额是"', '$balance', '"枚金币"']]
          ]
]
```

该语法树清晰地展示了脚本的结构，每个步骤包含多个动作指令，指令之间的层级关系明晰，便于解释器进行后续处理。

4.2.4.解析流程

语法分析器的工作流程如下：

1. **输入文本接收：**
- 接收用户输入的脚本文本，作为解析的原始数据。
2. **词法分析结果获取：**
- 通过调用词法分析器（Lexer），将脚本文本分解为一系列词法单元（tokens）。
3. **语法规则匹配：**
- 使用预定义的语法规则，通过 pyparsing 库对词法单元进行匹配和组织，构建语法树。
4. **语法树生成：**
- 将匹配成功的语法单元组织成树状结构（AST），反映脚本的逻辑层级和执行流程。
5. **错误处理：**
- 在匹配过程中，如果遇到不符合语法规则的内容，捕捉并报告语法错误，提示用户进行修正。

4.3. Interpreter 解释器

4.3.1.实现方法

interpreter.py 模块采用面向对象的设计，通过 Interpreter 类来管理脚本的执行流程。以下是解释器的主要实现步骤和设计要点：

1. **导入必要的组件：**

```
1. import time
2. import re
3. from src.parser import parse_script
```

解释器依赖于语法分析器提供的解析结果（语法树），并使用标准库中的 time 和 re 模块来处理时间控制和正则表达式操作。

2. **Interpreter 类定义：**

```
1. class Interpreter:
2.     def __init__(self, script):
3.         self.script = script
4.         self.variables = {}
5.         self.step_index = self.find_step_index("welcome")
6.         self.case_actions = []
7.         self.has_jumped = False
```

初始化方法：接收解析后的脚本语法树，初始化变量存储字典、当前步骤索引（默认从 "welcome" 步骤开始）、当前步骤的 Case 动作列表以及跳转标志。

3. **执行流程：**

```
1. def run(self):
2.     while self.step_index != -1:
3.         step = self.script[self.step_index]
4.         step_name = step["step_name"]
5.         self.update_case_actions(step)
6.         self.execute_step(step)
```

主执行循环：持续执行当前步骤，直到所有步骤完成或脚本显式退出（通过 Exit 动作）。

4. **步骤和动作执行：**

```
def execute_step(self, step):
    actions = step[1:]
    for action in actions:
        self.execute_action(action)
        if self.has_jumped:
            break
    self.has_jumped = False
```

执行当前步骤的所有动作：逐一执行步骤中的每个动作指令。如果发生跳转（如 goto 动作），则中断当前步骤的执行，转而执行跳转后的步骤。

5. **动作指令处理：**

```
def execute_action(self, action):
    if "timeout_amount" in action:
        self.execute_timeout(action)
    elif "speak_text" in action:
        self.execute_speak(action)
    elif "listen_start" in action:
        self.execute_listen(action)
    elif "goto_step" in action:
        self.execute_goto(action)
    elif "recharge_var" in action:
        self.execute_recharge(action)
    elif "consume_var" in action:
        self.execute_consume(action)
    elif "getNum_var" in action:
        self.execute_get_num(action)
    elif "case_input" in action:
        self.execute_case(action)
    elif "default" in action:
        self.execute_default(action)
    elif "Exit" in action:
        self.execute_exit(action)
```

动作类型判断：根据动作指令的类型，调用相应的执行方法，如 execute_speak、execute_listen 等。

6. **具体动作执行方法：**

解释器为每种动作指令定义了专门的执行方法，以确保每个动作按照预定逻辑正确执行。例如：

Speak 动作：

```
def execute_speak(self, action):
    speak_text = ''.join(action["speak_text"])
    speak_text = self.replace_variables(speak_text)
    speak_text = speak_text.replace("+", "")
    print(f"{speak_text}")
```

- **功能**：输出脚本指定的文本内容。
- **变量替换**：调用 replace_variables 方法，将文本中的变量替换为其当前值。

Listen 动作：

```
def execute_listen(self, action):
    listen_start = int(action["listen_start"])
    listen_stop = int(action["listen_stop"])
    listen_duration = listen_stop - listen_start

    start_time = time.time()
    user_input = None

    while time.time() - start_time < listen_duration:
        print("请回复: ")
        user_input = input()

        if time.time() - start_time >= listen_duration:
            print("您似乎不在了....")
            self.execute_default(action)
            return

        if user_input:
            break

        print("我在等您的回复...")

    case_matched = False
    for case_action in self.case_actions:
        case_input = case_action["case_input"]
        if case_input in user_input:
            self.step_index =
self.find_step_index(case_action['goto_step'])
```

```
            case_matched = True
            self.has_jumped = True
            break

    if not case_matched:
        print("我不明白您在说什么。请再说一遍好吗？")
        self.execute_listen(action)
```

- **功能**：模拟等待用户输入，并根据输入内容执行相应的跳转动作。
- **超时处理**：如果用户在指定时间内未输入，执行默认操作。

Goto 动作：

```
def execute_goto(self, action):
    goto_step = action["goto_step"]
    self.step_index = self.find_step_index(goto_step)
    self.has_jumped = True
```

- **功能**：跳转到脚本中的指定步骤。

Recharge 动作：

```
def execute_recharge(self, action):
    var = action["recharge_var"].lstrip('$')
    if '$' in action["recharge_amount"]:
        amount_var = action["recharge_amount"].lstrip('$')
        amount = self.variables[amount_var]
    else:
        amount = int(action["recharge_amount"])
    if var not in self.variables:
        self.variables[var] = 100
    self.variables[var] += amount
```

- **功能**：增加指定变量的值，用于管理账户余额等场景。

Consume 动作：

```
def execute_consume(self, action):
    var = action["consume_var"].lstrip('$')
    amount = int(action["consume_amount"])
    if var not in self.variables:
        self.variables[var] = 100
    self.variables[var] -= amount
```

- **功能**：减少指定变量的值，用于消费场景。

GetNum 动作：

```
def execute_get_num(self, action):
    print("请回复： ")
    user_input = input()
    var = action["getNum_var"].lstrip('$')
    self.variables[var] = int(user_input)
```

- **功能**：获取用户输入的数字并存储到指定变量中。

Case 和 Default 动作：

```
def execute_case(self, action):
    case_input = action["case_input"]
    user_input = input(f"请回复 {case_input}: ")
    if case_input in user_input:
        self.step_index = self.find_step_index(user_input)
    else:
        print("我不明白您在说什么。请再说一遍好吗？ ")
        self.execute_listen(action)

def execute_default(self, action):
    current_step_name = self.script[self.step_index]["step_name"]
    if current_step_name == "welcome":
        self.step_index = self.find_step_index("wait")
    else:
        self.step_index = self.find_step_index("welcome")
    self.has_jumped = True
```

- **功能**：根据用户输入的条件执行相应的跳转动作，或执行默认操作。

Exit 动作：

```
def execute_exit(self, action):
    exit(0)
```

- **功能**：安全地终止脚本执行。

Timeout 动作：

```
def execute_timeout(self, action):
    timeout_duration = int(action["timeout_amount"])
    time.sleep(timeout_duration)
    if "speak_text" in action:
        self.execute_speak(action)
    elif "goto_step" in action:
        self.execute_goto(action)
```

功能：处理超时操作，执行指定的动作指令。

7. **辅助方法：**

查找步骤索引：

```
def find_step_index(self, step_name):
    for i, step in enumerate(self.script):
        if step["step_name"] == step_name:
            return i
    return -1
```

- **功能**：根据步骤名称查找其在语法树中的索引位置，便于跳转执行。

变量替换：

```
def replace_variables(self, text):
    pattern = r'\$([a-zA-Z_][a-zA-Z0-9_]*)'
    matches = re.findall(pattern, text)

    for match in matches:
        if match not in self.variables:
            self.variables[match] = 100
        text = text.replace(f"${match}", str(self.variables[match]))

    return text
```

- **功能**：将文本中的变量替换为其当前值，确保动态内容的正确显示。

4.3.2.语法树执行流程

解释器通过以下流程执行解析后的语法树：

1. **初始化：**
    - 通过 Interpreter 类的实例化，传入解析后的脚本语法树，初始化变量存储和执行状态。
2. **执行循环：**
    - run 方法启动主执行循环，根据 step_index 持续执行当前步骤，直至脚本结束或被显式退出。
3. **步骤执行：**
    - execute_step 方法负责执行当前步骤中的所有动作指令。
    - 每个动作指令通过 execute_action 方法调用相应的执行方法，实现具体功能。
4. **动作处理：**
    - 各种动作指令（如 Speak、Listen、Goto 等）由专门的方法处理，确保每个动作按预定逻辑正确执行。
    - **用户交互**：通过 Listen 和 Case 动作模拟用户输入，并根据输入内容执行条件跳转。
    - **变量管理**：通过 Recharge 和 Consume 动作管理变量值，支持动态变量操作。
    - **流程控制**：通过 Goto 和 Timeout 动作实现脚本流程的灵活跳转和时间控制。
5. **跳转与退出：**
    - 通过 goto_step 动作指令或用户交互决定脚本的跳转步骤。
    - 通过 Exit 动作指令或脚本执行完毕终止脚本执行。

5. 测试设计与分析

5.1. 测试工具与环境

本项目使用了以下工具和环境来支持测试活动：

- **测试框架**：采用 unittest 作为主要的测试框架，提供结构化的测试用例管理和执行。
- **模拟工具**：利用 unittest.mock 模块模拟用户输入和外部依赖，确保测试的独立性和可控性。

5.2. 测试桩 test_lexer.py

5.2.1.主要内容

test_lexer.py 测试桩主要负责验证词法分析器在各种输入场景下的解析能力。其主要内容包括：

- **空输入测试**：确保词法分析器能够正确处理空白输入，不产生任何词法单元。
- **基本脚本测试**：验证词法分析器是否能够识别脚本中的常见关键词、标识符、变量和符号。
- **忽略空白字符测试**：测试词法分析器在处理包含多余空白字符的脚本时，是否能够正确提取词法单元。
- **标识符与关键词区分测试**：确保词法分析器能够区分保留关键词与用户定义的标识符，避免混淆。
- **变量与拼接符号测试**：验证词法分析器对变量和拼接操作符的识别能力，确保变量以 $ 开头且拼接符号 + 被正确解析。

5.2.2.测试设计

test_lexer.py 的测试设计采用了以下策略和方法，以全面覆盖词法分析器的各项功能：

1. **使用 unittest 框架：**

- 采用 Python 内置的 unittest 框架构建测试用例，提供结构化的测试管理和执行环境。

2. **模拟 pyparsing 行为：**

- 利用 unittest.mock 模块中的 MagicMock 对象，模拟 pyparsing 库中 MatchFirst 和 scanString 方法的行为。这种方法能够隔离词法分析器的测试环境，确保测试结果不受外部依赖影响。

3. **独立测试各类输入：**

- 每个测试方法针对特定的输入场景设计，确保词法分析器在各种情况下的表现均符合预期。

4. **断言验证：**

- 使用 assert 方法验证词法分析器的输出是否符合预期，包括词法单元的数量和内容。

5. **边界条件测试：**

- 设计测试用例涵盖边界条件和异常输入，如空输入、非法字符和未闭合的字符串，验证词法分析器的鲁棒性和错误处理能力。

具体测试用例设计如下：

- **测试空输入** (test_lex_input_empty)：
    - 输入为空白文本，期望词法分析器返回空的词法单元列表。
- **测试基本脚本** (test_lex_input_basic_script)：
    - 输入包含多个关键词、变量和符号的标准脚本，验证词法单元的正确提取。
- **测试忽略空白字符** (test_lex_input_ignore_whitespace)：
    - 输入包含多余空白字符的脚本，确保词法分析器能够正确识别并提取有效词法单元。
- **测试标识符与关键词的区分** (test_lex_input_identifier_vs_keyword)：

    - 输入包含与关键词相似的标识符，验证词法分析器能够正确区分保留关键词与用户定义的标识符。
- **测试变量与拼接符号** (test_lex_input_variable_and_operator)：
    - 输入包含变量和拼接操作符的脚本，确保变量以 $ 开头且拼接符号 + 被正确识别。

5.2.3.结果分析

五个模拟测试均通过，验证了词法分析器在各种输入场景下的解析能力和鲁棒性。通过模拟 pyparsing 的行为，确保了测试环境的可控性和独立性。

测试结果表明，词法分析器能够准确提取脚本中的关键词、标识符、变量和符号，具备良好的错误处理能力，满足项目的设计要求。

5.3. 测试桩 test_parser.py

5.3.1.主要内容

test_parser.py 测试桩的主要职责是验证语法分析器在各种输入场景下的解析能力，确保其能够准确地将词法分析器生成的词法单元（tokens）转换为语法树（AST）。具体内容包括：

- **基本脚本解析**：验证语法分析器是否能够正确解析包含基本关键词和动作指令的脚本。
- **复杂动作解析**：测试包含复杂动作指令（如 Recharge、goto 等）的脚本，确保语法树结构的正确性。
- **错误输入处理**：验证语法分析器在遇到无效输入时的错误检测与处理能力，确保其能够识别并报告语法错误。

5.3.2.测试设计

test_parser.py 的测试设计采用了以下策略和方法，以全面覆盖语法分析器的各项功能：

1. **使用 unittest 框架：**

- 采用 Python 内置的 unittest 框架构建测试用例，提供结构化的测试管理和执行环境。

2. **模拟词法分析器（Lexer）的行为：**

- 利用 unittest.mock 模块中的 patch 装饰器，模拟词法分析器中各词法单元（如 identifier、variable、string 等）的解析行为。这种方法能够隔离语法分析器的测试环境，确保测试结果不受实际词法分析器实现的影响。

3. **独立测试各类输入：**

- 每个测试方法针对特定的输入场景设计，确保语法分析器在各种情况下的表现均符合预期。

4. **断言验证：**

- 使用 assert 方法验证语法分析器的输出是否符合预期，包括语法树的结构

和内容。

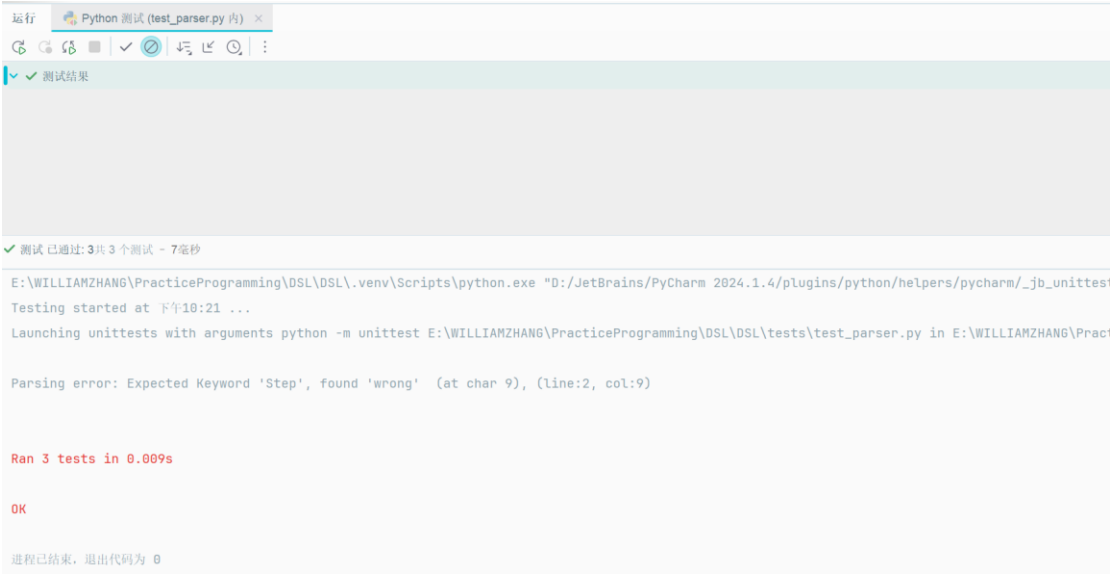
5. **边界条件与异常输入测试：**

- 设计测试用例涵盖边界条件和异常输入，如无效的语法结构，验证语法分析器的鲁棒性和错误处理能力。

具体测试用例设计如下：

- **测试用例 1**：包含 Step、Speak、Listen、Case 和 Default 等基本语法元素的脚本，验证语法分析器能否正确生成语法树。
- **测试用例 2**：包含 Recharge 动作的脚本，测试语法分析器对变量操作指令的解析能力。
- **测试用例 3**：包含无效输入的脚本，验证语法分析器能否正确检测并报告语法错误。

5.3.3.结果分析



所有设计的测试用例均通过，验证了语法分析器在各种输入场景下的解析能力和错误处理机制。通过模拟词法分析器的行为，确保了测试环境的可控性和独立性。

测试结果表明，语法分析器能够准确构建语法树，反映脚本的逻辑结构，并在遇到无效输入时能够有效地进行错误报告。

5.4. 测试桩 test_interpreter.py

5.4.1.主要内容

test_interpreter.py 测试桩的主要职责是验证解释器在各种脚本执行场景下的功能正确性和稳定性。其主要内容包括：

- **基本脚本执行测试**：验证解释器是否能够正确执行包含基本动作指令（如 Speak、Listen、Case、Goto 和 Exit）的脚本，并按照预期的流程跳转和终止。
- **变量管理与替换测试**：测试解释器对变量的初始化、更新和引用能力，确保变量操作（如 Recharge 和 Consume）能够正确反映在脚本执行过程中。
- **用户交互模拟测试**：通过模拟用户输入，验证解释器在处理 Listen 和 Case 指令时的响应能力和逻辑跳转正确性。
- **错误处理与流程控制测试**：确保解释器在遇到异常情况（如无效步骤跳转）时能够正确处理，避免系统崩溃或进入无限循环。

5.4.2.测试设计

test_interpreter.py 的测试设计采用了以下策略和方法，以全面覆盖解释器的各项功能：

1. **使用 unittest 框架：**

- 采用 Python 内置的 unittest 框架构建测试用例，提供结构化的测试管理和执行环境。

2. **模拟用户输入：**

- 利用 unittest.mock 模块中的 patch 装饰器，模拟 builtins.input 函数的行为，以模拟用户在脚本执行过程中的输入。

3. **模拟变量替换方法：**

- 使用 patch.object 装饰器重写解释器中的 replace_variables 方法，确保变量替换过程的可控性和可预测性。这有助于验证解释器在变量管理和文本输出方面的正确性。

4. **捕获和验证输出：**

- 通过 patch('sys.stdout', new_callable=StringIO) 模拟标准输出，捕获解释器的打印输出，并进行断言验证，以确保解释器的输出符合预期。

5. **独立测试各类脚本：**

- 每个测试方法针对特定的脚本场景设计，确保解释器在不同情况下的表现均符合预期。

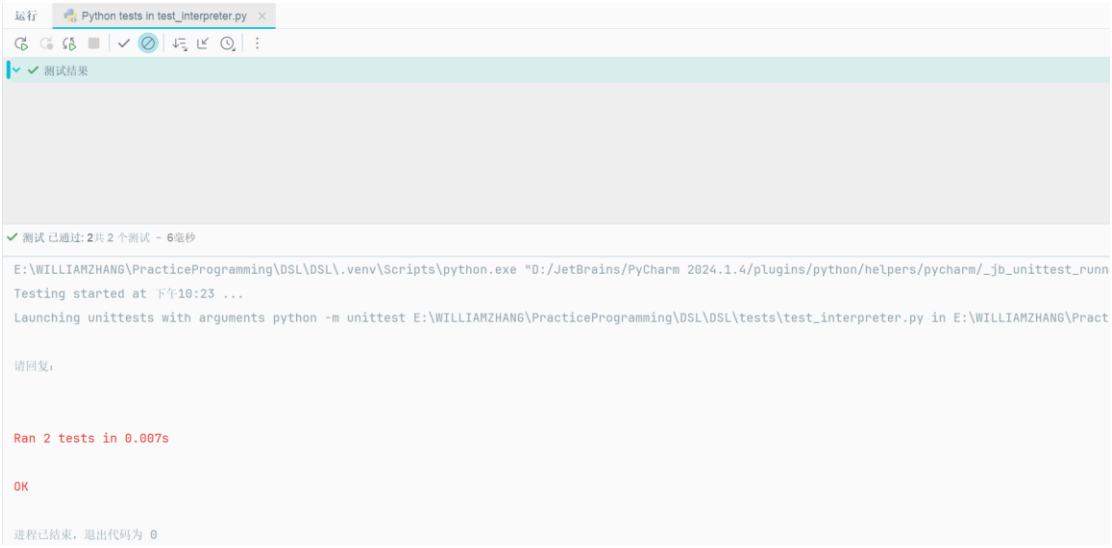
6. **断言验证：**

- 使用 assert 方法验证解释器的执行状态（如 step_index）、变量值和输出内容是否符合预期。

具体测试用例设计如下：

- **测试用例 1**：模拟用户依次输入 "投诉"、"没有"、"提交"、"买药水" 和空字符串，验证解释器是否能够按照脚本逻辑跳转到相应步骤并最终终止执行。
- **测试用例 2**：模拟用户依次输入 "查询" 和 "离开"，并重写变量替换方法，验证解释器在变量管理和文本输出方面的正确性，确保变量被正确替换且输出符合预期。

5.4.3.结果分析



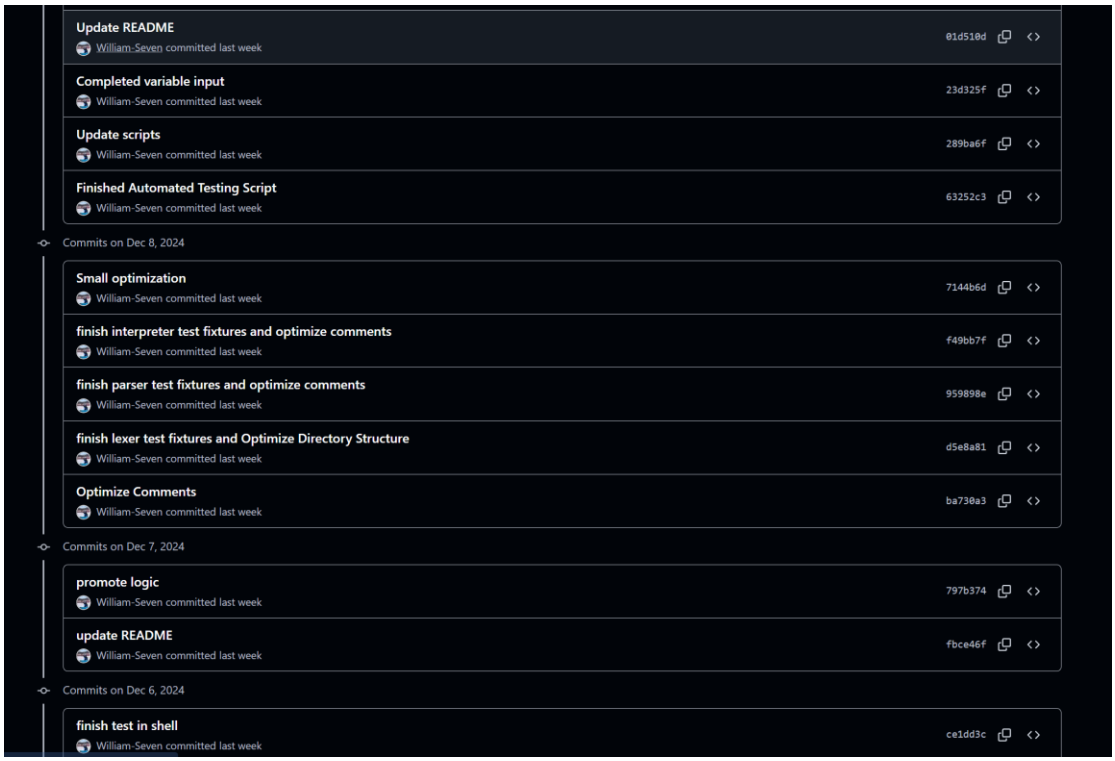
通过模拟用户输入和变量替换功能，测试桩确保了解释器在不同脚本执行场景下的可靠表现。测试结果表明，解释器能够准确执行脚本逻辑，正确管理变量，灵活处理用户交互，并按照预期进行流程控制。

6. 开发方法

6.1. 版本管理

我使用 Git 连接 GitHub 进行版本管理。即使是单人开发，使用版本控制系统也更加利于管理修改历史、回滚和多分支开发。

下图展示一些开发过程中的版本管理：



6.2. 代码规范与风格

为确保项目代码的一致性和高质量，本项目严格遵循了 Python 社区广泛认可的 **PEP8** 标准。

以下将详细介绍 PEP8 标准的主要内容及其在本项目中的具体应用。

6.2.1.PEP8 标准概述

PEP8（Python Enhancement Proposal 8）是 Python 社区制定的一套编码规范，旨在提高 Python 代码的可读性和一致性。PEP8 涵盖了代码格式、命名约定、注释风格等多个方面，具体包括但不限于以下内容：

- **缩进**：推荐使用 4 个空格进行缩进，禁止使用制表符（Tab）。
- **行长度**：每行代码的长度不超过 79 个字符，长字符串或注释可适当延长至 99 个字符。
- **空行**：模块级函数和类定义之间使用两个空行，类内方法定义之间使用一个空行。
- **导入顺序**：标准库导入、第三方库导入和本地库导入分别分组，并按字母顺序排列，每组之间用空行隔开。
- **空格使用**：避免在逗号、分号、冒号、括号等符号前添加空格；操作符两侧应各添加一个空格（如赋值、比较操作符）。
- **命名约定**：
    - **变量名**：使用小写字母和下划线（如 user_name）。
    - **函数名**：使用小写字母和下划线（如 calculate_total）。
    - **类名**：采用驼峰命名法，每个单词的首字母大写（如 Interpreter）。
    - **常量名**：使用全大写字母和下划线分隔（如 MAX_RETRIES）。
- **注释**：代码中应包含必要的注释，注释应清晰、简洁，解释代码的意图和逻辑。块注释和行注释应分别使用适当的格式。
- **文档字符串**：每个模块、函数和类应包含文档字符串（docstring），描述其功能、参数和返回值。

6.2.2.PEP8 在本项目中的应用

本项目在开发过程中严格遵循 PEP8 标准，以确保代码的高质量和一致性。具体应用包括：

1. **统一的缩进和行长度：**

- 全部代码文件均采用 4 个空格进行缩进，避免混用制表符和空格。
- 代码行长度控制在 79 个字符以内，长表达式和注释适当延长，但不超过 99 个字符，确保代码在不同编辑器和显示设备上的良好显示效果。

2. **严格的空格使用规范：**

- 操作符两侧添加空格，如赋值操作符（=）、比较操作符（==、!=）等。
- 函数参数和括号内不添加多余空格。

例如：

```
def execute_goto(self, action):
    goto_step = action["goto_step"]
    self.step_index = self.find_step_index(goto_step)
    self.has_jumped = True
```

3. **一致的命名约定：**

- 类名采用驼峰命名法，确保每个单词的首字母大写，如 Interpreter。
- 函数名和变量名采用小写字母和下划线分隔，如 execute_goto、step_index。

4. **清晰的注释和文档字符串：**

- 每个函数和类均包含文档字符串，详细描述其功能、参数和返回值。
- 代码中的关键逻辑和复杂部分添加行注释，解释代码的意图和实现细节。

例如：

```
class Interpreter:
    def __init__(self, script):
        """
        初始化解释器实例

        :param script: 解析后的脚本语法树
        """
        self.script = script  # 解析后的脚本语法树
        self.variables = {}  # 用于存储变量的字典
        self.step_index = self.find_step_index("welcome")  # 从 Step welcome 开始执行
```

```
        self.case_actions = []  # 声明用于存储当前步骤的 Case 动作
        self.has_jumped = False  # 跳转标志，初始为 False
```

5. **模块化和函数化设计：**

- 代码结构清晰，函数职责单一，每个函数完成特定的任务，便于理解和维护。
- 避免冗长的函数和类，确保代码的可读性和可维护性。

7. 心得总结

在本项目的开发过程中，我深刻体会到了软件开发不仅仅是编写代码的过程，更是一个系统化、结构化的工程。

通过构建词法分析器（Lexer）、语法分析器（Parser）和解释器（Interpreter）等核心模块，并配合完善的测试设计与代码规范，我获得了宝贵的经验和深刻的认识。

首先，在学习和设置测试桩的过程中，我认识到测试在确保代码质量和功能正确性方面的重要性。测试桩的设计不仅帮助我隔离和独立验证每个模块的功能，还提高了测试的效率，使得错误定位更加精准。此外，模拟用户输入和变量替换的测试方法，使我能够全面覆盖各种执行场景，确保解释器在不同条件下的稳定性和可靠性。

通过此次项目，我深刻理解到开发过程与单纯写代码的区别。

项目开发不仅涉及代码的实现，还包括需求分析、系统设计、模块划分、测试验证和文档撰写等多个环节。详细的需求分析和设计规划使得开发过程有条不紊，减少了返工和修改的可能性。

在代码规范与开发方法方面，我严格遵循了 Python 社区广泛认可的 PEP8 标准，确保了代码的一致性和高可读性。

总的来说，本项目不仅提升了我的技术能力，还培养了我系统化思考和工程化实践的能力。这些宝贵的经验为我未来的开发工作奠定了坚实的基础，使我更加注重代码质量、测试覆盖和团队协作，确保在未来的项目中能够高效、稳定地实现预期目标。