

基于领域特定脚本语言的客服机器人的设计与实现

班级：2022211304 学号：2022211119 姓名：赵宇鹏

1 概述

1.1 要求

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着广泛的应用）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 简介

本项目设计了一个领域特定语言，这个语言通过一种相对简单的文法，直接的逻辑，能够描述在线客服机器人的自动应答逻辑，同时也为之设计实现了一个解释器来解释执行这个脚本，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，同时本项目设计了相应的GUI页面来实现和用户的交互。

2 程序设计介绍

2.1 语法介绍

首先给出所定义语法的BNF范式：

```
1  <DSL> ::= <StepList>
2  <StepList> ::= <Step> | <Step> <StepList>
3  <Step> ::= "Step" <Identifier> "Begin" <StatementList> "End"
4  <StatementList> ::= <Statement> | <Statement> <StatementList>
5  <Statement> ::= <SpeakStatement>
6                  | <ListenStatement>
7                  | <SwitchStatement>
8                  | <UpdateStatement>
9                  | <ExitStatement>
10 <SpeakStatement> ::= "Speak" <Expression>
11 <ListenStatement> ::= "Listen" <Variable>
12 <SwitchStatement> ::= "Switch" <Variable> "Begin" <BranchList>
    <DefaultBranch> "End"
13 <BranchList> ::= <Branch> | <Branch> <BranchList>
14 <Branch> ::= "Branch" <StringLiteral> "Run" <Step>
15 <DefaultBranch> ::= "Default" "Run" <StepReference>
16 <UpdateStatement> ::= "Update" <Variable> "=" <Expression>
17 <ExitStatement> ::= "Exit"
18 <Expression> ::= <StringLiteral>
19                | <Variable>
```

```

20 | <Expression> ::= <StringLiteral> "+" <Expression>
21 | <Variable> ::= "$" <Identifier>
22 | <Identifier> ::= <Letter> <IdentifierTail>
23 | <IdentifierTail> ::= <LetterOrDigit> <IdentifierTail> | ε
24 | <StringLiteral> ::= "'" <StringContent> "'"
25 | <StringContent> ::= <Character> <StringContent> | ε
26 | <Letter> ::= [_a-zA-Z]
27 | <LetterOrDigit> ::= [a-zA-Z0-9]
28 | <Character> ::= [^"\\] | '\\' ["\\]

```

而脚本更具体的要求和规则如下：

2.1.1 关键字

- **Step**：用于定义步骤或过程。
- **Begin**：标记过程结构或分支结构的开始。
- **End**：标记过程结构或分支结构的结束。
- **Listen**：用于输入操作，默认等待时间为 10 秒。如果在超时后未输入内容，则认为用户已经断开通话，程序自动结束。
- **Speak**：用于输出内容，可以输出字符串、变量，或两者的组合。
- **Switch**：用于定义分支结构。
- **Branch**：定义具体分支的关键字。
- **Update**：用于更新符号表中的符号的值。
- **Default**：指定分支结构中的默认分支，当无分支条件匹配时执行。
- **Run**：执行其他步骤的指令。
- **Exit**：退出程序的指令，表示程序运行的终止。
- **\$**：变量标识符，类似于取址符，表示引用变量。
- **#**：单行注释的标识符。
- **@**：多行注释标识符，两个 @ 包围的内容为注释。

2.1.2 语法结构

1. 步骤定义

- 使用 **Step** 关键字定义一个步骤，步骤内的内容将按定义顺序执行。
- 每个步骤以 **Begin** 开头，以 **End** 结束。
- 默认第一个定义的步骤为主步骤，必须存在且至少包含一个 **Exit** 指令。

示例：

```

1 | Step Main
2 |   Begin
3 |     Speak "欢迎使用脚本解释器"
4 |   End

```

2. 输入与输出

- **Listen**：用于接受用户输入，将结果赋值给指定变量。如果变量未在变量表中，则自动插入变量表。
- **Speak**：用于输出，支持通过 + 对字符串和变量进行组合。

示例：

```
1 Speak "请输入您的名字: "  
2 Listen $name  
3 Speak "你好, " + $name
```

3. 分支结构

- **Switch**: 用于分支逻辑控制, 以 **Begin** 开始, 以 **End** 结束。
- 分支以 **Branch** 关键字定义, 每个 **Branch** 后接匹配条件 (字符串) 和 **Run** 指令, 用于跳转到指定步骤。
- **Default**: 当没有分支条件匹配时, 执行默认分支。

结构示例:

```
1 Switch $input  
2 Begin  
3     Branch "1" Run StepOne  
4     Branch "2" Run StepTwo  
5     Default Run Main  
6 End
```

4. 跳转与退出

- **Run**: 用于跳转到指定步骤, 只能在 **Branch** 中使用。
- **Exit**: 终止程序的指令, 必须在脚本中至少出现一次, 且只能在 **Run** 之后使用。

示例:

```
1 Switch $input  
2 Begin  
3     Branch "1" Run StepOne  
4     Branch "2" Run StepTwo  
5     Default Run Exit  
6 End
```

5. 注释

- 单行注释使用 **#**, 从 **#** 开始到行尾的内容为注释。
- 多行注释使用 **@** 包围。

示例:

```
1 # 这是一个单行注释  
2 @  
3 这是一个多行注释  
4 多行注释支持多行内容  
5 @
```

2.1.3 文法规定

1. 步骤要求

- 每个 **Step** 必须包含 **Begin** 和 **End**。
- 主步骤是脚本的入口, 必须存在, 且至少包含一个 **Exit** 指令。
- 步骤中的变量可在后续步骤中共享和引用。

2. Switch 限制

- Switch 只能在 Step 中使用。
- 每个 Switch 块以 Begin 开头，以 End 结束。
- Run 和 Exit 只能在 Switch 的分支中使用。

3. 变量规则

- 使用 \$ 标识变量。
- 变量可以通过 Listen、赋值语句声明。
- 在脚本文件的开头可以使用赋值语句直接对变量进行声明和初始赋值。

4. Exit 规则

- Exit 表示程序运行结束。
- 它只能作为 Run 的后续操作，不能单独使用，而 Run 只能在 Branch 或 Default 后使用。

5. Update 规则

- Update 用于更新符号表中的变量的值，支持直接赋值或计算操作，只能在 Step 中使用。

以下是一个脚本示例：

```

1  $billing = 0.0
2  $name = "aaa"
3  $trans = 0
4
5  Step welcome
6  Begin
7      Speak "你好, " + $name
8      Speak "请输入 余额 以查看余额, 输入 改名 以修改名字, 输入 投诉 来投诉, 输入 充值 来
进行充值, 输入 退出 以结束会话"
9      Listen $input
10     Switch $input
11     Begin
12         Branch "余额" Run Billing
13         Branch "改名" Run Rename
14         Branch "投诉" Run Complain
15         Branch "充值" Run Recharge
16         Branch "退出" Run Exit
17         Default Run defaultProc
18     End
19 End
20
21 Step Billing
22 Begin
23     Speak $name + ", 您的余额为" + $billing
24 End
25
26 Step Recharge
27 Begin
28     Speak "请输入您的充值金额, 金额必须为小数或者整数"
29     Listen $money
30     Speak "您的充值金额为:" + $money
31     Update $billing = $billing + $money
32     Update $trans = $trans + 1
33     Speak "您的余额为" + $billing
34 End
35
36 Step Complain
37 Begin

```

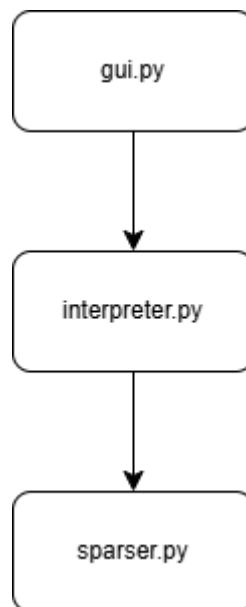
```

38     Speak "请输入您的建议"
39     Listen $suggestion
40     Speak "您的建议我们已经收到"
41 End
42
43 Step Rename
44 Begin
45     Speak "请输入您的新名字"
46     Listen $new_name
47     Update $name = $new_name
48 End
49
50 Step defaultProc
51 Begin
52     Speak "对不起，我刚刚没听清，请再说一遍"
53     Listen $input
54     Switch $input
55     Begin
56         Branch "余额" Run Billing
57         Branch "改名" Run Rename
58         Branch "投诉" Run Complain
59         Branch "退出" Run Exit
60     Default Run defaultProc
61     End
62 End

```

2.2 程序结构介绍

项目主要分为三个模块：GUI模块，解释器执行模块，词法语法分析模块，调用顺序如下：



接下来各模块进行说明：

2.2.1 词法语法分析模块

2.2.1.1 函数介绍

词法语法分析的函数如下：

1. `open_file(self, filename)`

功能：

负责打开指定文件并读取其中的DSL脚本内容。

参数：

- `filename`：字符串类型，表示需要进行语法分析的DSL脚本文件路径。

实现逻辑：

- 以只读模式打开文件，确保资源管理得当。
- 读取文件所有行并返回为列表格式，每一行为列表中的一个元素。
- 在文件未找到或读取失败时抛出异常并终止程序。

2. `is_linefeed(self, line)`

功能：

判断给定的行是否为空行。

参数：

- `line`：字符串类型，表示需要检测的当前行。

实现逻辑：

- 去除行首和行尾的空白字符。
- 判断行是否为空（长度为0或全是空白字符）。
- 返回布尔值：空行返回 `True`，非空行返回 `False`。

3. `is_comment(self, line)`

功能：

判断当前行是否为注释行。

参数：

- `line`：字符串类型，表示需要检测的当前行。

实现逻辑：

- 去除行首和行尾的空白字符。
- 判断行是否以 `#`，`@`

符号开头：

- `#` 表示单行注释。
- `@` 表示多行注释的开始或结束。
- 返回布尔值：注释行返回 `True`，否则返回 `False`。

4. `handle_comment(self)`

功能：

处理多行注释块。

实现逻辑：

- 遍历后续行直到找到以 @ 结尾的行，表示多行注释结束。
- 在找到结束标记前，跳过所有注释行，不执行任何逻辑。
- 如果未能找到结束标记，抛出异常提示多行注释块错误。

5. `handle_numeric(self, value_str, line)`

功能：

将字符串转换为数字类型。

参数：

- `value_str`: 字符串类型，表示需要转换的数值字符串。
- `line`: 字符串类型，表示出错时用于记录的当前行。

实现逻辑：

- 检查字符串是否包含小数点：
 - 包含小数点时尝试转换为浮点数。
 - 否则尝试转换为整数。
- 如果转换失败（值非法或格式错误），抛出异常并标记为语法错误。
- 返回成功转换的数值类型（整数或浮点数）。

6. `is_valid_identifier(self, identifier)`

功能：

验证给定的字符串是否为合法的标识符。

参数：

- `identifier`: 字符串类型，表示需要验证的标识符。

实现逻辑：

- 使用正则表达式检查标识符格式：
 - 以字母或下划线开头。
 - 后续字符可以是字母、数字或下划线。
- 返回布尔值：合法标识符返回 `True`，否则返回 `False`。

7. `is_number(self, s)`

功能：

判断给定的字符串是否为合法的数字（整数或浮点数）。

参数：

- `s`: 字符串类型，表示需要检测的字符串。

实现逻辑：

- 使用正则表达式匹配数字格式：
 - 整数：可选负号，后跟数字。
 - 浮点数：可选负号，数字后跟小数点和小数部分。
- 返回布尔值：合法数字返回 `True`，否则返回 `False`。

8. `parse_var(self, line)`

功能：

解析变量赋值语句，并将变量和其值存入变量表。

参数：

- `line`: 字符串类型，表示需要解析的当前行。

实现逻辑：

- 使用正则表达式匹配变量赋值语句：
 - 变量名以 `$` 开头，后跟赋值运算符 `=` 和初始值。
- 验证变量名是否合法。
- 检查初始值的格式：
 - 数字：解析为整数或浮点数。
 - 字符串：直接存储。
 - 含运算符的表达式则抛出语法错误。
- 将解析结果存入变量表，未赋值的变量默认为 `None`。

9. `parse_step(self, line)`

功能：

解析步骤定义，构建步骤树并存入语法树。

参数：

- `line`: 字符串类型，表示当前行的步骤定义。

实现逻辑：

- 使用正则表达式提取步骤名称。
- 验证步骤名称是否合法。
- 确保步骤块以 `Begin` 开头，以 `End` 结尾。
- 遍历步骤块内容：
 - 解析不同指令类型（如 `Speak`、`Listen`、`Switch` 等）。
 - 将解析后的指令存入步骤的过程列表。
- 确保第一个步骤包含至少一个 `Switch` 语句及 `Exit` 分支。

10. `parse_switch(self, token)`

功能：

解析 `Switch` 语句块及其分支。

参数：

- `token`: 列表类型，表示当前行的分词结果。

实现逻辑：

- 提取 `Switch` 变量并验证其合法性。
- 确保语句块以 `Begin` 开头，并解析每一行的分支语句。
- 对 `Branch` 和 `Default` 语句分别调用解析函数。
- 确保至少包含一个 `Default` 分支，否则抛出异常。

- 返回完整的 `switch` 语句结构，包括变量名和分支列表。

11. `parse_branch(self, token)`

功能：

解析 `Branch` 分支语句。

参数：

- `token`: 列表类型，表示当前行的分词结果。

实现逻辑：

- 使用正则表达式提取分支条件、执行动作及下一步骤。
- 验证动作合法性（支持 `Run` 或 `Exit`）。
- 返回解析后的 `Branch` 结构，包括条件、动作和下一步骤。

12. `parse_default(self, token)`

功能：

解析 `Default` 分支语句。

参数：

- `token`: 列表类型，表示当前行的分词结果。

实现逻辑：

- 使用正则表达式提取默认分支的动作及下一步骤。
- 返回解析后的 `Default` 结构，包括动作和下一步骤。

13. `exception(self, line=0, message="")`

功能：

处理语法分析中的异常情况。

参数：

- `line`: 字符串类型，表示出错时的当前行内容。
- `message`: 字符串类型，表示错误的详细信息。

实现逻辑：

- 打印包含行号和错误信息的详细提示。
- 终止程序运行。

14. `parse_file(self, filename)`

功能：

解析 `DSL` 脚本并构建完整的语法树。

参数：

- `filename`: 字符串类型，表示需要解析的文件路径。

实现逻辑：

- 调用 `open_file` 读取文件内容并存储每一行。
- 遍历文件的每一行，根据行内容调用不同的解析函数：

- 处理注释行。
- 解析变量赋值。
- 解析步骤定义。
- 逐行构建语法树，记录变量表和步骤内容。

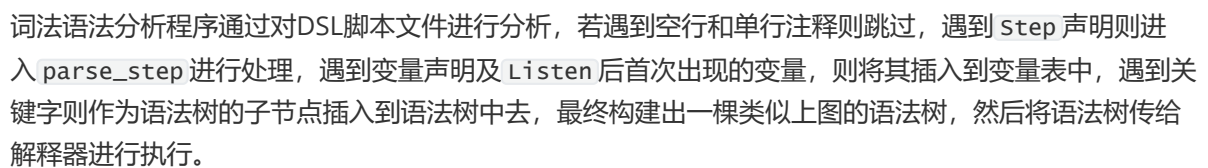
2.2.1.2 语法树结构

对于以下脚本：

```
1  @
2      你好机器人，专门用于进行初步测试
3  @
4  # 行注释测试
5  $name = 1
6  Step welcome
7  Begin
8  Speak $name + '你好，请问有什么可以帮您?'
9  Listen $input
10 Switch $input
11 Begin
12 Branch '你好' Run nihaoProc
13 Branch '你不好' Run nibuhaoProc
14 Branch '退出' Run Exit
15 Default Run defaultProc
16 End
17 End
18
19 Step nihaoProc
20 Begin
21 Speak '你好!!!'
22 End
23
24 Step nibuhaoProc
25 Begin
26 Speak '你不好!!!'
27 End
28
29 Step defaultProc
30 Begin
31 Speak '不好意思，请再说一遍'
32 Listen $input
33 Switch $input
34 Begin
35 Branch '你好' Run nihaoProc
36 Branch '你不好' Run nibuhaoProc
37 Branch '退出' Run Exit
38 Default Run defaultProc
39 End
40 End
```

解析得到的语法树如下：

具体结构如下图所示:



2.2.2 解释器运行模块

2.2.2.1 函数介绍

1. `input_with_timeout(x)`

功能:

实现带超时限制的用户输入功能。

参数:

- `x`: 整数类型, 表示用户输入的超时时间 (单位: 秒)。

逻辑实现:

- 创建一个独立线程, 通过 `InputThread` 类异步获取用户输入。
- 线程在超时时间内未完成输入, 将提示用户超时并退出程序。
- 返回用户输入的字符串; 如果超时, 则返回 `None`。

2. `get_script(self, file_list)`

功能:

解析 DSL 文件, 从词法语法分析程序获取构建的语法树并存储到 `tree` 中。

参数:

- `file_list`: 列表类型, 表示包含 DSL 文件路径的列表。

逻辑实现:

- 遍历文件列表, 调用 `parser.parse_file` 解析每个文件。
- 解析结果存储为语法树 `tree` 和变量表 `var`。

3. `initialize_step(self)`

功能:

初始化当前执行步骤。

参数:

无。

逻辑实现:

- 检查语法树是否为空, 若为空则打印错误信息并退出。
- 将当前步骤设置为语法树中的第一个步骤。

4. `fill_stepDic(self)`

功能:

将语法树中的步骤名称映射到步骤 ID, 并存储到 `stepDic` 中。

参数:

无。

逻辑实现:

- 遍历语法树的每个步骤。
- 为每个步骤生成唯一的步骤 ID, 并存入 `stepDic` 字典。

5. `handle_numeric(self, value_str, line)`

功能：

将字符串形式的数值转换为整数或浮点数。

参数：

- `value_str`: 字符串类型，表示数值字符串。
- `line`: 字符串类型，表示当前行内容（用于错误提示）。

逻辑实现：

- 判断字符串中是否包含小数点：
 - 含小数点则转换为浮点数。
 - 否则转换为整数。
- 如果转换失败，调用 `exception` 报告错误。

6. `is_number(self, s)`

功能：

判断给定字符串是否为合法的数字。

参数：

- `s`: 字符串类型，表示需要检测的字符串。

逻辑实现：

- 使用正则表达式匹配整数或浮点数格式：
 - 整数：可选负号开头，后接数字。
 - 浮点数：可选负号开头，数字中包含小数点。
- 返回布尔值：合法返回 `True`，否则返回 `False`。

7. `update_variable(self, line)`

功能：

解析变量赋值语句并将变量存入变量表。

参数：

- `line`: 字符串类型，表示当前的赋值语句。

逻辑实现：

- 使用正则表达式匹配赋值语句的变量名和表达式。
- 替换表达式中的变量名为其当前值。
- 判断表达式是否需要计算：
 - 数字或字符串直接赋值。
 - 复杂表达式调用 `eval` 计算。
- 将计算结果存入变量表。

8. `execute_script(self)`

功能：

逐步执行 DSL 脚本的逻辑。

参数：

无。

逻辑实现：

- 遍历当前步骤的指令列表，逐条执行：
 - **Speak 指令**：输出拼接后的字符串。
 - **Listen 指令**：获取用户输入并存储到变量表。
 - **Switch 指令**：根据变量值跳转到对应步骤。
 - **Run 指令**：跳转到指定步骤。
 - **Update 指令**：更新变量值。
- 如果当前步骤无明确下一步，自动回到第一个步骤。

9. `execute_run(self, args)`

功能：

执行 `Run` 指令，跳转到目标步骤。

参数：

- `args`: 列表类型，表示目标步骤的名称。

逻辑实现：

- 检查参数是否包含一个有效目标步骤。
- 若目标步骤为 `Exit`，标记脚本结束。
- 根据步骤名称从 `stepDic` 中获取步骤 ID，更新当前步骤。

10. `interpreter(action, file_list)`

功能：

解释 DSL 文件并执行脚本。

参数：

- `action`: `Action` 类实例，负责脚本解析与执行。
- `file_list`: 列表类型，表示包含 DSL 文件路径的列表。

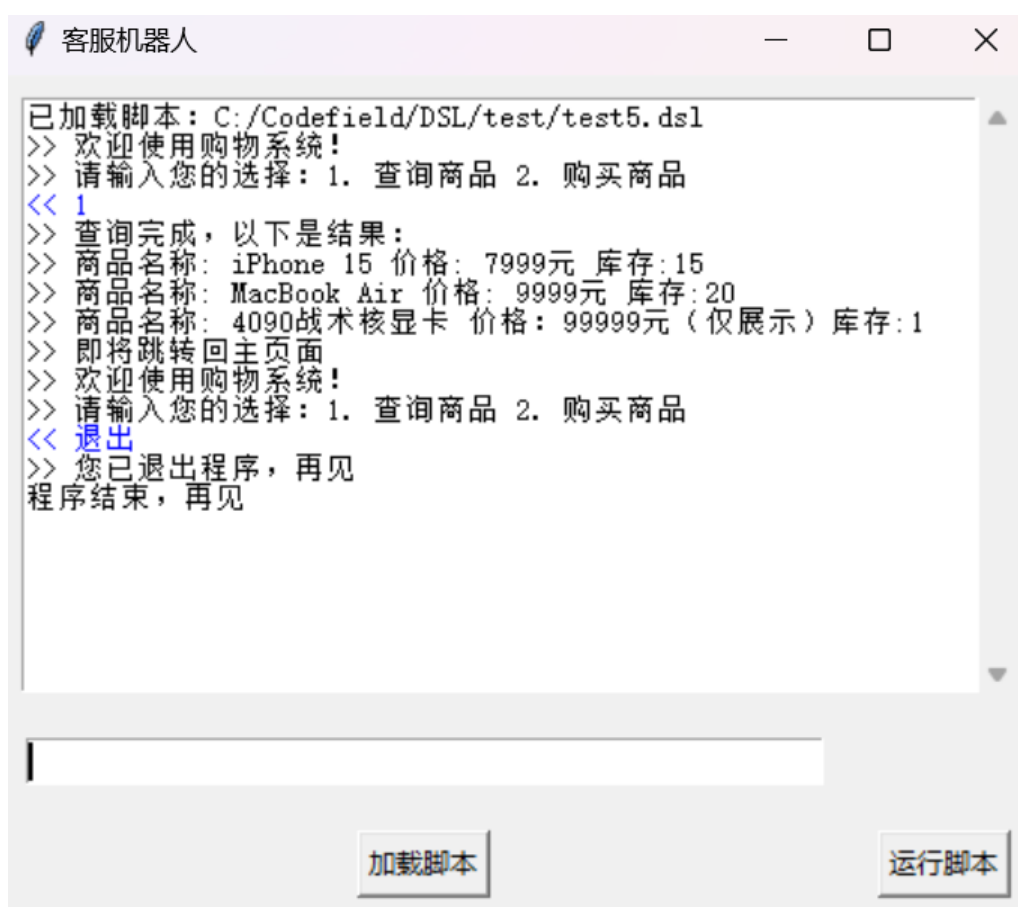
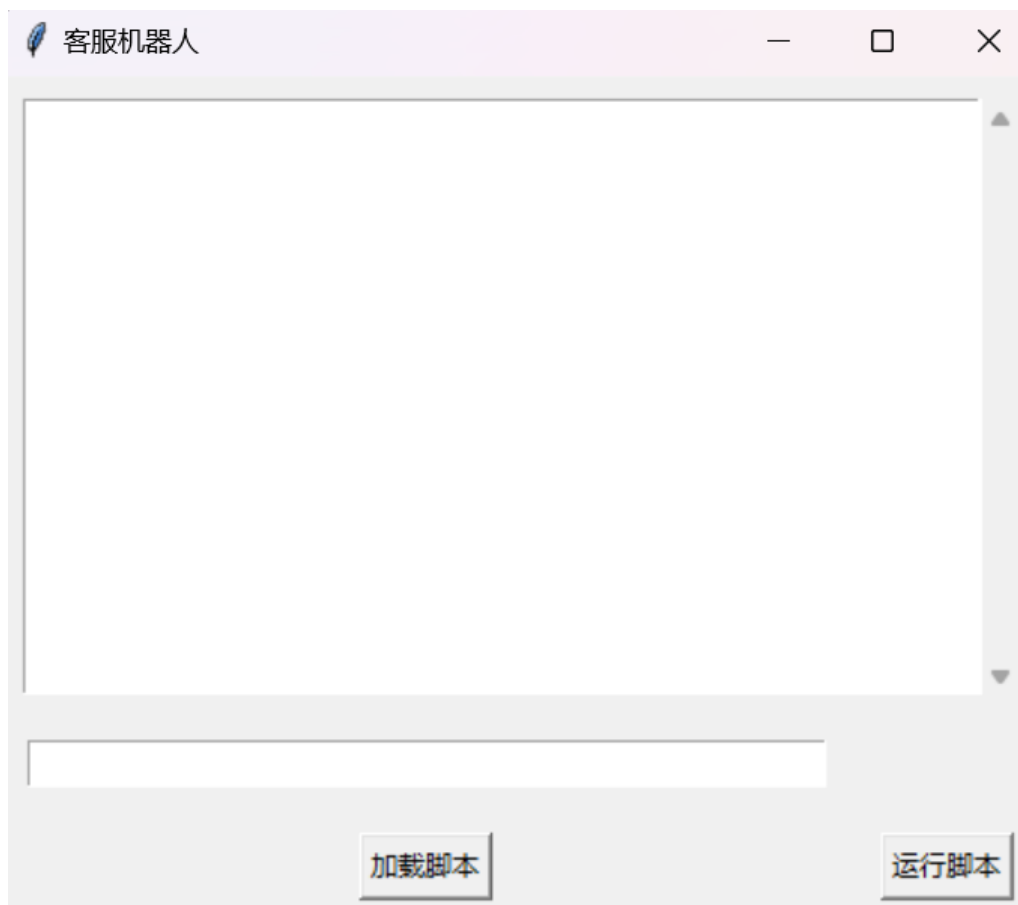
逻辑实现：

- 调用 `get_script` 生成语法树。
- 初始化步骤并填充步骤字典。
- 不断执行脚本步骤，直到脚本结束或用户退出。

解释器运行程序首先从词法语法分析程序处获得构建完成的语法树，然后根据生成的语法树来填充步骤字典，然后程序按顺序从步骤字典中取指令进行执行，执行时若遇到赋值语句，则调用 `update_variable` 函数对语句进行计算并更新变量表。若遇到过程调用语句则直接跳转到对应过程块，开始按序执行，执行结束后即可返回。

2.2.3 GUI介绍

本程序的GUI主要是调用python的 `tkinter` 库来绘制的，实现界面比较简单，通过将用户的输入储存到 `input_queue` 队列中，程序输出加入到 `output_queue` 中来实现消息的传递和输出，具体页面如下：



2.3 测试桩及自动测试程序介绍

2.3.1 test_sparsar

`test_sparsar` 是为尚未完全实现或测试的 `DSLSparsar` 模块设计的测试桩。该测试桩通过模拟 DSL 文件的输入内容，以及构造变量表和语法树，用于验证解析器模块的基本功能，辅助后续开发和功能测试。

测试桩结构与功能描述：

1. 模拟输入环境：

- 通过手动构造 DSL 文件的关键内容（包括 `Step`、`Speak`、`Listen`、`Switch` 等指令），模拟解析器可能面对的语法结构。
- 初始化变量表，并提供模拟的变量值，确保后续解析逻辑能正确解析变量。

2. 构造语法树：

- 使用 `DSLSparsar` 类的解析功能，将模拟的 DSL 文件内容逐步解析为语法树。
- 验证语法树中步骤、指令和分支逻辑是否符合预期。

3. 异常处理的初步验证：

- 模拟一些可能的语法错误（如非法的 `Step` 名称或不完整的 `Speak` 指令），确保解析器能正确捕获错误并终止解析。

4. 辅助未完成模块：

- 若解析器模块尚未完全实现，可以通过修改 `test_sparsar` 的测试逻辑，模拟解析结果。例如，直接返回预期的语法树或变量表，以支持其他模块的开发和测试。

测试桩功能细化：

- setUp 方法：** 初始化测试桩环境，包括创建解析器实例、模拟 DSL 文件内容和变量表，为解析器的功能测试提供上下文。
- 模拟完整解析逻辑：**
 - 构建的 DSL 文件包括多步骤、多分支场景，覆盖可能出现的关键指令组合。
 - 验证每一步骤中的指令是否能够正确解析，并按照逻辑关系构造语法树。
- 支持异常情况的验证：** 在特定测试中，可以通过构造非法输入，测试 `DSLSparsar` 模块的异常处理能力。
- 构造的语法树与期望输出：** 测试桩将期望的语法树结构直接写入代码中，用于对比解析器实际生成的结果。这样，在 `DSLSparsar` 模块未实现时，可以通过硬编码的方式模拟解析器的输出。

2.3.2 test_interpreter

`TestInterpreter` 是为尚未完全实现或测试的 `interpreter` 模块设计的测试桩，用于验证 DSL 脚本解析与执行过程中的核心功能模块（如语法树加载、变量更新、步骤执行等）的基本行为。通过模拟输入和检查输出，支持解析器与解释器功能的逐步完善。

测试桩结构与功能描述：

1. 支持脚本加载：

- 模拟加载 DSL 脚本文件，将语法树和变量表填充到解析器实例中。
- 验证解析器是否能正确构造脚本运行所需的上下文数据。

2. 验证基本功能模块：

- 检查是否能够正确执行核心功能，包括步骤初始化、变量更新、数字处理、步骤切换等。

3. 模拟脚本执行流程：

- 构建 DSL 脚本中的核心步骤和流程，验证解释器模块能否正确执行，并支持脚本执行过程的断点调试。

4. 辅助未完成模块：

- 若解释器的某些功能模块尚未实现，可通过调整测试桩逻辑直接模拟输出，为脚本执行的后续开发和测试提供支持。

测试桩功能细化：

- **setUp 方法：**
 - 初始化测试桩环境，包括创建解析器和解释器实例，确保测试桩中的功能可以在模拟的环境下独立运行。
 - 提供模拟 DSL 脚本文件路径，方便快速加载脚本内容。
- **输出验证：**
 - 测试桩直接打印步骤字典、变量表及语法树结构，便于调试和核对输出。
 - 验证每个测试点的运行结果是否与预期一致。

2.3.3 自动测试程序

本程序旨在自动化测试 DSL 脚本的解析与执行过程，通过批量处理脚本文件和输入数据，生成对应的输出文件，并对脚本的正确性进行验证。程序兼顾可扩展性与可维护性，适用于测试环境下的大规模 DSL 脚本测试任务。

程序结构与功能描述

1. 测试环境初始化：

- 通过 `setUp` 方法定义脚本测试目录 (`test`)、输入文件目录 (`input`) 和输出文件目录 (`output`)。
- 在首次运行时自动创建输出目录，确保程序能够顺利生成测试结果。

2. 脚本测试核心流程：

- 扫描测试目录中所有以 `.dsl` 结尾的 DSL 脚本文件。
- 对于每个脚本文件：
 - 根据脚本文件名匹配对应的输入文件和输出文件。
 - 若输入文件不存在，打印警告信息并跳过测试。
 - 打印测试进程的详细信息，包括当前处理的脚本、输入文件和生成的输出文件路径。
- 使用以下流程解析与执行 DSL 脚本：
 1. 解析与初始化：
 - 创建 `DSLParser` 实例进行脚本解析。
 - 使用 `Action` 类实例化脚本操作环境，初始化输入输出队列和其他上下文。
 - 加载脚本并初始化步骤信息。
 2. 加载输入文件：
 - 从对应的输入文件中逐行读取内容，并依次写入输入队列。
 3. 执行脚本：
 - 通过循环执行脚本指令，直至脚本执行完毕。
 - 在每次执行步骤后，捕获所有输出内容并存入输出队列。
 4. 生成输出文件：
 - 清空旧输出文件内容。
 - 将执行过程中的输出逐行写入对应的输出文件。

3. 输出验证与调试支持：

- 在控制台打印测试过程中的关键步骤和信息，包括当前测试文件、使用的输入文件及生成的输出文件路径。
- 输出文件内容保留脚本执行过程中的所有结果，便于对比分析与回归测试。

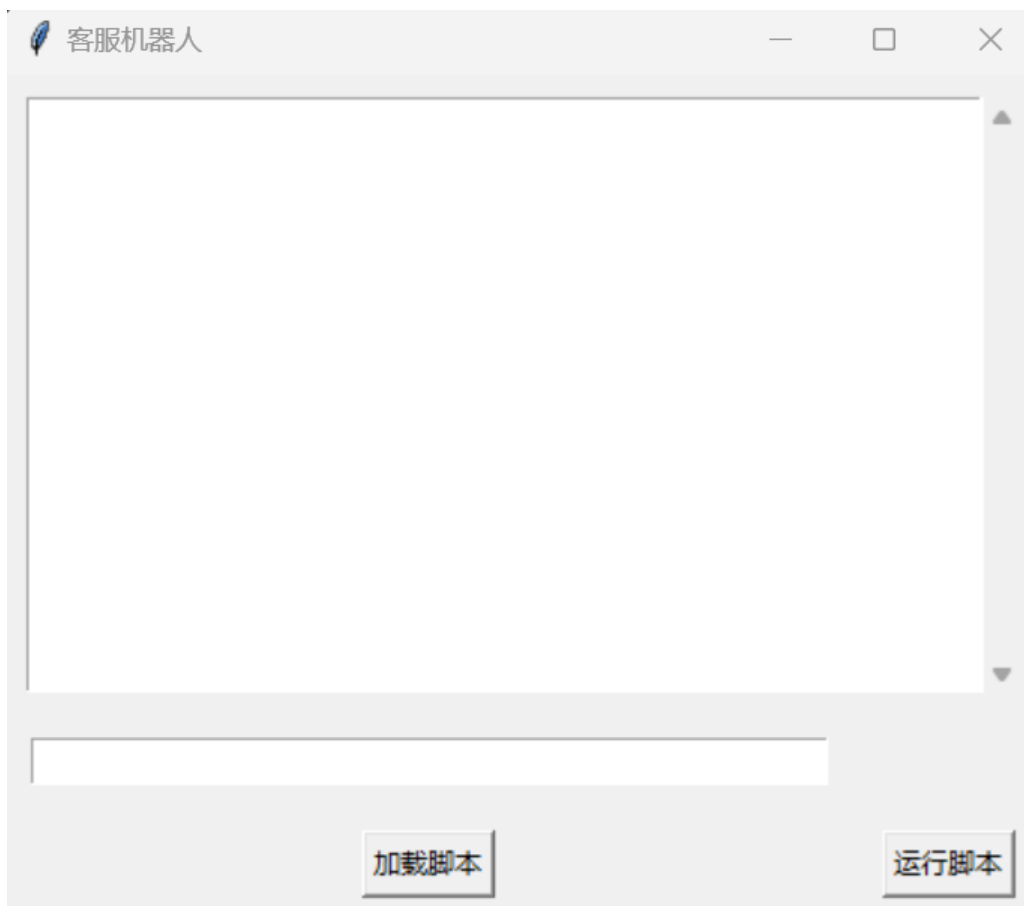
通过此自动化测试程序，用户可高效验证 DSL 脚本解析与执行的准确性，并快速定位问题，提升开发效率与测试质量。

3 用户使用说明







使用时文件结构需如下：

```
1 | project_root/  
2 |  
3 | └─ input/  存放自动测试程序需要读取的模拟用户输入，顺序需要与test中的对应脚本文件一致  
4 | └─ server/ 存放项目代码  
5 | └─ test/   存放相应的DSL脚本文件  
6 | └─ output/ 存放程序对应的输出
```

将脚本文件存放到test目录下后，打开GUI页面，点击加载脚本：



即可进入到脚本选取页面，选取需要执行的脚本打开：

 test_parser.dsl	2024/11/17 23:12	DSL 文件	1 KB
 test1.dsl	2024/11/19 13:54	DSL 文件	2 KB
 test2.dsl	2024/11/17 22:50	DSL 文件	1 KB
 test3.dsl	2024/11/17 23:08	DSL 文件	1 KB
 test4.dsl	2024/11/17 23:16	DSL 文件	2 KB
 test5.dsl	2024/11/19 9:27	DSL 文件	3 KB

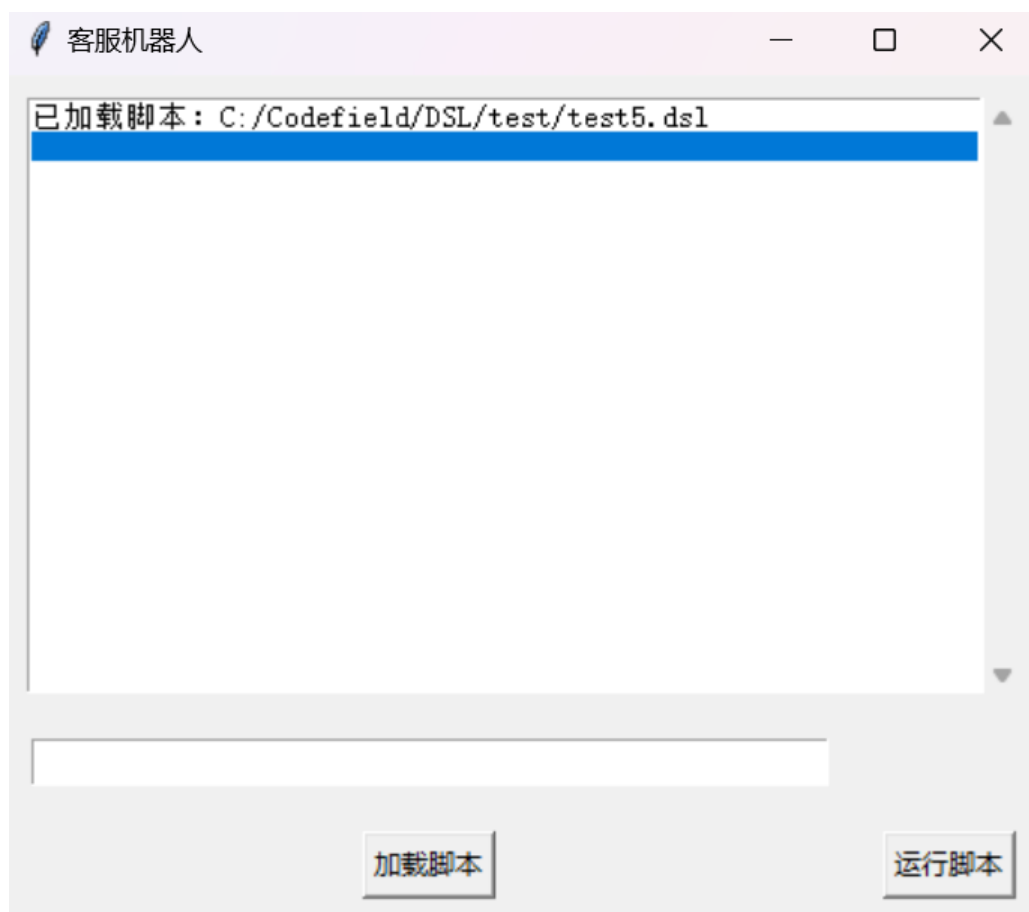
(N):

DSL Scripts (*.dsl)

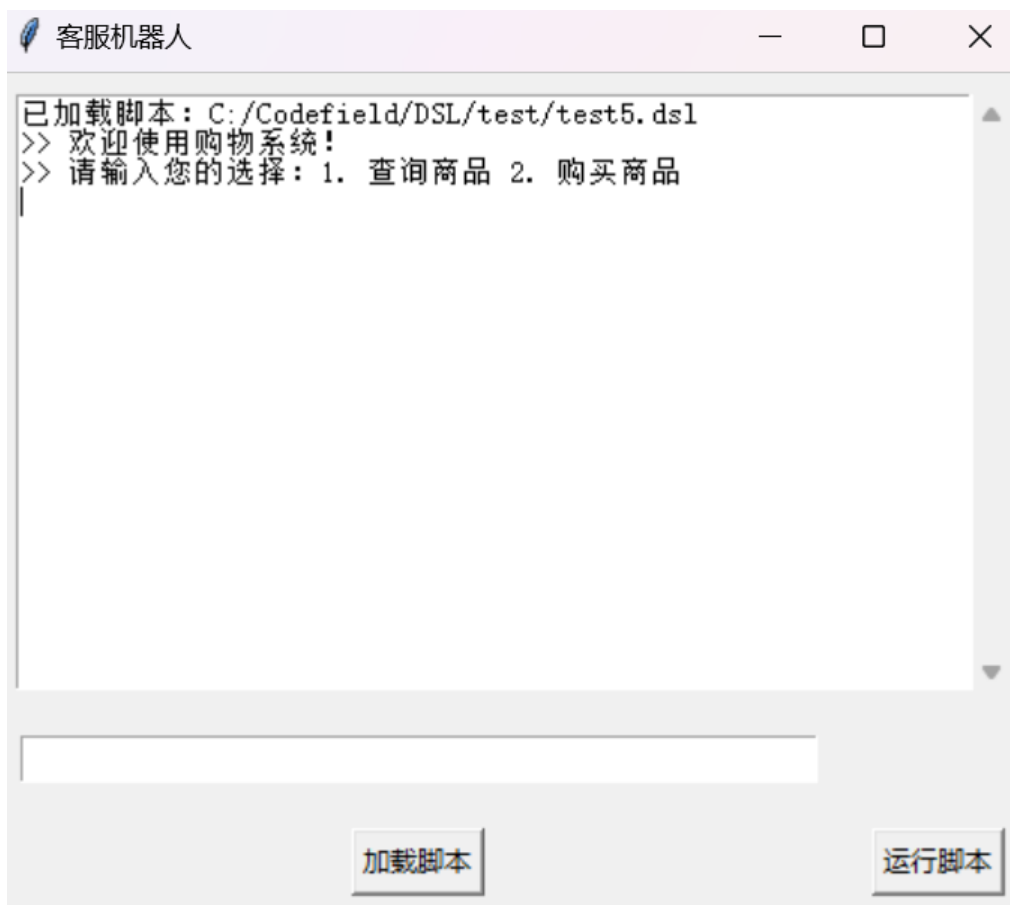
打开(O)

取消

脚本加载完成：



点击运行脚本：



相应的客服机器人即可启动并开始服务，用户可在下面的对话框输入对话进行操作，每次程序输出后再过5秒会自动跳转回开始页面，最后等待一段时间或直接输入退出即可退出客服机器人。

