《程序设计实践》课程实验报告

大作业：基于领域特定语言的客服机器人设计与实现

专业： 计算机科学与技术

班级： 2022211309

姓名： 马云霄

学号： 2022211320

北京邮电大学计算机学院

2024 年 11 月 30 日

目录

1 题目描述

1.1 题目背景

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 评分标准

本作业考察学生规范编写代码、合理设计程序、解决工程问题等方面的综合能力。满分 100 分，具体评分标准如下：

- 风格：满分 15 分，其中代码注释 6 分，命名 6 分，其它 3 分。
- 设计和实现：满分 30 分，其中数据结构 7 分，模块划分 7 分，功能 8 分，文档 8 分。
- 接口：满分 15 分，其中程序间接口 8 分，人机接口 7 分。
- 测试：满分 30 分，测试桩 15 分，自动测试脚本 15 分。
- 记法：满分 10 分，文档中对此脚本语言的语法的准确描述。

注意：抄袭或有意被抄袭均为 0 分

2 项目结构

```
dsl/
    lexer.py       # 词法分析器，负责将DSL脚本转化为tokens
    parser.py      # 语法解析器，负责将tokens解析成AST
    interpreter.py # 解释器，负责执行AST并进行交互
scripts/
    example1.dsl  # 示例DSL脚本
    example2.dsl  # 示例DSL脚本
    ...
tests/
    test_lexer.py  # 词法分析器测试
    test_parser.py  # 解析器测试
    test_interpreter.py # 解释器测试
    performance_test.py # 性能测试自动测试脚本
    stress_test.py # 压力测试自动测试脚本
```

```
docs/
    developer_guide.pdf # 开发者指南
    user_manual.pdf # 用户手册
    dsl_specification.pdf # DSL语言规范文档
photos/
    bot.png  # 机器人头像
    user.png  # 用户头像
ui/
    gui.py # ui界面设计
main.py # 主程序
requirements.txt # 实验相关依赖
实验报告.pdf # 本次实验总的实验报告
```

3 记法

本 DSL（领域特定语言）设计用于描述交互式客服机器人的对话流程。通过使用简单易懂的语法，用户可以编写基于状态机的自动应答脚本，支持根据用户输入在不同模式下进行条件判断、响应输出、模式切换及变量赋值等操作。

3.1 语言概述

DSL 脚本通过状态模式（MODE）来管理对话流程。每个模式下可以定义多条 if、elif 和 else 语句，用于处理用户输入，并根据条件选择不同的响应或行为。

语言特点：

- 基于状态的模式切换。
- 简单的条件判断（if/elif/else）。
- 支持输出响应、变量赋值和模式跳转。

3.2 关键字与运算符

DSL 语言的基本元素包括关键字、运算符、标识符和常量。以下是语言中常用的元素：

3.2.1 关键字

- start：定义对话开始的初始状态。
- end：定义对话结束。
- if：用于条件判断，后接条件和操作。
- elif：在 if 之后用于进一步的条件判断。
- else：在没有其他条件匹配时执行的操作。
- response：用于输出响应。
- go：用于模式跳转，指定下一个模式。
- set：用于变量赋值操作。

- in：检查某个值是否包含在列表或字符串中。
- user_input：引用用户输入的内容。

3.2.2 运算符

- =：赋值运算符，用于变量赋值。
- +：加法运算符，用于数字的加法操作。

3.2.3 标识符

- MODE：以大写字母组成的模式名称（如：INIT、ACCOUNT 等）。
- ID：变量名或其他标识符，通常由字母和下划线组成，支持字母数字组合。

3.3 语法规则（BNF）

以下是 DSL 语言的文法定义，采用 BNF 范式描述：

```
<program> ::= <start_mode> <mode_section>* <end>

<start_mode> ::= "start" <mode_name> <statement>*

<end> ::= "end"

<mode_section> ::= <mode_name> <statement>*

<mode_name> ::= [A-Z][A-Z]*  // 模式名称是大写字母

<statement> ::= <if_statement>
              | <elif_statement>
              | <else_statement>
              | <response_statement>
              | <go_statement>
              | <set_statement>

<if_statement> ::= "if" <condition> "then" <response>

<elif_statement> ::= "elif" <condition> "then" <response>

<else_statement> ::= "else" <response>

```

```
<response_statement> ::= "response" <string>

<go_statement> ::= "go" <mode_name>

<set_statement> ::= "set" <variable> "=" <expression>

<condition> ::= <expression> "in" <user_input> | <expression>
<expression> ::= <ID> | <number> | <user_input> | <string>
<variable> ::= <ID>

<user_input> ::= "user_input"
<number> ::= [0-9]+

<string> ::= '"' [^"]* '"'
```

3.4 AST 范式定义

3.4.1 根节点

根节点为 program 类型，它包含一个 statements 数组，表示整个对话脚本的语句。

```
{
  "type": "program",
  "statements": [
    // 具体的语句列表，包含多种类型的节点
  ]
}
```

3.4.2 模式节点

每个 mode 节点表示一个对话模式。模式名称是一个字符串（例如 INIT、ACCOUNT 等），并且每个模式下可以包含多个语句。每个 mode 节点都包含一个 statements 字段，它是该模式下所有语句的集合。

```
{
  "type": "mode",
  "mode": "<mode_name>", // 模式名称，如 "INIT"、"ACCOUNT" 等
  "statements": [       // 当前模式下的语句列表
    // 语句内容
```

```
  ]
}
```

3.4.3 条件判断节点

这些节点用于条件判断，根据用户输入执行不同的逻辑。每个 if、elif 或 else 节点包含条件和响应，并可能包括后续语句。节点的主要字段包括：

```
{
  "type": "<statement_type>",  // 'if' / 'elif' / 'else'
  "condition": ["<condition>"], // 条件，可能是一个字符串或多个字符串
  "response": "<response_text>", // 响应内容
  "next_statements": [         // 后续语句
    // 后续的语句内容，如 'go', 'set' 等
  ]
}
```

3.5 示例脚本

以下是一个示例脚本，展示了如何使用 DSL 语言定义客服机器人的对话逻辑。

```
start INIT
    if "你好" in user_input then
        response "您好，很高兴为您服务，请问您的需要是"
    elif "账户" in user_input then
        response "已转移至账户模式"
        go ACCOUNT
    elif "商品" in user_input then
        response "已转移至商品模式"
        go GOODS
    else
        response "抱歉，我没有理解您的问题"

ACCOUNT
    if "余额" in user_input then
        response "您的余额为 "
    elif "充值" in user_input then
        response "请输入您所充值的金额"
        set balance = balance + user_input
```

```
19      elif ”退出” in user_input then
20          response ”您已退出账户模式”
21          go INIT
22      else
23          response ”抱歉，我没有理解您的问题”
24
25  GOODS
26      if ”名称” in user_input then
27          response ”在售商品的名称为：商品 A, 商品 B, 商品 C”
28      elif ”查询” in user_input then
29          response ”已转移至查询模式，输入对应商品名称查询信息”
30          go QUERY
31      elif ”退出” in user_input then
32          response ”您已退出商品模式”
33          go INIT
34      else
35          response ”抱歉，我没有理解您的问题”
36
37  QUERY
38      if ”商品 A” in user_input then
39          response ”商品 A：价格：100 元，库存：50 件”
40      elif ”商品 B” in user_input then
41          response ”商品 B：价格：200 元，库存：30 件”
42      elif ”商品 C” in user_input then
43          response ”商品 C：价格：300 元，库存：20 件”
44      elif ”退出” in user_input then
45          response ”您已退出商品查询模式”
46      else
47          response ”抱歉，我没有理解您的问题”
48  end
```

该 dsl 代码描述了一个基本的多模式聊天机器人，能够根据用户输入的不同关键字进行响应并转移到相应的功能模式：

- **INIT（初始化模式）**：当用户输入” 你好” 时，机器人会迎接用户并询问需求。如果用户输入” 账户” 或” 商品”，则分别进入账户管理模式或商品模式。若用户输入其他内容，则提示“抱歉，我没有理解您的问题”。
- **ACCOUNT（账户模式）**：用户可以查询余额、充值或者退出。如果用户输入” 余额”，系统会返回当前余额。如果输入” 充值”，系统会提示输入充值金额并更新余额。如果用户选择” 退出”，则退出账户模式，返回初始化模式。

- **GOODS（商品模式）**：用户可以查询商品名称或进入商品查询模式。如果用户输入” 名称”，系统会返回所有商品名称。如果输入” 查询”，则转入查询模式，允许用户进一步查询具体商品的价格和库存信息。
- **QUERY（查询模式）**：用户可以根据商品名称查询详细信息，如商品 A、商品 B 和商品 C 的价格与库存。用户也可以选择退出查询模式，返回商品模式。

3.6 可扩展性

该 DSL 语言可以通过以下方式进行扩展：
(1) 添加更多的内建函数，如 cin()、max()，等用于处理更多不同领域的问题。
(2) 支持更多条件运算符，如 <、>、==，增强条件表达能力。
(3) 扩展响应类型，支持动态生成文本、与外部 API 交互等。

4 代码设计和实现

4.1 lexer.py

本节将介绍 ‘lexer.py‘ 文件的设计与实现。该文件实现了一个简单的词法分析器（Lexer），其主要任务是将输入源代码分解成可以供语法分析器使用的词法单元（tokens）。

4.1.1 设计思路

词法分析器主要依赖正则表达式来匹配源代码中的不同元素，包括关键字、标识符、数字、运算符等。通过定义一个 ‘TOKEN_SPECIFICATIONS‘ 列表来描述所有可能的词法单元，词法分析器将根据这些规范匹配输入文本，并生成一系列的 tokens。每个 token 都包含类型和对应的值，供后续语法分析器使用。

4.1.2 代码实现

```
import re

# 定义不同的词法记号（tokens）
TOKEN_SPECIFICATIONS = [
    ('NUMBER', r'\d+'), # 整数
    ('STRING', r'"([^"]*)"'), # 双引号中的字符串
    ('IF', r'if'), # 'if' 关键字
    ('ELIF', r'elif'), # 'elif' 关键字
    ('THEN', r'then'), # 'then' 关键字
    ('ELSE', r'else'), # 'else' 关键字
    ('RESPONSE', r'response'), # 'response' 关键字
```

```
    ('START', r'start'), # 'start' 关键字
    ('END', r'end'), # 'end' 关键字
    ('USER_INPUT', r'user_input'), # 'user_input' 变量
    ('GO', r'go'), # 'go' 关键字
    ('QUOTE', r'"'), # 双引号字符
    ('IN', r'in'), # 'in' 关键字，用于检查是否包含在列表或字符串中
    ('SET', r'set'), # 'set' 关键字，用于赋值
    ('MODE', r'[A-Z][A-Z]*'), # 大写字母的单词，作为模式（MODE）
    ('ID', r'[a-zA-Z_][a-zA-Z0-9_]*'), # 标识符（例如：'account'、'payment'）
    ('NEWLINE', r'\n'), # 换行符
    ('SKIP', r'[ \t]+'), # 跳过空格和制表符
    ('COMMENT', r'#.*'), # 注释，单行注释以'#'开头
    ('ASSIGN', r'='), # 赋值运算符
    ('PLUS', r'\+'), # 加法运算符
    ('MISMATCH', r'.'), # 任何其他字符（错误）
]

# 合并所有正则表达式
master_regex = '|'.join(f'(?P<{pair[0]}>{pair[1]})' for pair in TOKEN_SPECIFICATIONS)
```

首先，代码定义了一个 ‘TOKEN_SPECIFICATIONS‘ 列表，包含了所有词法单元的类型与其对应的正则表达式。每个正则表达式用于匹配特定类型的输入，例如整数、字符串、关键字等。‘master_regex‘ 将这些正则表达式合并成一个大正则表达式，用于在整个源代码文本中寻找匹配的词法单元。

```
class Lexer:
    def __init__(self, code):
        self.code = code
        self.line_number = 1
        self.position = 0
        self.tokens = []

    def tokenize(self):
        line = self.code
        while line:
            match = re.match(master_regex, line)
            if match:
                type_ = match.lastgroup
```

```
14              value = match.group(type_)
15
16              if type_ == 'NEWLINE':
17                  self.line_number += 1
18                  self.position = 0
19              elif type_ == 'COMMENT':
20                  # 忽略注释，不做任何操作，直接跳到下一个
21                  line = line[match.end():]
22                  continue
23              elif type_ == 'STRING':
24                  # 如果是字符串，去掉双引号
25                  value = value[1:-1]  # 去除双引号
26                  self.tokens.append((type_, value))
27              elif type_ != 'SKIP':
28                  self.tokens.append((type_, value))
29
30              line = line[match.end():]
31          else:
32              raise ValueError(f'Illegal character at line {self.line_number}, position {self.position}')
33      return self.tokens
```

接下来是 ‘Lexer‘ 类的实现。该类用于将源代码解析为词法单元。在 “ 方法中，初始化了源码文本、行号、当前位置和一个空的 ‘tokens‘ 列表。‘tokenize‘ 方法负责遍历源码并根据正则表达式匹配不同的词法单元。匹配到的结果会根据类型进行处理：例如，跳过空格和注释，处理字符串时去除双引号，其他匹配到的词法单元会被添加到 ‘tokens‘ 列表中。

```
1  # 接口方法
2  def lex_script(code):
3      lexer = Lexer(code)
4      return lexer.tokenize()
```

最后，‘lex_script‘ 函数提供了一个简单的接口，用于外部调用词法分析功能。通过该接口，外部代码可以将源代码字符串传递给 ‘Lexer‘ 类实例，然后调用 ‘tokenize‘ 方法生成所有的词法单元。

4.1.3 总结

通过定义一组正则表达式和使用 ‘Lexer‘ 类，‘lexer.py‘ 文件成功地将输入的源代码转化为一系列的词法单元。这些词法单元将作为输入传递给后续的语法分析器，帮助程序进一步理解代码结构。此实现简单易懂，适用于词法分析的基本需求，且具有扩展性，可以支持更多的词法单元类型。

4.2 parser.py

4.2.1 设计思路

在解析阶段，我们的目标是从词法分析器输出的 tokens 中提取结构化的程序语法。这部分代码主要是实现了一个简单的解析器，该解析器能够解析类似条件控制语句（如 if/elif/else）、赋值语句、模式定义以及其他控制命令（如 response 和 go）的逻辑。
该解析器使用递归下降解析（recursive descent parsing）方法，通过从 tokens 流中依次获取和处理符号，构建语法树。

4.2.2 代码实现

定义 Parser 类

```
import re
from dsl.lexer import Lexer

class Parser:
    def __init__(self, tokens):
        self.tokens = tokens  # 存储所有的 tokens（词法单元）
        self.position = 0  # 当前 token 的位置
        self.token = None  # 当前 token
        self.advance()  # 读取下一个 token
        self.modes = set()  # 用于跟踪已定义的模式
        self.found_init = False  # 用于标记是否已经找到 INIT 模式
```

功能说明：上述代码片段展示了 'Parser' 类的初始化方法。它接收词法分析器生成的 'tokens' 作为输入，初始化了多个变量用于解析状态的跟踪。'advance' 方法用于推进到下一个 token，'modes' 用于记录已经定义过的模式，'found_init' 用于标记是否已经找到 'INIT' 模式。

推进 token 指针

```
    def advance(self):
        """ 移动到下一个 token """
        if self.position < len(self.tokens):
            self.token = self.tokens[self.position]  # 获取当前 token
            self.position += 1  # 移动到下一个 token
        else:
            self.token = None  # 如果已经到达 tokens 末尾，设置 token 为 None
```

功能说明：‘advance‘ 方法用于推进 token 指针，如果未到达 tokens 的末尾，则更新当前 token，若已到达末尾，则将当前 token 设置为 ‘None‘。

parse 方法

```
1   def parse(self):
2       """ 解析整个脚本，从顶层开始 """
3       return self.program()
```

功能说明：‘parse‘ 方法是解析器的入口方法，它会调用 ‘program()‘ 方法，开始整个程序的解析过程。

解析整个程序

```
1   def program(self):
2       """ 解析整个程序，程序以 'start' 开头并以 'end' 结束 """
3       statements = []  # 用于存储解析出的语句
4
5       # 期望在开始处出现 'start' token
6       if self.token[0] == 'START':
7           self.advance()  # 跳过 'start' token
8
9           while self.token and self.token[0] != 'END':  # 直到遇到 'end' token
10              statement = self.statement()  # 解析单个语句
11              statements.append(statement)  # 将解析出来的语句添加到列表
12
13              # 如果当前语句是条件语句（if/elif/else），将其后续语句（go/set）也作为一部分加入
14              if statement['type'] in ['if', 'elif', 'else']:
15                  statements[-1]['next_statements'] = statement.pop('next_statements')
16
17          # 期望最后是 'end' token
18          if self.token[0] == 'END':
19              self.advance()
20          else:
21              raise SyntaxError("Expected 'end' but got {}".format(self.token))
22
23          # 确保脚本中包含 INIT 模式
24          self.check_init_mode()
```

```
25
26        return {'type': 'program', 'statements': statements}
```

功能说明： 在 ‘program‘ 方法中，程序从 ‘start‘ 语句开始，依次解析每个语句直到遇到 ‘end‘。如果程序中没有 ‘end‘ 或 ‘INIT‘ 模式定义，将会抛出 ‘SyntaxError‘ 异常。

解析每个语句调度方法

```
1     def statement(self):
2         """ 解析单个语句 """
3         if self.token[0] == 'IF':
4             return self.if_statement()
5         elif self.token[0] == 'ELIF':
6             return self.elif_statement()
7         elif self.token[0] == 'ELSE':
8             return self.else_statement()
9         elif self.token[0] == 'RESPONSE':
10            return self.response_statement()
11        elif self.token[0] == 'GO':
12            return self.go_statement()
13        elif self.token[0] == 'SET':
14            return self.set_statement()
15        elif self.token[0] == 'MODE':
16            return self.mode_statement()
17        else:
18            raise SyntaxError("Unexpected token: {}".format(self.token))
```

功能说明： ‘statement‘ 方法是用于解析每一个语句的调度方法。它根据当前 token 的类型（如 ‘IF‘、‘ELSE‘、‘GO‘ 等）选择相应的解析函数。如果遇到未预料到的 token 类型，则抛出 ‘SyntaxError‘。

解析各种语句、关键字情况

```
1     def if_statement(self):
2         """ 解析 'if' 语句：if <condition> then response <message> """
3         self.advance()  # 跳过 'if'
4         condition = self.condition()  # 解析条件
5
6         if self.token[0] == 'THEN':
7             self.advance()  # 跳过 'then'
```

```
        else:
            raise SyntaxError("Expected 'then' but got {}".format(self.token))

        if self.token[0] == 'RESPONSE':
            self.advance()  # 跳过 'response'
            response_message = self.response_message()  # 解析响应消息
        else:
            raise SyntaxError("Expected 'response' but got {}".format(self.token))

        # 解析接下来的 go 和 set 语句
        next_statements = []
        while self.token and self.token[0] in ['GO', 'SET']:
            if self.token[0] == 'GO':
                next_statements.append(self.go_statement())  # 解析 go 语句
            elif self.token[0] == 'SET':
                next_statements.append(self.set_statement())  # 解析 set 语句

        return {
            'type': 'if',
            'condition': condition,
            'response': response_message,
            'next_statements': next_statements  # 包含接下来的 go 和 set 语句
        }
```

功能说明：在 ‘if_statement‘ 方法中，首先解析 ‘if‘ 条件部分，然后解析响应消息（response），接着解析可能出现的 ‘go‘ 和 ‘set‘ 语句，最后将其封装为一个字典返回。

解析模块定义

```
    def mode_statement(self):
        """ 解析模式定义（如 'INIT', 'ACCOUNT' 等） """
        self.advance()  # 跳过 'mode'
        mode_name = self.token[1]  # 获取模式名称
        self.advance()

        if mode_name in self.modes:
            raise SyntaxError("Mode {} already defined".format(mode_name))

        self.modes.add(mode_name)  # 添加模式到已定义模式集
```

```
11        return {'type': 'mode', 'mode_name': mode_name}
```

功能说明：‘mode_statement‘ 用于解析模式定义语句，如 ‘INIT‘、‘ACCOUNT‘ 等。若模式已被定义，则抛出错误。

初始模块存在性检测

```
1     def check_init_mode(self):
2         """ 确保脚本中包含 INIT 模式 """
3         if not self.found_init:
4             raise SyntaxError("Expected 'INIT' mode to be defined in the script")
```

功能说明：‘check_init_mode‘ 方法用于检查脚本中是否定义了 ‘INIT‘ 模式。如果没有，则抛出错误。

4.2.3 总结

上述代码展示了一个简单的递归下降解析器，它能够从词法单元（tokens）中提取出程序的语法结构，并确保脚本符合预期的语法规则。解析器根据不同的语句类型（如 ‘if‘、‘else‘、‘go‘、‘set‘ 等）进行相应的处理，并最终生成一个包含语法树的数据结构，供后续的执行引擎使用。

4.3 interpreter.py

4.3.1 设计思路

解释器的设计目标是处理解析后的抽象语法树（AST），并根据用户输入和当前模式执行相关的操作。解释器维护了一个包含当前用户状态的上下文，能够根据模式切换、条件判断和变量操作来动态响应用户的请求。

解释器的主要功能包括：1. 状态管理：维护用户余额、当前模式和用户输入。2. 模式处理：根据解析后的 AST 中定义的模式（如 ‘INIT‘, ‘ACCOUNT‘）来处理不同的操作。3. 条件判断：根据用户输入，判断并执行与模式相关的 ‘if‘, ‘elif‘, ‘else‘语句。4. 用户交互：通过输入输出交互实现动态响应，处理用户充值等特定操作。

在设计上，解释器会根据当前模式的操作顺序执行语句，并根据用户的输入决定下一步的操作。每个模式下都有若干条件语句（‘if‘, ‘elif‘, ‘else‘），用于根据不同输入给出相应的反馈。

4.3.2 代码实现

初始化与状态设置

```
1 from dsl.lexer import Lexer
2 from dsl.parser import Parser
```

```
3
4  # 解释器类
5  class Interpreter:
6      def __init__(self, ast, balance=0.0):
7          # 初始化状态字典，记录每个模式下的操作
8          self.context = {
9              'balance': balance,
10             'user_input': '',
11             'current_mode': 'INIT',
12         }
13         self.program = self.parse_ast(ast)
```

功能说明：初始化‘Interpreter‘类的实例，设定用户的初始状态。该部分还调用了‘parse_ast‘方法，将 AST 解析为程序的内部数据结构，便于后续的模式处理。

解析抽象语法树（AST）

```
1      def parse_ast(self, ast):
2          # 解析 AST，将每个 mode 的操作存入字典
3          modes = {}
4          for statement in ast['statements']:
5              if statement['type'] == 'mode':
6                  current_mode = statement['mode']
7                  modes[current_mode] = {
8                      'if_conditions': [],
9                      'elif_conditions': [],
10                     'else_condition': None
11                 }
12             elif statement['type'] == 'if':
13                 modes[current_mode]['if_conditions'].append(statement)
14             elif statement['type'] == 'elif':
15                 modes[current_mode]['elif_conditions'].append(statement)
16             elif statement['type'] == 'else':
17                 modes[current_mode]['else_condition'] = statement
18         return modes
```

功能说明：该方法将抽象语法树（AST）中的每个语句类型（‘mode‘, ‘if‘, ‘elif‘, ‘else‘）解析并存储到一个字典中。每个模式（‘mode‘）会包含条件语句（‘if‘, ‘elif‘, ‘else‘）及其相应的操作。

充值功能实现

```
def prompt_for_recharge(self):
    """ 提示用户输入充值金额，并检查其合法性 """
    if self.context['current_mode'] != 'ACCOUNT':
        return " 无法进行充值操作。请先进入账户模式。"

    while True:
        try:
            recharge_amount = float(input(" 请输入您所充值的金额（浮动数）: "))
            if recharge_amount < 0:
                print(" 金额不能为负，请重新输入。")
                continue
            self.context['balance'] += recharge_amount
            return f" 充值成功！您的新余额为 {self.context['balance']:.2f} 元"
        except ValueError:
            print(" 输入无效，请确保您输入的是一个有效的数字。")
```

功能说明：该方法处理用户充值操作。首先检查是否在 'ACCOUNT'模式下，然后提示用户输入充值金额，并进行合法性验证。如果输入有效，余额将更新并反馈给用户。

输入处理与条件判断

```
def process_input(self, user_input):
    # 更新用户输入
    self.context['user_input'] = user_input

    # 获取当前模式下的操作
    current_mode = self.context['current_mode']
    mode_operations = self.program.get(current_mode, {})

    # 处理 if/elif/else 逻辑
    for condition in mode_operations['if_conditions']:
        if any(cond in user_input for cond in condition['condition']):
            response = condition['response']
            next_statements = condition['next_statements']
            self.handle_next_statements(next_statements)

```

```
            # 如果用户输入是 " 充值", 跳转到充值处理流程
            if '充值' in user_input:
                response = self.prompt_for_recharge()  # 处理充值流程
                return response

            # 处理余额输出时保留 2 位小数
            if '余额' in response:
                response = f"{response} {self.context['balance']:.2f}"
            return response
```

功能说明：根据当前模式和用户输入，逐一判断并执行相关的 'if'条件语句。如果满足条件，执行相应的操作。如果用户输入涉及充值操作，跳转到充值处理流程，并返回更新后的余额。

处理 'elif'和 'else'条件

```
    for condition in mode_operations['elif_conditions']:
        if any(cond in user_input for cond in condition['condition']):
            response = condition['response']
            next_statements = condition['next_statements']
            self.handle_next_statements(next_statements)

            # 如果用户输入是 " 充值", 跳转到充值处理流程
            if '充值' in user_input:
                response = self.prompt_for_recharge()  # 处理充值流程
                return response

            # 处理余额输出时保留 2 位小数
            if '余额' in response:
                response = f"{response} {self.context['balance']:.2f}"
            return response

    if mode_operations['else_condition']:
        response = mode_operations['else_condition']['response']
        next_statements = mode_operations['else_condition']['next_statements']
        self.handle_next_statements(next_statements)

        # 如果用户输入是 " 充值", 跳转到充值处理流程
        if '充值' in user_input:
            response = self.prompt_for_recharge()  # 处理充值流程
```

```
            return response

        # 处理余额输出时保留 2 位小数
        if '余额' in response:
            response = f"{response} {self.context['balance']:.2f}"
        return response
```

功能说明：在 ‘if‘条件不匹配时，继续判断 ‘elif‘和 ‘else‘条件。每个条件符合时，执行相应的操作。如果涉及余额查询或者充值操作，进行相应处理并更新输出。

处理后续操作（如模式切换和变量设置）

```
def handle_next_statements(self, next_statements):
    for statement in next_statements:
        if statement['type'] == 'go':
            self.context['current_mode'] = statement['mode']
        elif statement['type'] == 'set':
            var_name = statement['variable']
            expression = statement['expression']

            # 仅当表达式类型为加法运算时才进行处理
            if expression['type'] == 'addition':
                left = self.context.get(expression['left'], 0)

                # 确保只有数字才能参与加法运算
                try:
                    right = float(self.context['user_input']) # 这里是需要用户输入数字的地方
                    self.context[var_name] = left + right
                except ValueError:
                    # 如果是充值或其他非数字输入，跳过加法运算
                    if '充值' in self.context['user_input']:
                        print(" 正在处理充值，请输入金额。")
                    else:
                        print(f" 无效输入：'{self.context['user_input']}'，无法进行加法运算。")
```

功能说明：该方法处理 ‘go‘（模式切换）和 ‘set‘（变量赋值）操作。当遇到模式切换时，更新当前模式；当遇到变量赋值时，根据表达式进行计算（如加法运算）。如果计算过程中有无效输入，会跳过计算并提示用户。

运行解释器与用户交互

```
    def run(self):
        # 运行与用户交互的循环
        while True:
            user_input = input(" 请输入您的问题: ")
            if user_input.lower() == "exit":
                break
            print(self.process_input(user_input))

# 接口方法
def run_interpreter(ast, balance=0.0):
    interpreter = Interpreter(ast, balance)
    interpreter.run()
```

功能说明：‘run‘方法启动一个交互式循环，持续接收用户输入并根据输入进行处理。当用户输入‘exit‘时，终止循环。‘run_interpreter‘方法是外部接口，用于初始化解释器并开始交互。

4.3.3 总结

该代码实现了一个功能全面的解释器，能够处理用户输入并根据解析的抽象语法树（AST）执行不同模式下的逻辑。它通过条件判断、变量操作和模式切换，实现了灵活的用户交互。整体结构清晰，易于扩展，可以方便地添加新的操作模式和交互功能。

4.4 gui.py

4.4.1 设计思路

在本项目中，设计的主要目标是实现一个用户与客服机器人进行交互的界面。我们希望通过图形用户界面（GUI）让用户可以方便地输入问题，并通过脚本控制机器人进行适当的回答。为了实现这一目标，我们使用了 Tkinter 库来创建图形界面，结合自定义的 DSL 脚本解析和解释逻辑，最终实现了一个功能丰富且易于使用的机器人界面。

4.4.2 代码实现

创建图形用户界面（GUI） 首先，我们需要创建一个基本的 GUI 界面，用户可以在其中输入消息并接收机器人的回应。

```
import os
import tkinter as tk
```

```
3  from tkinter import scrolledtext
4  from tkinter import messagebox
5  from PIL import Image, ImageTk  # 导入 Pillow 库
6  from dsl.lexer import Lexer
7  from dsl.parser import Parser
8  from dsl.interpreter import Interpreter
9
10 class ChatbotGUI:
11     def __init__(self, root):
12         self.root = root
13         self.root.title("在线客服机器人")
14         self.root.geometry("500x600")
15
16         self.chat_box = scrolledtext.ScrolledText(self.root, height=25, width=60, wrap=tk.WORD, state=tk.D
17         self.chat_box.grid(row=0, column=0, padx=10, pady=10)
18
19         self.user_input = tk.Entry(self.root, width=50)
20         self.user_input.grid(row=1, column=0, padx=10, pady=10)
21
22         self.send_button = tk.Button(self.root, text="发送", width=10, command=self.send_message)
23         self.send_button.grid(row=2, column=0, padx=10, pady=10)
24
25         self.script_input = tk.Entry(self.root, width=50)
26         self.script_input.grid(row=3, column=0, padx=10, pady=10)
27         self.script_input.insert(0, "scripts/example1.dsl")  # 默认文件路径
28
29         self.load_button = tk.Button(self.root, text="加载脚本", width=10, command=self.load_script)
30         self.load_button.grid(row=4, column=0, padx=10, pady=10)
```

功能说明：该代码段创建了聊天界面的基本布局。包括一个显示聊天记录的滚动文本框（'scrolledtext.ScrolledText'），一个用户输入框（'tk.Entry'），以及发送按钮（'tk.Button'）。用户可以在输入框中输入消息，点击发送按钮后，消息将发送至聊天框中。

发送消息并展示聊天内容 接下来，我们实现了 'send_message' 方法，用于处理用户输入的消息，并调用解释器处理脚本逻辑后显示机器人回应。

```
1  def send_message(self):
2      user_text = self.user_input.get()
```

```
3
4       if user_text.lower() == "exit":
5           self.chat_box.config(state=tk.NORMAL)
6           self.chat_box.insert(tk.END, "退出聊天模式\n")
7           self.chat_box.config(state=tk.DISABLED)
8           self.root.quit()
9           return
10
11      if self.interpreter:
12          if self.interpreter.context['current_mode'] == 'ACCOUNT' and "充值" in user_text:
13              self.prompt_for_recharge() # 弹出充值窗口
14              response = None # 不需要显示机器人回复
15          else:
16              response = self.interpreter.process_input(user_text) # 修改为 process_input
17
18          if response is not None:
19              self.chat_box.config(state=tk.NORMAL)
20              self.create_message_box("您", self.user_avatar, user_text)
21              self.create_message_box("机器人", self.bot_avatar, response)
22              self.chat_box.config(state=tk.DISABLED)
23              self.chat_box.see(tk.END)
24
25      else:
26          self.chat_box.config(state=tk.NORMAL)
27          self.chat_box.insert(tk.END, "请先加载一个脚本。\n")
28          self.chat_box.config(state=tk.DISABLED)
29      self.user_input.delete(0, tk.END)
```

功能说明： ‘send_message‘ 方法获取用户输入的文本，并通过解释器处理。如果用户输入”exit”，则退出程序。如果解释器已加载，方法会调用 ‘process_input‘ 处理用户的请求，并显示机器人的回应。如果解释器未加载，则提示用户先加载脚本。

创建消息框及显示用户与机器人对话内容 为了让聊天界面更加友好，我们定义了一个 ‘create_message_box‘ 方法，用于将用户与机器人的消息按格式显示在聊天框中，并附带头像。

```
1   def create_message_box(self, name, avatar, message):
2       message_frame = tk.Frame(self.chat_box, padx=10, pady=5, relief=tk.RAISED, bd=2, bg='#f0f0f0')
3       message_content_frame = tk.Frame(message_frame, bg='#f0f0f0')
```

```
4
5       avatar_label = tk.Label(message_content_frame, image=avatar, bg='#f0f0f0')
6       avatar_label.grid(row=0, column=0, padx=5, pady=5, sticky="w")
7
8       message_label = tk.Label(message_content_frame, text=message, bg='#f0f0f0', width=40, wraplengt
9       message_label.grid(row=0, column=1, padx=5, pady=5)
10
11      message_content_frame.pack(anchor="w", padx=5, pady=2)
12
13      name_label = tk.Label(message_frame, text=name, bg='#f0f0f0', font=("Arial", 10, "bold"))
14      name_label.pack(anchor="w", padx=5, pady=2)
15
16      self.chat_box.window_create(tk.END, window=message_frame)
17      self.chat_box.insert(tk.END, "\n")
```

功能说明： ‘create_message_box‘ 方法将每条消息与头像放入一个独立的框架中，以便整齐显示。头像显示在左侧，消息显示在右侧，并且每个消息框下方会显示发送者的名字。该方法有助于使聊天记录的显示更加整洁和直观。

加载脚本并初始化解释器 为了使聊天机器人能够根据不同的脚本执行任务，我们实现了 ‘load_script‘ 和 ‘execute_script‘ 方法，用于加载外部脚本并初始化解释器。

```
1   def load_script(self):
2       script_file = self.script_input.get().strip()
3       if not script_file:
4           messagebox.showerror("错误", "脚本文件路径不能为空！")
5           return
6
7       script_code = self.load_script_from_file(script_file)
8       if script_code:
9           self.chat_box.config(state=tk.NORMAL)
10          self.chat_box.insert(tk.END, f"加载脚本: {script_file}\n")
11          self.chat_box.config(state=tk.DISABLED)
12          self.interpreter = self.execute_script(script_code)
13
14  def load_script_from_file(self, file_path):
15      if not os.path.exists(file_path):
16          messagebox.showerror("错误", f"脚本文件 '{file_path}' 不存在。")
```

```
17            return None
18        with open(file_path, 'r', encoding='utf-8') as file:
19            return file.read()
20
21    def execute_script(self, script_code):
22        lexer = Lexer(script_code)
23        tokens = lexer.tokenize()
24
25        parser = Parser(tokens)
26        ast = parser.parse()
27
28        interpreter = Interpreter(ast)
29        return interpreter
```

功能说明： 这些方法用于加载脚本并初始化解释器。‘load_script‘ 获取脚本文件路径，并调用 ‘load_script_from_file‘ 加载脚本内容。‘execute_script‘ 使用词法分析器（Lexer）将脚本转换为标记，并通过语法解析器（Parser）生成抽象语法树（AST），最终通过解释器执行脚本。

充值功能（示例扩展） 为了进一步增强聊天机器人的功能，我们增加了充值功能。当用户处于账户模式并输入” 充值” 时，弹出一个充值窗口让用户输入金额。

```
1     def prompt_for_recharge(self):
2         def submit_recharge():
3             try:
4                 recharge_amount = float(recharge_entry.get())
5                 if recharge_amount < 0:
6                     messagebox.showerror(” 错误”, ” 金额不能为负！ ”)
7                     return
8                 self.chat_box.config(state=tk.NORMAL)
9                 self.chat_box.insert(tk.END, f” 充值成功！ 充值金额为: {recharge_amount:.2f} 元\n”)
10                self.chat_box.config(state=tk.DISABLED)
11                recharge_window.destroy()
12            except ValueError:
13                messagebox.showerror(” 错误”, ” 请输入有效的充值金额。”)
14
15        recharge_window = tk.Toplevel(self.root)
16        recharge_window.title(” 充值”)
17        recharge_label = tk.Label(recharge_window, text=” 请输入充值金额： ”)
```

```
18          recharge_label.pack(padx=10, pady=10)
19          recharge_entry = tk.Entry(recharge_window)
20          recharge_entry.pack(padx=10, pady=10)
21          submit_button = tk.Button(recharge_window, text="提交", command=submit_recharge)
22          submit_button.pack(pady=10)
```

功能说明： ‘prompt_for_recharge‘ 方法弹出一个新窗口，允许用户输入充值金额，并验证金额的有效性。如果充值成功，更新聊天记录并关闭窗口。

4.4.3 总结

通过本项目的实现，我们展示了如何利用 Tkinter 库创建图形用户界面，并通过自定义 DSL 脚本解析和执行实现了一个功能丰富的聊天机器人。系统不仅支持用户的简单交互，还能根据不同的脚本文件执行复杂的逻辑。未来，可以进一步优化机器人逻辑和界面设计，增加更多智能化的功能，例如语音识别、自然语言处理等。

4.5 main.py

```
1  # main.py
2  from ui.gui import main  # 从 gui.py 中导入 main 函数
3
4  if __name__ == "__main__":
5      main()  # 启动应用
```

该 main.py 文件的作用是作为程序的入口。它会调用 gui.py 中定义的 main() 函数来启动应用。

5 接口描述

本节将介绍 lexer.py、parser.py、interpreter.py、gui.py 和 main.py 的接口设计与功能描述。这些模块共同构成了程序的核心架构，负责处理用户输入、解析和执行脚本以及通过图形界面展示结果。

5.1 lexer.py 接口描述

作用： lexer.py 主要负责将输入的脚本文本转换成一系列的词法单元（tokens），这些词法单元会被传递到后续的解析阶段。

主要类：

- Lexer

方法：

- tokenize()

该方法负责扫描输入的脚本代码并生成词法单元（tokens）。返回值是一个包含所有词法单元的列表。

示例用法：

```
lexer = Lexer(script_code)
tokens = lexer.tokenize()
```

功能描述：

- Lexer 类通过正则表达式对输入脚本进行逐字符扫描，识别出语言中的关键字、变量、数字等词法单元。
- 返回的 tokens 列表将作为 parser.py 中解析器的输入。

5.2 parser.py 接口描述

作用： parser.py 负责对 lexer.py 提供的词法单元进行语法分析，生成抽象语法树（AST），为后续的执行阶段做准备。

主要类：

- Parser

方法：

- parse()

该方法负责解析词法单元列表，生成抽象语法树（AST）。它会递归地分析并构造 AST 节点，最终返回一个完整的抽象语法树。

示例用法：

```
parser = Parser(tokens)
ast = parser.parse()
```

功能描述：

- Parser 类根据预定义的文法规则（通常是上下文无关文法）对词法单元进行解析。
- 解析过程中，生成 AST 来表示脚本的结构。AST 方便后续的执行过程中的逻辑操作。

5.3 interpreter.py 接口描述

作用： interpreter.py 负责执行通过 parser.py 生成的抽象语法树（AST），并与外部环境（如用户输入、脚本上下文）进行交互，产生最终的输出。

主要类：

- Interpreter

属性：

- context：保存脚本执行过程中的上下文数据，包含当前状态（如余额、用户信息等）。
- ast：保存解析后的抽象语法树，执行时会根据此树进行操作。

方法：

- process_input(user_input)
- execute_script()

process_input 方法根据用户输入和当前脚本上下文进行脚本执行。返回结果可以是一个响应消息、状态更新或其他反馈。

示例用法：

```
interpreter = Interpreter(ast, balance)
response = interpreter.process_input(user_text)
```

功能描述：

- Interpreter 类根据输入的 AST 执行脚本逻辑。它处理各种操作符、条件判断、变量赋值等，并根据脚本中的语义更新程序状态。
- process_input 方法允许用户与脚本进行交互，并且根据上下文信息生成适当的响应。
- context 属性用于存储执行过程中的环境信息，这些信息会被用来决定后续操作。

5.4 gui.py 接口描述

作用： gui.py 负责创建并管理应用程序的图形用户界面，用户通过 GUI 输入脚本，并通过该界面与程序进行交互。

主要类：

- ChatbotGUI

方法：

- send_message()：发送用户输入并获取机器人回应。
- load_script()：加载用户输入的脚本文件。
- load_script_from_file()：从脚本文件加载代码。
- execute_script()：执行 DSL 脚本并返回解释器。

示例用法：

```
from ui.gui import main

main()
```

功能描述：

- ChatbotGUI 类是图形用户界面的核心，负责初始化界面元素，监听用户输入，并通过事件处理器响应用户操作。
- 用户输入通过程序主窗口调用一些主要方法实现图形化逻辑并正确反馈 dsl 脚本。

5.5 main.py 接口描述

作用： main.py 是应用程序的入口，负责启动整个程序，导入并运行 gui.py 中的图形用户界面。它是用户与程序交互的接口。

功能：

- 导入并调用 gui.py 中的 main 函数，启动图形界面并等待用户输入。

主要功能：

- 启动应用：main.py 通过调用 gui.py 中的 main() 函数，启动了图形界面界面（GUI），并在其中创建用户交互界面。
- 当用户输入脚本时，main.py 将调用 lexer.py、parser.py 和 interpreter.py 进行处理，并展示结果。

示例用法：

```
1 from ui.gui import main
2
3 if __name__ == "__main__":
4     main()  # 启动应用
```

功能描述：

- main.py 中的 main() 函数是程序的入口点，它会加载和启动整个 GUI 应用，连接到后端处理逻辑（如脚本执行、交互）。
- 在执行过程中，用户通过 GUI 输入命令，main.py 会启动相关的解释和执行模块来响应用户操作。

5.6 总结

通过对各个模块（lexer.py、parser.py、interpreter.py 和 main.py）的接口描述，可以看出整个应用的架构是清晰的、模块化的。每个模块负责处理特定的功能，分工明确且紧密协作：

- lexer.py 将脚本文本转换为可处理的词法单元。
- parser.py 对词法单元进行语法分析，生成抽象语法树（AST）。
- interpreter.py 负责执行 AST，处理脚本逻辑并与用户进行交互。
- main.py 是程序的入口，负责启动图形用户界面并连接各模块的功能。

这种分层架构使得程序易于维护和扩展，同时也保证了每个模块的功能能够单独测试和调试。

6 测试说明

6.1 程序测试集

本次实验为程序 lexer.py、parser.py 和 interpreter.py 设计了程序测试集，分别是 test_lexer.py、test_parser.py 和 test_interpreter.py。

6.1.1 test_lexer.py

test_lexer.py 是一个针对 lexer.py 中 Lexer 类的单元测试文件，旨在验证词法分析（tokenization）功能是否正常工作。该文件使用了 unittest 框架进行测试，涵盖了多种不同类型的输入，并检查了解析后的 token 是否符合预期。

主要功能：

- 测试了数字、字符串、关键字、赋值语句等基本语法的词法分析。
- 验证了多种不同 token 类型的处理，包括关键字（如 if、elif、response）和标识符（如 x、INIT）。
- 测试了换行符和空格的处理，确保这些字符被正确跳过，并且有效代码被正确识别。
- 验证了注释的处理，确保注释部分被忽略，仅处理有效的代码部分。

各个测试函数：

- test_tokenize_number: 测试数字（如 123）是否正确被识别为 NUMBER 类型。
- test_tokenize_string: 测试字符串（如 "hello world"）是否正确被识别为 STRING 类型。
- test_tokenize_if: 测试关键字 if 是否被正确识别为 IF 类型。
- test_tokenize_elif: 测试关键字 elif 是否被正确识别为 ELIF 类型。
- test_tokenize_keyword_response: 测试关键字 response 是否被正确识别为 RESPONSE 类型。
- test_tokenize_assignment: 测试赋值语句（如 set x = 10）是否正确被识别为 SET、ID、ASSIGN 和 NUMBER 类型。
- test_tokenize_go_mode: 测试 go INIT 模式是否正确被识别为 GO 和 MODE 类型。
- test_tokenize_multiple_tokens: 测试多个不同的 token（如 if x in user_input then response "hello"）是否被正确识别。
- test_tokenize_newline_and_skip: 测试换行符和空白字符的跳过功能，确保只处理有效代码。
- test_tokenize_comment: 测试注释的跳过，确保注释内容不影响后续的词法分析。

test_lexer.py 文件确保了 Lexer 类对各种输入的词法分析行为是正确的，并能够正确地跳过无效字符（如注释、空格、换行符），处理标准语法，并返回预期的 token 列表。

6.1.2 test_parser.py

test_parser.py 文件包含了对 dsl.parser 模块中 Parser 类的单元测试，主要测试了不同代码结构的解析逻辑以及对各种语法和逻辑错误的处理。测试覆盖了以下几种情况：

- **test_no_init_mode:** 测试代码中缺少 INIT 模式，确保在缺少 INIT 模式时抛出 SyntaxError 错误。
- **test_no_end:** 测试代码中缺少 end 关键字，缺少 end 时应抛出 TypeError 错误，因为 end 是必需的。
- **test_invalid_token:** 验证遇到无效的标记（如 sfdsa）时，能够正确抛出 SyntaxError 错误。
- **test_multi_init:** 确保在 INIT 模式重复出现时抛出 SyntaxError 错误。

- **test_parse_if_statement:** 测试解析 if 语句，验证生成的抽象语法树（AST）是否正确处理条件和响应。
- **test_parse_elif_statement:** 验证解析 elif 语句时，紧跟在 if 语句后面时是否能正确生成 AST 结构。
- **test_parse_if_else_statement:** 测试解析 if-else 语句，确保生成的 AST 结构正确反映了 if-else 的流程。
- **test_parse_if_elif_else_statement:** 确保解析 if-elif-else 语句链时，能够正确生成 AST。
- **test_parse_assignment_statement:** 验证解析赋值语句时，AST 是否能正确反映赋值操作。
- **test_parse_go_statement:** 测试解析 go 语句及模式切换，确保能够正确处理不同模式之间的流程切换，例如从 INIT 切换到 TEMP 模式。

每个测试用例都确保解析器能正确地将 token 转换为抽象语法树（AST），并且能够正确反映代码的结构和语义。此外，对于缺少必要关键字或无效标记等错误，也能够正确地抛出相应的异常。

6.1.3 test_interpreter.py

test_interpreter.py 文件中包含了对 Interpreter 类的单元测试，主要测试了在不同模式下的交互逻辑，以及对用户输入的处理。测试覆盖了多个模式（INIT、ACCOUNT、GOODS、QUERY、WEATHER）以及不同的用户输入，确保解释器能够正确地根据状态变化给出合适的响应。

- **test_initial_state:** 测试在初始状态下，用户输入的不同内容（如” 你好”、” 账户”、” 商品” 等）是否能正确触发对应的响应和状态转换。
- **test_account_mode:** 测试账户模式下的交互，验证账户余额、充值、退出等操作是否能正确处理，并确保在状态转换时给出适当的反馈。
- **test_goods_mode:** 测试商品模式下的交互，检查商品名称查询、商品查询模式切换等操作是否能正常工作。
- **test_query_mode:** 测试商品查询模式下，用户输入商品名称（如” 商品 A”）时是否能正确返回商品信息，并测试退出查询模式的操作。
- **test_recharge:** 通过测试天气查询模式，验证用户查询天气的流程是否正确，确保能根据不同城市返回天气信息，并支持退出天气查询模式。

每个测试用例通过模拟用户输入，确保解释器能够根据不同的 DSL 脚本逻辑给出正确的响应，并能在不同模式间正确切换，保证系统的交互性和准确性。以下是各个测试用例的描述：

- **test_initial_state:** 在初始状态（INIT）下，用户输入“你好”时，系统会返回“您好，很高兴为您服务，请问您的需要是”；输入“账户”时，切换到账户模式；输入“商品”时，切换到商品模式；输入“退出”时，退出账户模式并切换回 INIT 状态。
- **test_account_mode:** 测试账户模式下，输入“余额”时返回当前余额，输入“充值”时模拟充值操作，输入“退出”时返回退出账户模式的反馈。
- **test_goods_mode:** 在商品模式下，测试用户输入“名称”时返回商品名称列表，输入“查询”时切换到查询模式，输入“退出”时退出商品模式。
- **test_query_mode:** 测试在查询模式下，用户输入具体商品名称（如“商品 A”）时返回商

品的详细信息，输入“退出”时退出查询模式。

- **test_recharge:** 测试天气查询模式下，用户输入“天气查询”时，系统会引导用户输入城市名称，输入“北京”时返回北京的天气信息，输入“退出”时退出天气查询模式。

每个测试用例确保了不同状态下的用户输入能得到正确的解析和处理，验证了解释器在执行过程中能够正确处理和转换状态，确保代码逻辑的准确性和稳定性。

6.2 性能测试

该代码段的功能是进行性能测试和响应时间统计，通过多次模拟 DSL 代码的生成、词法分析、语法解析和解释执行，评估系统的响应时间和吞吐量。主要流程包括生成 DSL 代码、随机用户输入以及性能统计信息的记录。接下来，我们将代码拆分为几个片段并逐一描述其功能。

6.2.1 生成随机字符串和随机模块名称

第一个代码片段用于生成随机的字符串和模块名称，它们会在 DSL 代码中使用，确保每次生成的代码都是唯一的。

```
import random
import string

# 生成一个长度为`length`的随机字符串
def generate_long_string(length=2000):
    return ''.join(random.choices(string.ascii_letters + string.digits, k=length))


# 随机生成全大写字母组成的模块名称
def generate_random_mode_name(length=2000):
    return ''.join(random.choices(string.ascii_uppercase, k=length))
```

功能描述：- ‘generate_long_string(length)‘：生成一个指定长度的随机字符串，包含字母和数字。- ‘generate_random_mode_name(length)‘：生成一个全大写字母组成的随机模块名称。
这些函数用于在 DSL 代码中插入随机值，模拟多变的输入。

6.2.2 生成随机变量名

第二个代码片段用于生成随机变量名。变量名以字母 ‘z‘ 开头，并且后续字符为字母和数字的随机组合。

```
# 随机生成一个以'z'字母开头的变量名
def generate_variable_name(length=1000):
```

```
    return 'z' + ''.join(random.choices(string.ascii_letters + string.digits, k=length - 1))
```

功能描述：- ‘generate_variable_name(length)‘：生成一个指定长度的以字母 ‘z‘ 开头的随机变量名，用于在 DSL 代码中动态地命名变量。

6.2.3 生成随机用户输入

该片段用于生成随机的用户输入，模拟不同的用户请求。

```
# 随机生成英文用户输入
def generate_random_user_input():
    inputs = [
        "Account Balance", "Recharge", "Order Inquiry", "Shopping", "Payment", "Exit",
        "Insufficient Balance", "Order Details", "Check Account", "Order Payment",
        "View Balance", "Account Info", "Order Status", "Payment Successful", "Confirm Payment"
    ]
    return random.choice(inputs)
```

功能描述：- ‘generate_random_user_input()‘：从预定义的用户输入列表中随机选择一个输入，模拟用户请求的不同类型。

6.2.4 生成 DSL 代码

这个片段的功能是模拟生成 DSL 代码，通过调用前面定义的函数来构建 DSL 代码段。

```
# 模拟生成一些随机 DSL 代码
def generate_dsl():
    second_mode = generate_random_mode_name()

    dsl_code = f"start\n"
    dsl_code += f"    INIT\n"

    # 在 INIT 模式下，生成 if 语句
    dsl_code += f"        if \"{generate_long_string()}\" in user_input then\n"
    dsl_code += f"            response \"{generate_long_string()}\"\n"
    dsl_code += f"            set {generate_variable_name()} = {random.randint(1, 10000000000)}\n"
    dsl_code += f"            go {second_mode}\n"

    # 随机生成第二个模式
```

```
    dsl_code += f"    {second_mode}\n"
    dsl_code += f"        if \"{generate_long_string()}\" in user_input then\n"
    dsl_code += f"            response \"{generate_long_string()}\"\n"
    dsl_code += f"        elif \"{generate_long_string()}\" in user_input then\n"
    dsl_code += f"            response \"{generate_long_string()}\"\n"
    dsl_code += f"        else\n"
    dsl_code += f"            response \"What{generate_long_string()} can I{generate_long_string()} say? M
    dsl_code += f"            set {generate_variable_name()} = {random.randint(1, 10000000000)}\n"
    dsl_code += f"            go INIT\n"

    dsl_code += "    end\n"
    return dsl_code
```

功能描述：- ‘generate_dsl()‘：生成一段完整的 DSL 代码，包括初始化模式（‘INIT‘）、条件判断、响应生成、变量赋值等逻辑。代码中会插入之前生成的随机字符串、模块名称、变量名等内容。

6.2.5 性能测试函数

这个片段定义了性能测试的主函数，它模拟一定次数的请求，记录每个请求的响应时间，统计成功的请求数量，并输出性能总结。

```
# 性能测试和响应时间统计
def performance_test(num_iterations=1000):
    total_time = 0  # 总共耗时
    max_time = 0  # 最大响应时间
    min_time = float('inf')  # 最小响应时间
    successful_requests = 0  # 成功请求数

    for i in range(num_iterations):
        start_time = time.time()  # 记录请求开始时间

        # 每次请求生成新的 DSL 代码
        dsl_code = generate_dsl()
        lexer = Lexer(dsl_code)
        tokens = lexer.tokenize()

        parser = Parser(tokens)
        ast = parser.parse()

```

```
19          interpreter = Interpreter(ast)
20
21          # 生成随机用户输入
22          user_input = generate_random_user_input()
23          print(f"Test Iteration {i + 1}: {user_input}")
24          response = interpreter.process_input(user_input)
25
26          # 计算每次请求的响应时间
27          response_time = time.time() - start_time
28          total_time += response_time
29          max_time = max(max_time, response_time)
30          min_time = min(min_time, response_time)
31
32          # 统计成功请求
33          if response is not None:
34              successful_requests += 1
35
36          # 每 100 次打印一次响应和性能数据
37          if i % 100 == 0:
38              print(f"Iteration {i + 1} Response: {response}")
39              print(f"Iteration {i + 1} Response Time: {response_time:.4f} seconds")
40
41      # 计算平均响应时间和吞吐量
42      avg_time = total_time / num_iterations
43      throughput = successful_requests / total_time if total_time > 0 else 0
44
45      # 打印总结信息
46      print("\nPerformance Test Summary:")
47      print(f"Total Requests: {num_iterations}")
48      print(f"Successful Requests: {successful_requests}")
49      print(f"Average Response Time: {avg_time:.4f} seconds")
50      print(f"Max Response Time: {max_time:.4f} seconds")
51      print(f"Min Response Time: {min_time:.4f} seconds")
52      print(f"Throughput: {throughput:.2f} requests/second")
53
54
55  if __name__ == '__main__':
56      # 进行性能测试，模拟 1000 次迭代
```

```
57     performance_test(num_iterations=1000)
```

功能描述：- ‘performance_test(num_iterations)‘：执行性能测试，模拟一定数量（‘num_iterations‘）的 DSL 请求，记录每次请求的响应时间，并统计成功请求的数量。最终输出性能总结，包括总请求数、成功请求数、平均响应时间、最大响应时间、最小响应时间以及吞吐量。

性能测试的主要目标是通过生成随机 DSL 代码并进行词法分析、解析和解释，来测试系统的性能，确保它在高负载情况下能够正确响应。通过多次迭代，收集响应时间等指标，帮助开发者了解系统在实际应用中的表现。

6.3 压力测试

该代码段的功能是进行压力测试，模拟系统在高负载情况下的表现，并监控系统资源的占用情况，如 CPU、内存、磁盘、网络等。通过多次生成 DSL 代码并执行词法分析、语法解析和解释执行，模拟在高压力下系统的响应时间和资源占用，确保系统的稳定性和性能。以下将代码拆分为多个片段并逐一描述其功能。

6.3.1 生成随机字符串和模块名称

第一个代码片段用于生成随机字符串和模块名称，模拟不同的用户输入和 DSL 代码内容，确保每次生成的代码都是唯一的。

```
1  # 生成一个长度为`length`的随机字符串
2  def generate_long_string(length=2000):
3      return ''.join(random.choices(string.ascii_letters + string.digits, k=length))
4
5
6  # 随机生成全大写字母组成的模块名称
7  def generate_random_mode_name(length=2000):
8      return ''.join(random.choices(string.ascii_uppercase, k=length))
```

功能描述：- ‘generate_long_string(length)‘：生成一个指定长度的随机字符串，包含字母和数字，用于在 DSL 代码中插入随机值。- ‘generate_random_mode_name(length)‘：生成一个全大写字母组成的随机模块名称，用于模拟不同的 DSL 模式。

6.3.2 生成随机变量名

第二个代码片段用于生成随机变量名，模拟动态变量名的生成，确保每次生成的代码包含不同的变量名。

```
# 随机生成一个以'z'字母开头的变量名
def generate_variable_name(length=1000):
    return 'z' + ''.join(random.choices(string.ascii_letters + string.digits, k=length - 1))
```

功能描述：- ‘generate_variable_name(length)‘：生成一个以字母 ‘z‘ 开头的随机变量名，用于在 DSL 代码中作为变量的名称。

6.3.3 生成随机用户输入

该片段模拟了生成随机的用户输入，用于测试系统在处理不同类型输入时的表现。

```
# 随机生成英文用户输入
def generate_random_user_input():
    inputs = [
        "Account Balance", "Recharge", "Order Inquiry", "Shopping", "Payment", "Exit",
        "Insufficient Balance", "Order Details", "Check Account", "Order Payment",
        "View Balance", "Account Info", "Order Status", "Payment Successful", "Confirm Payment"
    ]
    return random.choice(inputs)
```

功能描述：- ‘generate_random_user_input()‘：从预定义的用户输入列表中随机选择一个，模拟用户在系统中输入的不同请求。

6.3.4 生成 DSL 代码

该片段用于模拟生成 DSL 代码，调用之前定义的函数，构建不同的 DSL 代码段。

```
# 模拟生成一些随机 DSL 代码
def generate_dsl():
    second_mode = generate_random_mode_name()

    dsl_code = f"start\n"
    dsl_code += f"    INIT\n"

    # 在 INIT 模式下，生成 if 语句
    dsl_code += f"        if \"{generate_long_string()}\" in user_input then\n"
    dsl_code += f"            response \"{generate_long_string()}\"\n"
    dsl_code += f"            set {generate_variable_name()} = {random.randint(1, 10000000000)}\n"
    dsl_code += f"            go {second_mode}\n"
```

```
13
14      # 随机生成第二个模式
15      dsl_code += f"    {second_mode}\n"
16      dsl_code += f"        if \"{generate_long_string()}\" in user_input then\n"
17      dsl_code += f"            response \"{generate_long_string()}\"\n"
18      dsl_code += f"        elif \"{generate_long_string()}\" in user_input then\n"
19      dsl_code += f"            response \"{generate_long_string()}\"\n"
20      dsl_code += f"        else\n"
21      dsl_code += f"            response \"What{generate_long_string()} can I{generate_long_string()} say? M
22      dsl_code += f"            set {generate_variable_name()} = {random.randint(1, 10000000000)}\n"
23      dsl_code += f"            go INIT\n"
24
25      dsl_code += "    end\n"
26      return dsl_code
```

功能描述：- ‘generate_dsl()‘：生成一段包含多个模式、条件判断、响应和变量赋值的 DSL 代码，插入之前生成的随机字符串、模块名称和变量名。

6.3.5 压力测试函数

该片段执行压力测试，模拟多次请求和资源占用情况，记录每次请求的响应并监控系统资源占用。

```
1   # 模拟压力测试和资源占用
2   def stress_test(num_iterations=1000):
3       for i in range(num_iterations):
4           # 每次压力测试都生成新的 DSL 代码
5           dsl_code = generate_dsl()
6           lexer = Lexer(dsl_code)
7           tokens = lexer.tokenize()
8
9           parser = Parser(tokens)
10          ast = parser.parse()
11
12          interpreter = Interpreter(ast)
13
14          # 生成随机用户输入
15          user_input = generate_random_user_input()
16          print(f"Test Iteration {i + 1}: {user_input}")
```

```
        response = interpreter.process_input(user_input)

        # 每 100 次打印一次响应
        if i % 100 == 0:
            print(f"Iteration {i + 1} Response: {response}")

        # 模拟资源占用情况
        simulate_cpu_memory_disk_network()

        # 每 100 次打印一下压力测试状态
        if i % 100 == 0:
            print(f"Iteration {i + 1} resource stress simulated.")

    print("Stress Test completed.")
```

功能描述：- ‘stress_test(num_iterations)‘：执行多次（‘num_iterations‘）压力测试，每次生成新的 DSL 代码并进行词法分析、解析和执行，同时模拟系统的资源占用情况（CPU、内存、磁盘、网络等）。

6.3.6 模拟资源占用情况

该片段模拟了系统资源的高占用情况，监控 CPU、内存、磁盘和网络使用，并启动占用资源的进程来测试系统在高负载下的表现。

```
# 模拟资源占用情况
def simulate_cpu_memory_disk_network():
    # 模拟占用 CPU
    print("Simulating high CPU usage...")
    cpu_usage = psutil.cpu_percent(interval=2)
    print(f"CPU Usage: {cpu_usage}%")

    # 模拟占用内存
    print("Simulating high memory usage...")
    memory_usage = psutil.virtual_memory().percent
    print(f"Memory Usage: {memory_usage}%")

    # 模拟磁盘占用
    print("Simulating high disk usage...")
    disk_usage = psutil.disk_usage('/').percent
```

```
16      print(f"Disk Usage: {disk_usage}%")
17
18      # 模拟网络带宽占用
19      print("Simulating network usage...")
20      network_usage = psutil.net_io_counters()
21      print(f"Sent: {network_usage.bytes_sent / (1024  2):.2f} MB, Received: {network_usage.bytes_recv / (10
22
23      # 模拟一个 CPU 占用高的进程
24      print("Launching a high CPU process...")
25      subprocess.Popen(["python", "-c", "while True: pass"]) # 启动一个占用 CPU 的进程
26
27      # 模拟一个大内存占用的进程
28      print("Launching a high memory process...")
29      memory_hog = subprocess.Popen(["python", "-c", "a = ['a' * 107 for _ in range(100)]"]) # 启动一个占用
30
31      # 模拟磁盘占用接近 100%
32      print("Simulating disk full condition...")
33      with open('large_test_file.txt', 'w') as f:
34          f.write('a' * 109) # 写入一个大文件，占用磁盘空间
35
36      # 模拟网络带宽占用的进程（例如，下载一个大文件）
37      print("Simulating network bandwidth usage...")
38      subprocess.Popen(["curl", "-O", "https://speed.hetzner.de/100MB.bin"]) # 使用 curl 下载一个大文件
39
40      # 模拟网络断开
41      print("Simulating network disconnect...")
42      os.system("ifconfig eth0 down") # 断开网络
43      time.sleep(2)
44      os.system("ifconfig eth0 up") # 恢复网络
45
46      # 模拟电源故障
47      print("Simulating power failure (mock)...")
48      # 这里是模拟一个设备重启的情况
49      time.sleep(1)
50      print("System rebooted.") # 模拟系统重启
```

功能描述：- ‘simulate_cpu_memory_disk_network()‘：模拟多个资源占用场景，包括 CPU 占用、内存占用、磁盘占用、网络带宽占用，并通过启动占用资源的进程来增加系统负载。

通过以上压力测试和资源占用模拟，可以检测系统在高负载情况下的稳定性和性能，确保系统在复杂的 DSL 代码执行和多次请求处理中保持良好的响应和资源管理能力。