基于领域特定语言的客服机器人设计与实现

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。

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。

实现

本程序采用rust进行实现，因为rust具有一些显著优点：

1. 具有包管理器和配置工具cargo
2. 内置单元测试和集成测试，而且通过cargo可以自动运行所有测试桩，不用找其他工具进行额外配置
3. 严格的编译期管理和检查，对于每个不符合rust码风要求的地方都会warning来提醒里改善代码，使得该项目代码有着良好的风格

脚本文法介绍

符号说明

- 关键词：
  - `global`:用于声明全局变量
  - `speak`:用于输出机器人回答
  - `input`:用于获取用户输入
  - `if` :用于条件判断

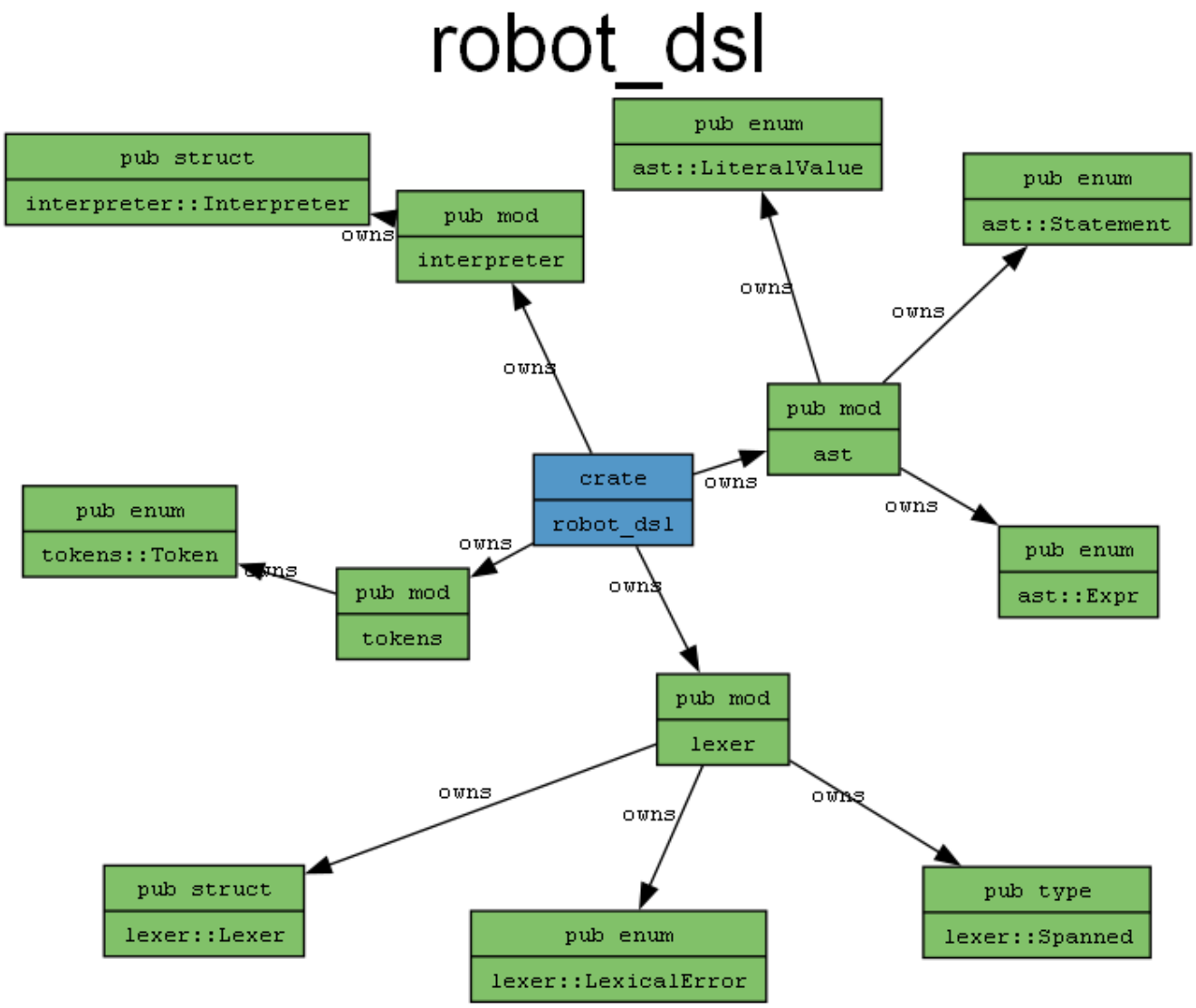
    - exit:用于终止程序
    - loop:用于进行无限循环
- 标识符:以字母开头，其余可为'_',字母和数字
- 数字:只支持十进制整数和小数
- 字符串:被""包围部分
- 注释: "#"后一整行

语句定义规则

```
program := globalvariable* mainloop

globalvariable := "global" identifier "=" (String|Number) ";"

mainloop := "loop" "{" expr* "}"

expr := "speak" expr ";"
     |  "input" identifier ";"
     |  "if" "(" expr ")" "{" expr "}"  ";"
     |  expr + expr
     |  expr – expr
     |  expr * expr
     |  expr / expr
     |  expr = expr
     |  expr == expr
     |  (expr)
     |  Number
     |  String
     |  identifier
     |  "exit"
```

解释器的实现思路

模块划分与功能实现过程



通过以下代码实现各模块之间合作：

```
let source_code = std::fs::read_to_string(args[1].clone())?;
let lexer = Lexer::new(&source_code[..]);
let parser = grammar::ProgramParser::new();
let ast = parser.parse(lexer)?;
let mut interpreter= Interpreter::new(ast);
interpreter.interpret();
```

由代码可知，解释器先调用 lexer 模块对脚本代码的词素进行扫描和分析后，再调用parser 模块进行解析，最后调用interpreter 模块进行解释，从而实现解释器的功能。

lexer实现

lexer部分本人采用了logos进行实现，直接返回分析后的token list。详见token.rs部分

parser实现

核心数据结构便是一棵抽象语法树(Abstract Syntax Tree,AST)，采用了智能指针Box<>,节点部分如下：

```
/*
 * 语法树中表达式的枚举类型，方便递归下降分析
 * 共有如下表达式：
 * - 变量
 * - 二元表达式
 * - 字面量
 * - 赋值语句
 */

#[derive(Debug, Clone, PartialEq)]
pub enum Expr {
    /* 赋值表达式 example: a=b+c */
    Assign {
        name: String,
        value: Box<Expr>,
    },
    /* 二元表达式 example: b+c */
    Binary {
        left: Box<Expr>,
        operator: Token,
        right: Box<Expr>,
    },
    /* 字面量 string或number的值 */
    Literal {
        value: LiteralValue,
    },
    /* example input x 中的x */
    Variable {
        name: String,
    },
}

/*
 * 字面量枚举类型，作为表达式中的字面量类型使用
 * 共有如下类型：
 * - 数字
 * - 字符串
 */

#[derive(Debug, Clone, PartialEq)]
pub enum LiteralValue {
    Number(f64),
    String(String),
```

```
}

impl LiteralValue {
    pub fn trans(&self) -> String {
        match self {
            LiteralValue::Number(n) => n.to_string(),
            LiteralValue::String(s) => s.clone(),
        }
    }
}

impl fmt::Display for LiteralValue {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            LiteralValue::Number(n) => write!(f, "{}", n),
            LiteralValue::String(s) => write!(f, "{}", s),
        }
    }
}

impl fmt::Display for Expr {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{:?}", self)
    }
}

/*
 * 语法树中语句的枚举类型
 * 共有如下语句:
 * - 表达式语句
 * - 打印语句(speak ...)
 * - 变量声明语句(global id=value)
 * - 输入语句(input id)
 * - 循环语句(loop{ code })
 * - 条件语句(if (expr) func();)
 * - 函数声明语句(fn id(){code})
 * - 退出语句(exit;)
 */

#[derive(Debug, Clone, PartialEq)]
pub enum Statement {
    /* 块语句 */
    Block {
```

```
        statements: Vec<Box<Statement>>,
    },
    /* 表达式语句 */
    Expression {
        expression: Box<Expr>,
    },
    /* 分支语句 */
    Branch {
        /// 分支语句中的条件表达式
        condition: Box<Expr>,
        /// 分支语句中的执行语句
        then: Box<Statement>,
    },
    /* mainloop 中的语句 */
    Loop {
        body: Box<Statement>,
    },
    /* 打印语句 */
    Speak {
        expression: Box<Expr>,
    },
    /* 输入字符串语句 */
    Input {
        /* 输入字符串语句中的变量名 */
        input: String,
    },
    /*  变量声明语句  */
    Var {
        /* 变量声明语句中的变量名 */
        name: String,
        /* 变量声明语句中的变量值 */
        init: Box<Expr>,
    },
    /// 退出语句
    Exit,
}
```

建立AST的部分利用了lalrpop进行实现,以下面这段为例进行分析:

```
pub Statement: Box<ast::Statement> = {
    "global" <name:"identifier"> "=" <init: Expression> ";" => {
      Box::new(ast::Statement::Var { name , init })
    },
    "if" "(" <condition:Expression> ")" <then:Block> ";" => {
        Box::new(ast::Statement::Branch{condition,then})
    },
    "loop" <body:Block>  =>{
        Box::new(ast::Statement::Loop{body})
    },
    "speak" <Expression> ";" =>{
        Box::new(ast::Statement::Speak{expression:<>})
    },
    "input" <input:"identifier"> ";" => {
        Box::new(ast::Statement::Input{input})
    },
    "exit" ";" => {
        Box::new(ast::Statement::Exit)
    },
    Block,
    <Expression> ";"=> {
        Box::new(ast::Statement::Expression {
            expression: <>
        })
    }
}
```

`"global" name:"identifier" "=" <init: Expression> ";" => { ... }`:这个分支表示一个全局变量声明语句，它以 `"global"` 关键字开始，后面是一个标识符 `name`、等号 `=`、以及一个表达式 `init`，然后以分号 `;` 结束。当这种类型的语句匹配时，它会创建一个 `ast::Statement::Var` 节点，其中包含了 `name` 和 `init`，并将其封装在 `Box` 中。

`"if" "(" condition:Expression ")" then:Block ";" => { ... }`:这个分支表示一个条件语句（if语句），它以 `"if"` 关键字开始，后跟括号中的条件表达式 `condition`、一个代码块 `then`，以及分号 `;`。当这种类型的语句匹配时，它会创建一个 `ast::Statement::Branch` 节点，其中包含了 `condition` 和 `then`，并将其封装在 `Box` 中。

`"loop" body:Block => { ... }`:这个分支表示一个循环语句，以 `"loop"` 关键字开始，后面是一个代码块 `body`。当这种类型的语句匹配时，它会创建一个 `ast::Statement::Loop` 节点，其中包含了 `body`，并将其封装在 `Box` 中。

"speak" ";" => { ... }:这个分支表示一个输出语句，以 "speak" 关键字开始，后面是一个表达式 Expression，以及分号 ;。当这种类型的语句匹配时，它会创建一个 ast::Statement::Speak 节点，其中包含了 Expression，并将其封装在 Box 中。

"input" <input:"identifier"> ";" => { ... }:这个分支表示一个输入语句，以 "input" 关键字开始，后面是一个标识符 input，以及分号 ;。当这种类型的语句匹配时，它会创建一个 ast::Statement::Input 节点，其中包含了 input，并将其封装在 Box 中。

"exit" ";" => { ... }:这个分支表示一个退出语句，以 "exit" 关键字开始，后面是一个分号 ;。当这种类型的语句匹配时，它会创建一个 ast::Statement::Exit 节点，不包含任何附加信息，并将其封装在 Box 中。

Block:这个分支表示一个语句块。它没有特定的开始关键字，而是由花括号 { 和 } 包围一系列语句（<stmts:Statement*>）来定义。当这种类型的语句块匹配时，它会创建一个 ast::Statement::Block 节点，其中包含了 stmts 中的多个语句，将它们封装在 Box 中。

";" => { ... }:这个分支表示一个表达式语句，它由一个表达式 Expression 后跟分号 ; 构成。当这种类型的语句匹配时，它会创建一个 ast::Statement::Expression 节点，其中包含了 Expression，并将其封装在 Box 中。

其余部分详见grammar.lalrpop

interpreter实现

首先需要实现解释器环境管理，实现较为简单：

本解释器唯一需要注意的是variable和value之间的一一对应，因此可以环境内部便是一个map，key=variable，value=value。

环境之间管理采用stack结构，当进入一个block时可以将父环境拷贝，然后处理新增变量，执行完该block后可以直接pop掉，并对环境进行更新

```
/* 添加新环境 */
fn add_new_env(&mut self) {
    let new_env = self.env.last().cloned().unwrap(); // 克隆上一个环境
    self.env.push(new_env);
}
/* 从栈顶弹出当前环境 */
fn rm_now_env(&mut self) {
    self.update_env();
    self.env.pop();
}
/* 添加变量到当前环境中 */
fn add_new_var(&mut self, name: String, init: Box<ast::Expr>) {
    if let Some(cur_env) = self.env.last_mut() {
        cur_env.insert(name, init.trans(cur_env.clone()));
    }
}
```

最重要的是update_env方法，用于比较解释器环境栈中的最后两个环境，并在倒数第二个环境中更新那些在最后一个环境中有变化的键值对。这有助于确保环境栈中的环境在嵌套时能够正确地维护变量和值之间的关联关系。

```rust
fn update_env(&mut self) {
        if self.env.len() < 2 {
            return; // 如果环境中的 HashMap 小于 2 个，无法进行比较
        }

        let last_idx = self.env.len() - 1;
        let second_last_idx = last_idx - 1;

        let last_env = self.env.get(last_idx).cloned();
        let second_last_env = self.env.get_mut(second_last_idx);

        if let (Some(last_env), Some(second_last_env)) = (last_env, second_last_env) {
            // 创建一个克隆的 HashMap，用于存储需要更新的键值对
            let updates: HashMap<String, String> = last_env
                .iter()
                .filter(|(key, value)| {
                    // 仅保留与倒数第二个 HashMap 不同的项
                    second_last_env.get(*key) != Some(value)
                })
                .map(|(key, value)| (key.clone(), value.clone()))
                .collect();

            // 更新倒数第二个 HashMap
            second_last_env.extend(updates);
        }
    }
```

能够正确管理环境后便是递归处理AST，详见interpreter.rs