# 程序设计实践

# DSL 解释器 – 程序设计文档

2021211110 杜抒泽

2023 年 12 月 20 日

## 目录

**前言 – 项目概述** 3
**1 DSZL 语法说明** 4
**1.1 类型系统与运算** 4
**1.2 词法介绍** 5
**1.3 语法介绍** 7
1.3.1 程序主体 8
1.3.2 声明语句 9
1.3.3 退出、输出、等待、表达式语句 9
1.3.4 分支、循环语句 9
1.3.5 输入语句 10
1.3.6 表达式 10
**1.4 代码示例** 11
**2 DSZL 设计思路与程序间接口** 14
**2.1 技术选型简介** 14
**2.2 设计思路与模块结构** 14
**2.3 后端内接口：DSZL 语言解释器** 15
**2.4 前后端间接口** 18
2.4.1 数据结构设计 18
2.4.1.1 抽象语法树 AST 18
2.4.1.2 另一棵树：运行环境 22
**3 客服机器人前端：人机接口** 24
**3.1 前端概述** 24
**3.2 启动！** 24
**3.3 终端用户界面 TUI** 24
**3.4 图形用户界面 GUI** 25
**4 测试** 28

**4.1 单元测试** ..... 28
4.1.1 测试 `crate::lang::parser::test::complicated` ..... 29
4.1.2 Mock 测试桩 `DummyInteractor` 与复杂测试 ..... 31
**4.2 集成功能测试脚本** ..... 34
**附录 A DSZL 配套设施** 44
**附录 A.1 代码配套提供的超多文档！** ..... 44
**附录 A.2 语法高亮支持：`dszl.sublime-syntax`** ..... 46

# 前言 – 项目概述

本项目实现了一个 DSL 解释器，名为 DSZL（全称 Domain Specific Zestful Language）[1]，DSZL 同时也是这个语言的名字。项目的全部内容基于 Rust 1.74.1 实现。

解释器包含手动实现的后端，附带了两种前端。

DSZL 采用命令式语法，实现了基本的算术运算，以及分支、循环的控制流，是图灵完备的语言；提供了输入、输出、睡眠的原语，输入可以方便地进行定时，并设定超时时进行的动作。由于这些特点，DSZL 适合用作客服机器人等交互友好程序。

此外，DSZL 提供了中英文两套关键字，这使得想要使用中文进行编程的人也能够轻松上手[2]。

本文档的后续部分将详细介绍 DSZL 语言和解释器的设计与实现，开发测试过程，用户界面使用指南等内容。

[1] 起这个名字的另一个原因是我的名字的 acronym 是 DSZ。

[2] 忍俊不禁

# 1 DSZL 语法说明

将 DSZL 采用的文法定义为 $G = (N, T, P, S)$。其中 $N$，$T$，$P$，$S$ 分别为非终结符、终结符、生成式的集合以及开始符号。本章将逐个对它们进行说明。

## 1.1 类型系统与运算

在了解 DSZL 的语法之前，首先需要了解 DSZL 的类型系统以及它们所能进行的运算。DSZL 的类型系统包含 5 种类型：

- `Boolean`：布尔值。
- `Callable`：可调用对象。相当于普遍意义上的过程。
- `Nil`：空类型。
- `Number`：符合 IEEE 754 标准的 64 位浮点数。
- `Str`：字符串，能够存储任意长度的 Unicode 字符序列。

它们之间定义的全部运算如下。每当运算符上的操作数类型不符合下列定义时，DSZL 解释器报告发生了类型错误与所在的行号，然后立即错误退出（如同 Python 解释器在遇到相似情况时抛出异常后的行为，但不会输出调用栈）。

- 布尔转换[1]：
  - 对于 `Boolean`，返回自身。
  - 对于 `Callable`，返回 `true`。
  - 对于 `Nil`，返回 `false`。
  - 对于 `Number`，当且仅当其值不为 0 时返回 `true`，否则返回 `false`。
  - 对于 `Str`，当且仅当其长度不为 0 时返回 `true`，否则返回 `false`。
- （一元）逻辑非 `!`：返回布尔转换的逻辑非。
- 逻辑与 `&&`：两操作数都是 `Boolean` 时，进行逻辑与运算，返回 `Boolean`。
- 逻辑或 `||`：两操作数都是 `Boolean` 时，进行逻辑或运算，返回 `Boolean`。
- （一元）单位 `+`：对于任何类型的操作数都有效，返回操作数本身。
- 加法 `+`
  - 两操作数都是 `Number` 时，进行 IEEE 754 浮点数加法，返回 `Number`。
  - 两操作中至少一个是 `Str`，另一个是 `Number` 时，返回 `Str`。
- （一元）取反 `-`：只对 `Number` 有效，返回操作数的 IEEE 754 相反数。

- 减法 `-`：两操作数都是 `Number` 时，进行 IEEE 754 浮点数减法，返回 `Number`。
- 模糊匹配 `~=`：两操作数都是 `Str` 时，进行模糊匹配，返回 `Boolean`。模糊匹配返回 `true`，当且仅当两个操作数之一是另一个的子串，或右操作数能被解析为正则表达式，且能匹配左操作数。
- 等于 $=$
  - 两操作数都是 `Number` 时，进行 IEEE 754 浮点数等于比较，返回 `Boolean`。
  - 两操作数都是 `Str` 时，进行字符串等于比较，返回 `Boolean`。
  - 两操作数都是 `Boolean` 时，进行布尔值等于比较，返回 `Boolean`。
  - 两操作数都是 `Nil` 时，返回 `Boolean`，且总为 `true`。
- 小于 `<`
  - 两操作数都是 `Number` 时，进行 IEEE 754 浮点数小于比较，返回 `Boolean`。
  - 两操作数都是 `Str` 时，进行字符串小于比较，返回 `Boolean`。
- 不等于 $\neq$，大于 `>`，小于等于 $\leqslant$，大于等于 $\geqslant$ 全部采用 $=$、`<` 和 `!` 的组合实现。例如，`lhs` $\leqslant$ `rhs` 等价于 `!(rhs < lhs)`。

## 1.2 词法介绍

DSZL 中的词法元素（token）对应文法中的终结符 $T$，以大驼峰命名，如下表。词法分析按照类似于 C/C++ 的最大吞噬原则，贪心地匹配可能最长的 token，而不一定按照唯一正确的方式分词。字符串支持转义序列 `[[LF]]` 和 `[[TAB]]`。

下面的两个表格很长，建议直接点击链接跳转到表格后。

| 类别 | 名称 | 描述 | 实体或描述 |
|---|---|---|---|
| 记号 (Punctuation) | `<semicolon>` | 分号 | `;` |
| | `<LeftParen>` | 左圆括号 | `(` |
| | `<RightParen>` | 右圆括号 | `)` |
| | `<LeftBrace>` | 左花括号 | `{` |
| | `<RightBrace>` | 右花括号 | `}` |
| | `<Assign>` | 等号 | `=` |
| | `<DoubleQuote>` | 双引号 | `"` |

[1]这个术语的名字是从 C/C++ 借来的。

| | | | |
|---|---|---|---|
| 运算符 | <Plus> | 加 | + |
| | <Minus> | 减 | - |
| | <LAnd> | 逻辑与 | && |
| | <LOr> | 逻辑或 | \|\| |
| | <Match> | 匹配 | ~= |
| | <Equal> | 等于 | == |
| | <NotEqual> | 不等于 | ≠ |
| | <Less> | 小于 | < |
| | <LessEqual> | 小于等于 | ≤ |
| | <Greater> | 大于 | > |
| | <GreaterEqual> | 大于等于 | ≥ |
| | <Bang> | 逻辑非 | ! |
| 关键字 | <Branch> | 分支语句 | branch 或“分支” |
| | <Else> | 分支语句 | else 或“否则” |
| | <While> | 循环语句 | while 或“循环” |
| | <State> | 状态声明 | state 或“状态” |
| | <Let> | 变量声明 | let 或“定义” |
| | <Input> | 输入语句 | input 或“输入” |
| | <InputNum> | 输入数字语句 | input_num 或“输入数字” |
| | <Timeout> | 两种输入语句指定超时 | timeout 或“超时” |
| | <Output> | 输出语句 | output 或“输出” |
| | <Exit> | 退出语句 | exit 或“退出” |
| | <Sleep> | 等待语句 | sleep 或“等待” |
| 标识符 | <Identifier> | 标识符 | 所有由字母、数字、Unicode 中的非 ascii 字符连接成的字符串，首字符不为数字，且不与关键字冲突者。 |

| | | | |
|---|---|---|---|
| 字面量 | <Number> | 数字 | 能被正则表达式 /[.\d][eE.\d]*/ 匹配者。 |
| | <String> | 字符串 | 能被正则表达式 /"[^"]*"/ 匹配者。 |
| | <True> | 布尔值真 | true 或“真” |
| | <False> | 布尔值假 | false 或“假” |
| | <Nil> | 空值 | nil 或“空” |

## 1.3 语法介绍

DSZL 中的语法元素集合对应文法中的非终结符 $N$，如下表。其中，$S =$ <program> 是文法的开始符号。

| 类别 | 名称 | 描述 |
|---|---|---|
| 程序主体 | <program> | 程序 |
| 语句 | <statement> | 语句 |
| | <block> | 代码块 |
| | <exit-statement> | 退出语句 |
| | <if-statement> | 条件语句 |
| | <while-statement> | 循环语句 |
| | <input-statement> | 输入语句 |
| | <output-statement> | 输出语句 |
| | <sleep-statement> | 休眠语句 |
| | <expression-statement> | 纯表达式语句 |
| | <state-declaration> | 状态声明 |
| | <variable-declaration> | 变量声明 |

| | | |
|---|---|---|
| 表达式 | <expression> | 表达式 |
| | <comparison> | 比较表达式 |
| | <logic> | 逻辑表达式 |
| | <arithmetic> | 算术表达式 |
| | <unary> | 一元表达式 |
| | <call> | 调用 / 状态切换 |
| | <arguments> | 状态实参列表 |
| 原子 | <atom> | 原子 |
| | <identifier> | 标识符 |
| | <number> | 数字 |
| | <string> | 字符串 |
| | <boolean> | 布尔值 |
| | <nil> | 空值 |

点击回到表格前

通过下文即将介绍的产生式容易发现，该文法中的任意生成式不含有歧义和左递归，我们直接使用递归下降分析法进行语法分析。

### 1.3.1 程序主体

程序的主体结构是按顺序编写的一系列声明或语句。解释器会按照顺序依次遍历这些声明或语句：对于一个声明，解释器会将其添加到当前状态中；对于一个语句，解释器会执行它。这和 Python 中当 `__name__` 为 `"__main__"` 时的情形相同。

程序主体的 BNF 范式描述如下。

```
<program> ::= <statement>*

<block> ::= "{" <program> "}"

<statement> ::= <block>
              | <exit-statement>
              | <branch-statement>
              | <while-statement>
              | <input-statement>
              | <output-statement>
```

```
                | <sleep-statement>
                | <expression-statement>
                | <state-declaration>
                | <variable-declaration>
```

特别地，对于一个语句块，解释器会在一个新的作用域中执行其中的语句。这个作用域会继承当前作用域的环境。若子环境中声明了一个与父环境中同名的变量，则会覆盖父环境中的同名变量。当语句块执行完毕后，解释器会将作用域恢复到执行语句块前的状态。若子环境访问一个未声明的变量，则会向上递归地在父环境中查找，直到找到该变量或到达全局作用域。若在全局作用域中仍未找到该变量，则解释器报告发生了错误，并立即错误退出（如同 Python 解释器在遇到相似情况时抛出异常后的行为，但不会输出调用栈）。

### 1.3.2 声明语句

声明语句包括状态声明和变量声明。二者会将声明的内容添加到当前状态的环境中。正如上文所述，状态的环境会在状态切换时被保存，而变量的环境则不会。它们的 BNF 范式描述如下。

```
<state-declaration> ::= "state" <identifier> "=" <block>

<variable-declaration> ::= "let" <identifier> "=" <expression> ";"
```

### 1.3.3 退出、输出、等待、表达式语句

这些语句都比较简单。退出语句会立即终止程序的执行，输出语句会将表达式的值输出到标准输出流，等待语句会让程序休眠一段时间，表达式语句会计算表达式的值并丢弃（或返回）。

```
<exit-statement> ::= "exit;"

<output-statement> ::= "output" <expression> ";"

<sleep-statement> ::= "sleep" <expression> ";"

<expression-statement> ::= <expression> ";"
```

### 1.3.4 分支、循环语句

分支语句是编程语言中最常见的语句之一。它的 BNF 范式描述如下。

```
<branch-statement> ::= "branch" "(" <expression> ")" <statement> ("else"
<statement>)?
```

其中的 else 子句是可选的。expression 的值会被执行前文提到的布尔转换，若结果为 true，则执行第一个语句，否则执行第二个语句（若存在）。注意到，两个分支都为语句，而不是语句块。这样可以简单地实现 else branch 的功能。

循环语句是另一个控制流语句。它的 BNF 范式描述如下。

```
<while-statement> ::= "while" "(" <expression> ")" <statement>
```

expression 的值会被执行前文提到的布尔转换，若结果为 true，则执行语句，然后再次执行 expression，直到 expression 的值为 false 为止。

分支语句和循环语句是让 DSZL 实现图灵完备的关键。通过它们，我们可以实现任意的控制流结构。

### 1.3.5 输入语句

输入语句是 DSZL 中最特殊的语句。通过它，DSZL 得以适合于交互式的场景。它的 BNF 范式描述如下。

```
<input-statement> ::= "input" <identifier> ("timeout(" <expression> ")"
<statement>)? ";"

<input-num-statement> ::= "input_num" <identifier> ("timeout(" <expression> ")"
<statement>)? ";"
```

若 <identifier> 未在当前状态的环境中声明，则解释器会将其添加到环境中。对于 input 语句，解释器会从标准输入流中读取一行字符串并赋值给 <identifier>。对于 input_num 语句，解释器会从标准输入流中读取一行字符串，然后将其转换为数字，并赋值给 <identifier>。若用户输入的字符串不能被转换为数字，则解释器报告发生了错误，并重新等待用户输入。

expression 的值必须为数字，单位为秒。若用户提供了这个值，则启用输入时限。若用户在规定时间内输入了一行字符串，则将其转换为数字并赋值给 <identifier>，然后执行第一个语句。若用户在规定时间内未输入，则执行第二个语句。

### 1.3.6 表达式

表达式是 DSZL 中最重要的语法元素。它的 BNF 范式描述如下。

```
<expression> ::= <identifier> "=" <expression>
               | <comparison>

<comparison> ::= <logic> ("~=" | "==" | "≠" | "<" | "≤" | ">" | "≥") <logic>

<logic> ::= <arithmetic> (("&&" | "||") <arithmetic>)*

<arithmetic> ::= <unary> (("+" | "-") <unary>)*

<unary> ::= ("+" | "-" | "!") <unary>
          | <call>

<call> ::= <atom> ("(" <arguments> ")")?

<arguments> ::= <expression>? (" " <expression>)*

<atom> ::= <identifier>
         | <number>
         | <string>
         | <boolean>
         | <nil>
         | "(" <expression> ")"
```

表达式的优先级从高到低依次为 atom（原子单位），call（调用），unary（一元表达式），arithmetic（算术表达式），logic（逻辑表达式），comparison（比较表达式），expression（赋值表达式）。我们通过这种文法，实现了运算符的优先级。

## 1.4 代码示例

以下是三段简单的 general purpose DSZL 代码示例。其中第一段代码展示了 DSZL 对嵌套作用域的处理，第二段代码展示了 DSZL 对正则表达式的支持，第三段代码展示了 DSZL 对中文关键字和简单数学计算的支持。

```
// ——— 嵌套测试 ———

state nested2() {
  output "nested2 ?";
}

state nested() {
    let captured = 114;
```

```
    state nested2() {
        output "nested2 !";
        output "captured = " + captured; // 输出 "captured = 114"
    }
    nested2(); // 输出 "nested2 !"
}

nested();
nested2(); // 输出 "nested2 ?"
// output "captured = " + captured; // 报错: 变量 captured 未定义

output "以上是嵌套测试。";
```

```
// ——— 匹配测试 ———
let re = "1+4\d*";
output "正则: " + re;
let a = "Default String";
output "请在 5 秒内输入字符串 a: ";
input a timeout(5) {
  output "[[LF]]超时! ";
};
branch (a ~= re) {
  output "匹配成功。";
} else {
  output "匹配失败。";
}
```

```
// ——— 中文关键字求斐波那契 ———
定义 斐波那契 = 1;
输出 "请输入要求斐波那契数列的项数: ";
输入数字 项数;
定义 当前下标 = 0;
定义 当前项 = 1;
定义 上一项 = 0;
循环 (当前下标 < 项数) {
  定义 临时 = 当前项;
  当前项 = 当前项 + 上一项;
  上一项 = 临时;
  当前下标 = 当前下标 + 1;
```

```
  输出 "第 " + 当前下标 + " 项是 " + 当前项 + "。";
  等待 0.3;
}
```

# 2 DSZL 设计思路与程序间接口

## 2.1 技术选型简介

DSZL 语言的解释器实现使用 Rust 语言。Rust 具有以下优点：

- 内存安全、并发安全、线程安全，同时具有较高的性能。同时编译器具有较高的错误检查能力，可以在编译期间检查出大部分错误。
- 抽象能力极强，提供了诸多现代语言的特性，如模块化、泛型、闭包、模式匹配等。
- 提供完备的工具链，包括包管理、编译、从注释生成文档、**自动单元测试**等功能。
- 是系统级语言，可以直接调用 C 语言库（含系统调用库）。

在本项目中，我使用到了上述的以下特性：

- 安全的并发支持。代码里的 `Arc<Mutex<T>>` 数量多到我三天睡不着觉。
- 高度模块化，将项目分为多个模块，模块之间通过 `pub(crate)` 和 `trait` 进行通信。
- 超强的类型抽象能力，代码中遍布闭包、模式匹配等特性，使代码清晰易懂[1]。其结果是代码基本过了编译就能运行出正确的结果，逻辑错误几乎绝迹，调试代价大大降低。
- 方便的包管理，使用 Cargo 管理项目的依赖，使得项目的依赖管理更加简单。
- 自动注释文档生成和自动单元测试。
- 调用 `libc` 中的系统调用 `select`，实现读入定时的功能。

项目一共 17 个 Rust 代码文件，3155 sloc Rust 代码，另有 1150 行注释，其大多为文档注释。

## 2.2 设计思路与模块结构

```
$ tree --gitignore
.
├── Cargo.lock
├── Cargo.toml
├── assets
│   └── ...
├── config
│   └── log4rs.yaml
...
└── src
    ├── api
    │   ├── cli.rs
    │   └── interact.rs
```

[1]可能没写过 Rust 朋友观感会很乱，但是真的很好懂。

```
    ├── api.rs
    ├── lang
    │   ├── environment.rs
    │   ├── expr.rs
    │   ├── lexer.rs
    │   ├── parser.rs
    │   ├── result.rs
    │   ├── statement.rs
    │   ├── token.rs
    │   ├── variable.rs
    │   └── visitor.rs
    ├── lang.rs
    ├── main.rs
    ├── ui
    │   ├── gui.rs
    │   └── tui.rs
    └── ui.rs
```

除素材文件 assets，和配置文件 config 外，项目文件结构分为以下模块：

- Cargo.lock 和 Cargo.toml，分别是项目的依赖锁定文件和依赖管理文件，其中包含了项目的依赖信息，以及项目的元信息。
- src，项目源代码目录，包含了项目的所有源代码。
  - api，接口模块，包含了项目的命令行用户接口以及前后端程序间接口实现。
  - lang，项目的语言模块，包含了 DSZL 语言本体和解释器的实现。
  - main.rs，项目的主模块，包含程序的入口。
  - ui，项目的用户界面模块，包含了项目的用户界面实现。

api 模块中的内容之后单独介绍。ui 模块中的两个文件相互独立，是程序的两个前端（用户界面）实现，其中 gui.rs 是图形用户界面的实现，tui.rs 是文本用户界面的实现。

以下重点介绍 lang 模块中的内容。

## 2.3 后端内接口：DSZL 语言解释器

lang 又分为以下模块：

- environment.rs：环境模块，包含了 DSZL 语言的运行环境实现，如变量表、函数表。
- expr.rs：表达式模块，包含了 DSZL 语言的表达式实现。
- lexer.rs：词法分析模块，包含了 DSZL 语言的词法分析器实现。
- parser.rs：语法分析模块，包含了 DSZL 语言的语法分析器实现。

- `result.rs`：`Result<T, E>` 是 Rust 实现用类型进行错误处理与传播的能力的工具。我在这个模块里定义了 DSZL 语言 Lexer、Parser、Visitor 的的错误类型。
- `statement.rs`：语句模块，包含了 DSZL 语言的语句实现。
- `token.rs`：词法元素模块，包含了 DSZL 语言的词法元素实现。
- `variable.rs`：变量模块，包含了 DSZL 语言的变量实现。
- `visitor.rs`：遍历者（解释器）模块，包含了 DSZL 语言的遍历者实现。它能在 AST 上进行遍历，执行程序。

它们之间的关系如图所示：

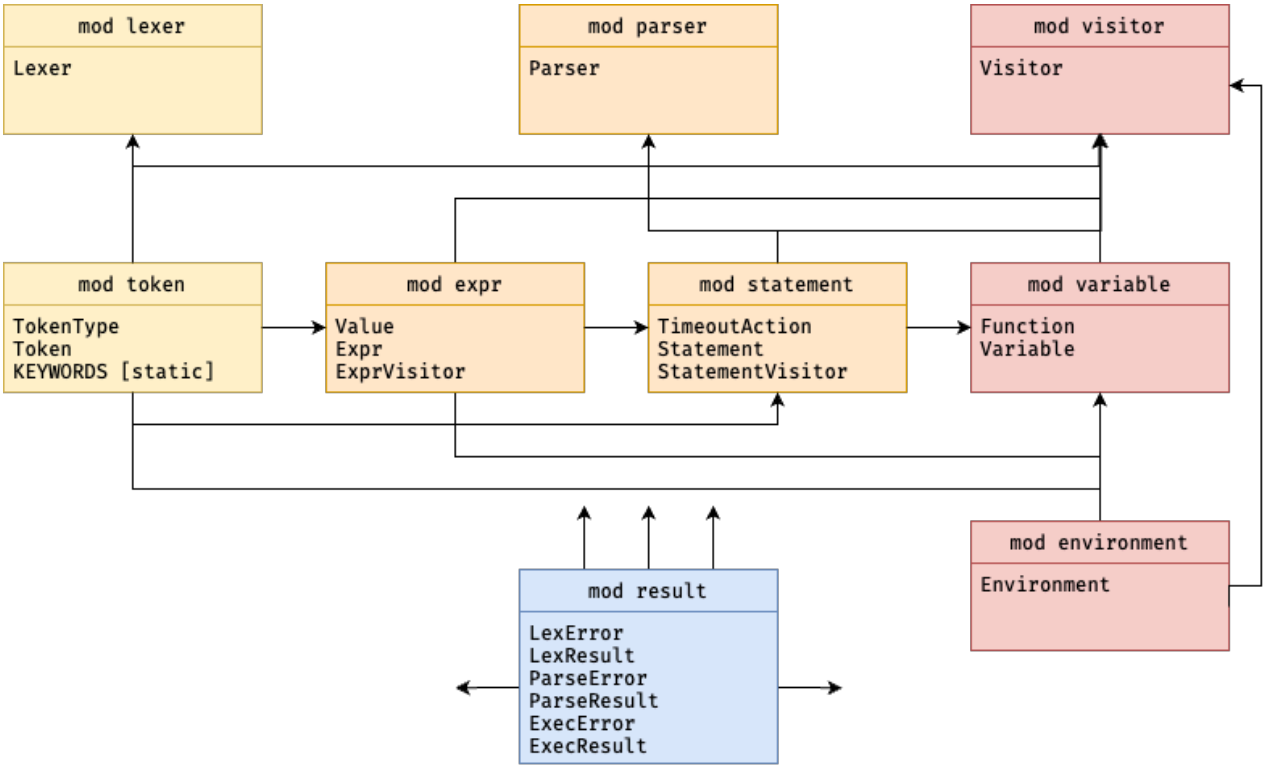


图 2.1　模块关系图

可以看到，总体依赖顺序为按照从左到右、从上到下的顺序抽象层级依次提高。

对于最上层的三个类型 `Lexer`，`Parser`，`Visitor`，它们暴露出的接口分别如下：

```rust
impl Lexer {
    pub(crate) fn with_source(src: &str) -> Self;

    pub(crate) fn lex_effective(&mut self) -> (Vec<Token>, Vec<Token>);
}

impl Parser {
    pub(crate) fn with_tokens(tokens: Vec<Token>) -> Self;
```

```
    pub(crate) fn parse_effective(&mut self) → (
        Vec<Statement>, Vec<ParseError>
    );
}

impl Visitor {
    pub(crate) fn with_interactor(interactor: Arc<dyn Interact>) → Self;

      pub(crate) fn executes(&mut self, statements: Vec<Statement>) →
ExecResult<()>
}
```

它们之间相似但各有不同。一般的具体调用方式如下：

```
let src: &str = /* ... 源代码 ... */;

let (tokens, errors) = Lexer::with_source(src).lex_effective();
// tokens 为词法元素，errors 为词法错误
if !errors.is_empty() {
  /* 有词法错误，处理错误 */
}

let (statements, errors) = Parser::with_tokens(tokens).parse_effective();
// statements 为语句，errors 为语法错误
if !errors.is_empty() {
  /* 有语法错误，处理错误 */
}

// trait Interact 由后端定义，interactor 由前端提供与实现
let interactor: Arc<dyn Interact> = /* ... 交互器 ... */;
let result = Visitor::with_interactor(interactor).executes(statements);
// result 为执行结果
match result {
  Ok(()) ⇒ { /* 执行成功 */ },
  Err(e) ⇒ { /* 执行失败，处理错误 */ },
}
```

以上是后端内部的接口，以及后端如何使用前端提供的接口。我们下面介绍前后端之间的接口。

## 2.4 前后端间接口

前后端间通过一个叫作 `Interact` 的特征进行通信。它由后端定义，前端实现。它的定义如下：

```rust
pub(crate) trait Interact {
    /// 阻塞，直到读入一行输入 s 返回 s。
    fn input(&self) -> String;

    /// 阻塞，直到读入一行输入 s 返回 Some(s)，或超时返回 None。
    fn input_timeout(&self, duration: std::time::Duration) -> Option<String>;

    /// 前端输出 s。
    fn output(&self, s: &str);

    /// 前端提供输出的历史记录。这个方法只会被测试用例调用，因此提供默认实现直接 panic。
    fn output_history(&self) -> String {
        unimplemented!();
    }
}
```

前端需要实现一个 `impl` 了 `trait Interact` 的类型，将其封装到一个 `Arc<dyn Interact>` 中，传递给后端。后端通过这个接口与前端进行交互。

需要注意的是，这个 `dyn Interact` 不会被可变借用！这确保了它的线程安全性，但这给前端的实现带来了一定的困扰。一般的解决方案有两个：

1. 前端根本不需要可变借用。例如直接向终端输出的前端就可以直接采用这个方案。
2. 前端实现一个允许可变借用的 `InteractorHelper` 类，由 `dyn Interact` 存储 `Arc<Mutex<InteractorHelper>>`，然后调用它的方法。这个方案被实现 GUI 和测试桩时实现。

总之这个接口是十分精妙的，它用很少的代码将前后端的交互抽象为了一个接口，使得前后端可以独立开发，互不影响。

### 2.4.1 数据结构设计

数据结构设计这个我很在行啊。以前是打竞赛的。

#### 2.4.1.1 抽象语法树 AST

我的程序会将 DSZL 脚本先序列化成词法元素，然后进行语法分析，将其按顺序分为若干条语句（或声明），每条语句都是一棵抽象语法树（AST），可以认为整个程序是一棵巨大的 AST。

具体地，有三个类型和脚本的 AST 直接相关，分别是语句 `Statement`，计算表达式 `Expr` 和脚本变量 `Variable`。看了前两的定义，自然就知道了这为什么是一棵树[2]：

```
pub(crate) enum Statement {
    Block(Vec<Statement>),
    Exit,
    Expression(Expr),
    State { ident: Token, params: Vec<Token>, body: Vec<Statement> },
    If {
        condition: Expr,
        then_branch: Box<Statement>,
        else_branch: Option<Box<Statement>>,
    },
    Input { ident: Token, timeout_action: Option<TimeoutAction> },
    InputNum { ident: Token, timeout_action: Option<TimeoutAction> },
    Output(Expr),
    Sleep(Expr),
    Let { ident: Token, initialiser: Option<Expr> },
    While { condition: Expr, body: Box<Statement> },
}
```

```
pub(crate) enum Expr {
    Assign { lhs: Token, rhs: Box<Expr> },
    Binary { lhs: Box<Expr>, op: Token, rhs: Box<Expr> },
    Call { func: Box<Expr>, args: Vec<Expr> },
    Literal(Value),
    Unary { op: Token, rhs: Box<Expr> },
    Variable(Token),
}
```

可以看到，语句 `Statement` 和计算表达式 `Expr` 都是枚举类型，它们的每个分支都是一个节点。`Statement` 会利用指针嵌套其他 `Statement` 和 `Expr`，使之成为子节点；`Expr` 也会嵌套其他 `Expr` 成为子节点。这样就形成了一棵树。

[2]这个语法高亮不好读，全是黑的

我的脚本在运行时会向日志文件中写入语法树的全貌。例如，我们可以看到下面这个脚本的语法树：

```
let name = "";
let age = 0;
input name;
input_num age;
output name + " is " + age + " years old.";
```

运行程序，得到语法树如下：

```
Statements: [
    Let {
        ident: Token {
            typ: Identifier,
            lexeme: "name",
            line_num: 1,
        },
        initialiser: Some(
            Literal(
                Str(
                    "",
                ),
            ),
        ),
    },
    Let {
        ident: Token {
            typ: Identifier,
            lexeme: "age",
            line_num: 2,
        },
        initialiser: Some(
            Literal(
                Number(
                    0.0,
                ),
            ),
        ),
    },
    Input {
        ident: Token {
```

```
            typ: Identifier,
            lexeme: "name",
            line_num: 3,
        },
        timeout_action: None,
    },
    InputNum {
        ident: Token {
            typ: Identifier,
            lexeme: "age",
            line_num: 4,
        },
        timeout_action: None,
    },
    Output(
        Binary {
            lhs: Binary {
                lhs: Binary {
                    lhs: Variable(
                        Token {
                            typ: Identifier,
                            lexeme: "name",
                            line_num: 5,
                        },
                    ),
                    op: Token {
                        typ: Plus,
                        lexeme: "+",
                        line_num: 5,
                    },
                    rhs: Literal(
                        Str(
                            " is ",
                        ),
                    ),
                },
                op: Token {
                    typ: Plus,
                    lexeme: "+",
                    line_num: 5,
                },
```

```
                rhs: Variable(
                    Token {
                        typ: Identifier,
                        lexeme: "age",
                        line_num: 5,
                    },
                ),
            },
            op: Token {
                typ: Plus,
                lexeme: "+",
                line_num: 5,
            },
            rhs: Literal(
                Str(
                    " years old.",
                ),
            ),
        },
    ),
]
```

十分庞大。这个语法树是我们程序的核心，我们的整个解释器都是基于这个语法树进行遍历的。

#### 2.4.1.2 另一棵树：运行环境

实际上，不止 AST，我们的程序还有另一棵树。这棵树是运行环境 `Environment`，它的内部包含一个存储了当前环境所有变量的值的哈希表和一个指向父环境的指针。

实际上说它是树有点过誉了，这棵树只在逻辑上是一棵树，但实际运行时在任意时刻只会存储其中的一条链（因为解释器是单线程执行的）。不过这不影响我们把它当作一棵树来看待。它的定义如下：

```rust
pub(crate) struct Environment {
    pub(crate) parent: Option<Rc<RefCell<Environment>>>,
    pub(crate) variables: HashMap<String, Variable>,
}
```

可以看到，类型定义似乎很复杂。这是 Rust 的安全性和高度抽象性共同导致的。`Option` 表示一个值可能存在（`Some <T>` 变体）或为空（`None` 变体）；`Rc` 表示一个引用计数指针，它

可以在多个地方引用同一个对象，当所有引用都被销毁时，对象才会被销毁；`RefCell` 表示一个可变借用，它用于维护指向的可变性。在现在的这个场景（儿子反向指向父亲）下，Rust 编译器无法确定解引用的安全性，因此需要用 `RefCell` 来将编译期检查转移到运行时，利用逻辑确保解引用的安全性，从而通过编译。

`Environment` 由解释器 `Visitor` 通过以下方式创建：

- 当 `Environment` 被创建时，它的 `parent` 会被设置为 `None`，表示它是一个根环境；它的 `variables` 会被初始化为空的哈希表。
- 在根环境上向下创建子环境时，子环境的 `parent` 会被设置为 `Some(Rc::new(RefCell::new(&self.env)))`，表示它的父环境是当前环境；它的 `variables` 会被初始化为空的哈希表。

`Visitor` 用 `Environment` 维护变量的方式是：

- 当 `Visitor` 需要创建变量时，它会向当前环境的 `variables` 中插入一个新的变量。
- 当 `Visitor` 需要读取或修改变量时，它会向当前环境的 `variables` 中查找变量，如果找不到，就迭代地向向父环境查找，直到找到或到达根环境。如果找到了，就读取或修改变量的值；如果到达根环境仍然找不到，报错并退出（如同 Python 解释器在类似情况下抛出异常并退出）。

这样，我们就可以在运行时维护变量的作用域了。

项目中仍然有其他利用数据结构知识的地方，这里只介绍了其中之二，其余不再赘述。

# 3 客服机器人前端：人机接口

## 3.1 前端概述

本项目的前端需求，同时也是人机接口，是人和机器人的交互界面。由于 Rust 是静态语言，极其难以实现代码的动态加载界面，我选择让程序在启动时获取 DSZL 脚本，然后解析并执行，只在运行时动态加载机器人界面。

我共实现了三个前端，其中两个人机接口（分别是命令行界面和图形界面），一个是单元测试使用的伪前端（也许可以称作测试桩的一部分）。

本章介绍前两个接口的使用方法及细节，第三个接口的使用方法在[测试]章节一并介绍。

## 3.2 启动！

程序在启动时会读取命令行参数提供的信息，具体如下：

```
$ ./target/release/dszl --help
Usage: dszl [OPTIONS]

Options:
  -f, --file <PATH>  The script file to execute
  -r, --run <UI>     The user interface to use [default: gui] [possible values: tui, gui]
  -v, --version      Print the version information and exit
  -h, --help         Print help

# fa_555 @ fa555-kooBcaM in ~/Documents/repos/dszl on git:master x [0:09:02]
$ ./target/release/dszl
Error: `--file <PATH>` is required.
Or input the path relative to the executable:
```

图 3.1　命令行参数

有以下几个参数：

- `-f, --file <PATH>`：指定 DSZL 脚本文件路径。

  **考虑到双击运行程序的运行方式对用户更加友好，这个参数是可选的。如果不指定这个参数，会在运行时提示用户输入脚本的相对路径，如同图中第二次运行展示的那样。**

  如果对程序稍作修改，Windows 用户将可以直接把脚本文件拖到程序上运行。但是我没有 Windows 设备，所以没有实现这个特性。
- `-r --run <UI>`：指定要启动的前端类型，可选值为 `tui` 和 `gui`。默认值为 `gui`。
- `-h, --help`：显示帮助信息，如同图中第一次运行展示的那样。
- `-v, --version`：显示版本信息。

## 3.3 终端用户界面 TUI

只需要在命令行参数中指定 `-r tui` 或 `--run tui`，就可以启动 TUI 前端。

```
$ cargo run --release -- -f examples/test2.dszl -r tui
    Finished release [optimized] target(s) in 0.22s
     Running `target/release/dszl -f examples/test2.dszl -r tui`
[00:17:08 INFO] Log loaded to `./instance/run.[#].log`.
[00:17:08 INFO] TUI start up with file `examples/test2.dszl`.
│ 欢迎使用本系统，请问您的用户名是？
├─ fa_555
│ fa_555，您好！
│ 请问有什么可以帮您？（投诉 / 账单）
├─ 我要投诉！！！！！！！！！
│ 您的意见是我们进步的动力，请问您有什么意见？
├─ 我不会写代码。
│ 收到意见“我不会写代码。”。我们会尽快处理您的投诉。
│ 感谢您的使用。
│
│ 请问有什么可以帮您？（投诉 / 账单）
├─ 账单
│ 您的本月账单为 114514 元。
│ 感谢您的使用。
```

图 3.2　TUI 运行示例

可以看到，我实现了一个简单的 TUI。它的底层使用了系统调用 `select` 实现了等待 IO 事件的功能。

在用户体验上，它将用户的输入和机器人的输出用不同的颜色高亮显示，以区分开来。用户可以直接在终端中输入文字与机器人交流，也可以自行修改终端设置，如字体种类，字体大小，颜色等。它的表现已经足够好。

## 3.4 图形用户界面 GUI

这个 GUI 就更有说头了。它使用了一个名为 `egui` 的 Rust GUI 框架，基于即时 GUI 的技术，在运行时加载界面而无需传统的事件循环。并且，这个 GUI 需要的中文字体占据了可执行文件体积的 80%，由它引起的代码量占据了总代码量的 20% 以上。使用如图：

图 3.3　GUI 运行示例：图 1

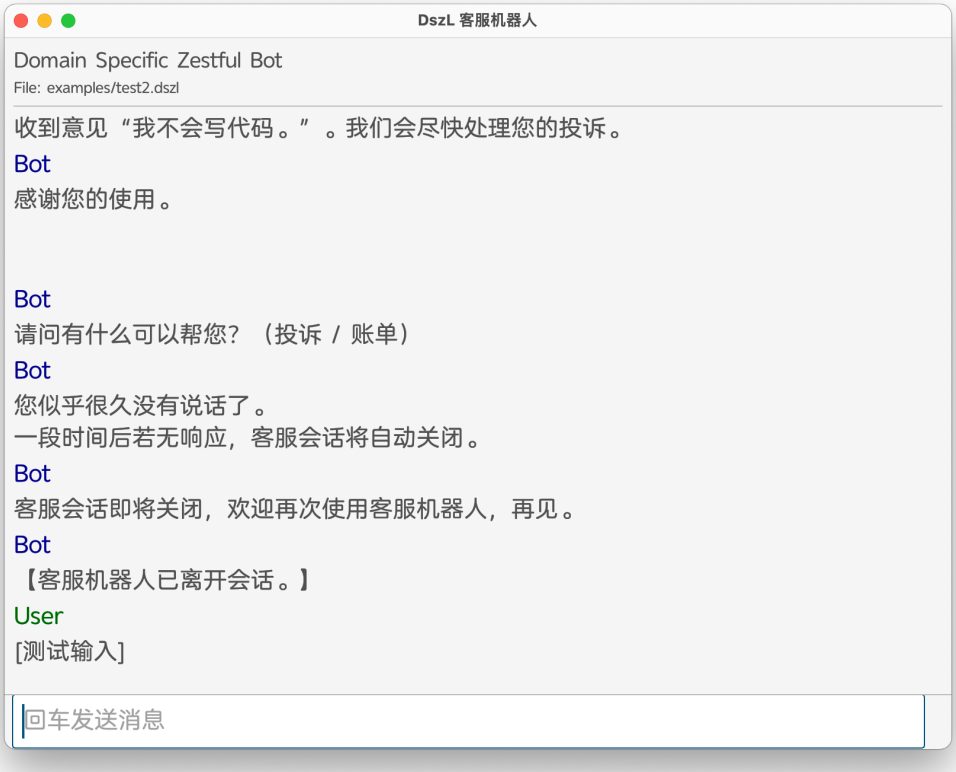


图 3.4　GUI 运行示例：图 2

在 GUI 中，用户和客服机器人的输入被名字分隔开。对于一条用户输入，如果被机器人捕获，那么它的下方会有下划线（如图 1 中的三条消息），否则没有（如图 2 中的消息）。这样，用户可以很清楚地看到自己的输入是否被机器人响应。

由于我实在不善编写和描述前端，因此还请自行运行程序体验。

# 4 测试

本项目的测试分为分模块单元测试和集成功能测试两部分。

## 4.1 单元测试

Rust 作为一门现代语言，其工具链提供了自动进行测试的能力。实践中一般采用的具体用法如下：

- 在每个模块内编写一个专门负责测试的子模块 `mod test` 并用过程宏 `#[cfg(test)]` 修饰。
- 在测试模块中编写测试用例。测试用例的所有函数都要用过程宏 `#[test]` 修饰，有些应当失败特殊的测试用例（例如给出错误的输入，需要在正确的地方崩溃指出错误）还需要用过程宏 `#[should_panic]` 修饰。
- 函数内部编写一系列测试桩用来模拟测试环境，然后编写测试脚本，使用 `assert!`，`assert_eq!` 等宏进行断言。
- 运行 `cargo test`（或 `cargo test [模块名]`）进行测试。
- 工具链会以十分友好的方式指出具体哪些测试成功、哪些测试失败、为什么失败、预期输出如何、实际输出如何。接着，我们根据这些信息进行调试。

我一共对 7 个模块编写了 46 个单元测试，全部通过，截图见下页。在截图之后，我会就某几个测试用例进行详细说明。

```
$ cargo t
    Finished test [unoptimized + debuginfo] target(s) in 0.24s
     Running unittests src/main.rs (target/debug/deps/dszl-7af27a111ef31c7f)

running 46 tests
test lang::expr::tests::expr_display ... ok
test lang::environment::test::state ... ok
test lang::lexer::test::creation ... ok
test lang::lexer::test::advance ... ok
test lang::lexer::test::efficient_chinese ... ok
test lang::lexer::test::lex_error ... ok
test lang::lexer::test::lex ... ok
test lang::lexer::test::complicated_script ... ok
test lang::lexer::test::next_token ... ok
test lang::lexer::test::peek ... ok
test lang::lexer::test::peek_offset ... ok
test lang::lexer::test::next_token_effective ... ok
test lang::lexer::test::reached_end ... ok
test lang::lexer::test::try_match ... ok
test lang::parser::test::block ... ok
test lang::parser::test::exit ... ok
test lang::parser::test::complicated ... ok
test lang::parser::test::complicated_wrong ... ok
test lang::parser::test::expr_statement ... ok
test lang::parser::test::expr ... ok
test lang::parser::test::if_ ... ok
test lang::parser::test::expr2 ... ok
test lang::parser::test::let_ ... ok
test lang::parser::test::loop_ ... ok
test lang::parser::test::input_output ... ok
test lang::parser::test::sleep ... ok
test lang::result::test::exec_error ... ok
test lang::parser::test::state ... ok
test lang::result::test::exec_result ... ok
test lang::result::test::lex_error ... ok
test lang::result::test::parse_error ... ok
test lang::result::test::parse_result ... ok
test lang::statement::test::statement_display ... ok
test lang::token::test::creation ... ok
test lang::token::test::display ... ok
test lang::token::test::keyword_map ... ok
test lang::visitor::test::function ... ok
test lang::visitor::test::input ... ok
test lang::visitor::test::input_timeout ... ok
test lang::visitor::test::branch ... ok
test lang::visitor::test::while_output ... ok
test lang::visitor::test::incorrect_parameter_amount - should panic ... ok
test lang::visitor::test::incompatible_operand_types - should panic ... ok
test lang::parser::test::parse_expr_wrong - should panic ... ok
test lang::visitor::test::variable_not_found - should panic ... ok
test lang::lexer::test::very_many_tokens ... ok

test result: ok. 46 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.05s
```

图 4.1 单元测试全貌

### 4.1.1 测试 crate::lang::parser::test::complicated

其代码如下：

```rust
#[test]
fn complicated() {
    #[rustfmt::skip]
    let src = r#"
```

```
状态 测试() {
    input_num x;
    输出 "测试状态 " + (x + 1);
}

while (true) {
    input str;
    branch (str == "in") {
        输出 "in";
    } else {
        输出 "out";
    }
}
"#;
    let (tokens, errors) = Lexer::with_source(src).lex_effective();
    assert!(errors.is_empty());

    let statements = Parser::with_tokens(tokens).parse();
    assert_eq!(statements.len(), 2);
    assert_eq!(
        format!("{}", statements[0].as_ref().unwrap()),
        "state [Ident: 测试]() { input_num [Ident: x]; output [Expr: [Binary +
[Literal \"测试状态 \"] [Binary + [Variable x] [Literal 1]]]]; }"
    );
    assert_eq!(
        format!("{}", statements[1].as_ref().unwrap()),
        "while ([Literal true]) { input [Ident: str]; if ([Binary == [Variable
str] [Literal \"in\"]]) { output [Expr: [Literal \"in\"]]; } else { output [Expr:
[Literal \"out\"]]; } }"
    );
}
```

可以看到，这个测试用例中包含测试桩 `src: &str`，和用受测试的模块 `Lexer` 和 `Parser` 以及 `assert!` 和 `assert_eq!` 宏写成的测试脚本。

首先，我们使用 `Lexer` 对源代码进行词法分析，得到词法单元流 `tokens` 和词法错误流 `errors`。我们断言词法错误流为空，即词法分析成功。

接着，我们使用 `Parser` 对词法单元流进行语法分析，得到语法树节点流 `statements`。我们断言语法树节点流的长度为 2，即语法分析得到了两个语法树节点。

然后，我们使用 `format!` 宏将语法树节点流中的两个语法树节点转换为字符串，断言它们的字符串形式与预期相同。测试最终成功了。

### 4.1.2 Mock 测试桩 `DummyInteractor` 与复杂测试

在人机接口中我曾提到，在测试 `crate::lang::visitor` 模块时，由于无法启动真正的前端测试，我使用了一个 Mock 测试桩 `DummyInteractor` 来模拟人机接口。它的代码如下（点此跳转到代码后）：

```
mod dummy {
    use crate::api::interact::Interact;
    use std::sync::{Arc, Mutex};

    pub(crate) struct DummyInteractorHelper {
        pub(crate) upcoming_input: Vec<Option<String>>,
        cur_pos: usize,
        pub(crate) output_stream: String,
    }

    impl DummyInteractorHelper {
        pub(crate) fn new() -> Self {
            Self {
                upcoming_input: Vec::new(),
                cur_pos: 0,
                output_stream: String::new(),
            }
        }

        pub(crate) fn load_input(&mut self, s: &[Option<&str>]) {
                            self.upcoming_input = s.iter().map(|s| s.map(|s|
s.to_string())).collect();
            self.cur_pos = 0;
        }

        pub(crate) fn consume(&mut self) -> Option<String> {
            let s = self.upcoming_input.get(self.cur_pos).cloned().flatten();
            self.cur_pos += 1;
            s
        }

        pub(crate) fn output_push(&mut self, s: &str) {
            self.output_stream.push_str(s);
```

```
        }

        pub(crate) fn output_stream(&self) -> &str {
            &self.output_stream
        }
    }

    pub(crate) struct DummyInteractor {
        helper: Arc<Mutex<DummyInteractorHelper>>,
    }

    impl DummyInteractor {
        pub(crate) fn new() -> Self {
            Self {
                helper: Arc::new(Mutex::new(DummyInteractorHelper::new())),
            }
        }

        pub(crate) fn load_input(&mut self, s: &[Option<&str>]) {
            let mut helper = self.helper.lock().unwrap();
            helper.load_input(s);
        }
    }

    impl Interact for DummyInteractor {
        fn input(&self) -> String {
            let mut helper = self.helper.lock().unwrap();
            loop {
                match helper.consume() {
                    Some(s) => return s,
                    None => {}
                }
            }
        }

      fn input_timeout(&self, _duration: std::time::Duration) -> Option<String>
{
            let mut helper = self.helper.lock().unwrap();
            helper.consume()
        }
```

```
        fn output(&self, s: &str) {
            let mut helper = self.helper.lock().unwrap();
            helper.output_push(s);
        }

        fn output_history(&self) -> String {
            let helper = self.helper.lock().unwrap();
            helper.output_stream().to_string()
        }
    }
}
```

正如在前文中所说，前端实现 `trait Interact` 的途径之一是实现一个允许可变借用的 `InteractorHelper` 类，由 `dyn Interact` 存储 `Arc<Mutex<InteractorHelper>>`，然后调用它的方法。这个方案在实现这个 mock / dummy 桩时被完美采用了。

在这个桩中，我利用 `Option` 的两个变体 `Some` 和 `None` 来模拟输入流。前者模拟正常输入，后者模拟输入超时。而输出会被记录到一个 `String` 中，可以通过实现的 `output_history()` 方法获取。（还记得前面讲接口的时候吗？这个方法只会在测试类中实现。）

总之，这个桩提供了一个完美的测试环境，使得我可以在没有前端的情况下测试 `crate::lang::visitor` 模块。

下面是测试用例 `crate::lang::visitor::test::input_timeout`，它测试了输入超时的情况：

```
#[test]
fn input_timeout() {
    #[rustfmt::skip]
    let src =
r#"
let a = "";
input a timeout(5) {
    a = "timeout";
};

let b = 0;
    input_num b timeout(5) {
    b = 114514;
};
```

```
output a + b;
"#;
    let (tokens, errors) = Lexer::with_source(src).lex_effective();
    assert!(errors.is_empty());

    let (statements, errors) = Parser::with_tokens(tokens).parse_effective();
    assert!(errors.is_empty());

    let mut interactor = DummyInteractor::new();
    interactor.load_input(&[None, Some("1919810")]);
    let mut visitor = Visitor::with_interactor(Arc::new(interactor));
    match visitor.executes(statements) {
        Ok(_) | Err(ExecError::Exit) => {}
        Err(e) => panic!("Execution failed with error: {}", e),
    }

    assert_eq!(visitor.interactor.output_history(), "timeout1919810");
}
```

可以看到，这个测试用例中，我首先使用 `Lexer` 和 `Parser` 对源代码进行词法分析和语法分析，得到词法单元流 `tokens`、词法错误流 `errors`、语法树节点流 `statements` 和语法错误流 `errors`。我断言词法错误流和语法错误流为空，即词法分析和语法分析成功。

我构造了一个 `DummyInteractor`，并使用它的 `load_input` 方法将输入流设置为 `&[Option<&str>]` 类型的 `[None, Some("1919810")]`。这代表第一次输入超时，第二次输入为字符串 `"1919810"`。阅读代码可知，在输入后，变量 `a` 的值应当为 `"timeout"`，变量 `b` 的值应当为 `1919810`。于是我们断言输出流为 `"timeout1919810"`。

测试成功。

## 4.2 集成功能测试脚本

以下是验收时展示的 5 个集成测试用例的脚本源代码。它们都是领域特定的客服机器人实现。此处不再过多赘述，读者可以自行阅读与运行程序。

`../examples/test1.dszl`

```
// ——— 嵌套测试 ———

状态 nested2() {
    输出 "nested2 ?";
}
```

```
状态 nested() {
    定义 captured = 114;
    状态 nested2() {
        输出 "nested2 !";
        输出 "captured = " + captured; // 输出 "captured = 114"
    }
    nested2(); // 输出 "nested2 !"
}

nested();
nested2(); // 输出 "nested2 ?"
// 输出 "captured = " + captured; // 报错: 变量 captured 未定义

输出 "以上是嵌套测试。";

// ——— 匹配测试 ———
let re = "1+4\d*";
output "正则: " + re;
let s = "Default String";
output "请在 5 秒内输入字符串 s: ";
input s timeout(5) {
    output "[[LF]]超时! [[LF]]采用默认值 s = `" + s + "`";
};
branch (s ~= re) {
    output "能匹配上";
} else {
    output "匹配不上说是";
}

// ——— 中文关键字求斐波那契 ———
定义 斐波那契 = 1;
输出 "请输入要求斐波那契数列的项数: ";
定义 项数 = 0;
输入数字 项数;
定义 当前下标 = 0;
定义 当前项 = 0;
定义 上一项 = 1;
循环 (当前下标 < 项数) {
    定义 临时 = 当前项;
    当前项 = 当前项 + 上一项;
```

```
    上一项 = 临时;
    当前下标 = 当前下标 + 1;
    输出 "第 " + 当前下标 + " 项是 " + 当前项 + "。";
}
```

../examples/test2.dszl

```
let amount = 114514;
let username = "";

state welcome() {
    output "欢迎使用本系统，请问您的用户名是？";
    input username;
    output username + "，您好！";
}

state complain() {
    output "您的意见是我们进步的动力，请问您有什么意见？";
    let complaint = "";
    input complaint;
    output "收到意见“" + complaint + "”。我们会尽快处理您的投诉。";
    thanks();
}

state bill() {
    output "您的本月账单为 " + amount + " 元。";
    thanks();
}

state thanks() {
    output "感谢您的使用。[[LF]][[LF]]";
}

while (true) {
    branch (username == "") welcome();

    output "请问有什么可以帮您？（投诉 / 账单）";
    let cmd = "";
    while (cmd == "") {
        input cmd timeout(10) {
            output "[[LF]]您似乎很久没有说话了。[[LF]]一段时间后若无响应，客服会话将自动
关闭。";
        };
```

```
        branch (cmd == "")
        input cmd timeout(5) {
            output "[[LF]]客服会话即将关闭，欢迎再次使用客服机器人，再见。";
            exit;
        };
    }
    branch (cmd ~= "投诉") complain();
    else branch (cmd ~= "账单") bill();
    else output "我不明白您的意思。请重新输入。";
}
```

../examples/test3.dszl

```
let balance = 20;
let data_usage = 2048; // MB
let cur_plan = "家庭套餐";

state check_balance() {
    output "您的当前余额为 " + balance + " 元。";
}

state check_data_usage() {
    output "您本月已使用的数据流量为 " + data_usage + " MB。";
}

state change_plan() {
    output "您现在的套餐为：" + cur_plan + "。[[LF]]请选择您想要更换的套餐（例如家庭套餐、
个人套餐、超值套餐），手续费 10 元。";
    let plan = "";
    input plan;
    while (!(plan ~= "[(家庭)(个人)(超值)]套餐")) {
        output "未知套餐，请重新输入。";
        input plan;
    }
    branch (cur_plan == plan) {
        output "您已经订阅了" + cur_plan + "！";
    } else branch (balance - 10 >= 0) {
        balance = balance - 10;
        cur_plan = plan;
        output "您已成功更换至" + plan + "。";
    } else {
        output "余额不足，无法更换！";
    }
```

```
}

state recharge() {
    output "请输入充值金额：";
    let amount = 0;
    input_num amount;
    branch (amount > 0) {
        balance = balance + amount;
        output "充值成功，您的余额现在为 " + balance + " 元。";
    } else {
        output "充值金额必须为正数！";
    }
}

state report_issue() {
    output "请描述您遇到的问题：";
    let issue = "";
    input issue;
    output "您报告的问题为：“" + issue + "”。我们的技术人员将在三个工作日内回复。";
}

state get_support() {
    output "正在为您转接至客服，请稍等。";
    sleep 2;
    output "[[LF]]……[[LF]]……[[LF]]【满意度调查】尊敬的客户，您好！为了改善中国移通服务质量，邀请您点击链接 https://m.10068.jp/SLServer/h4ShVa1uε 对相关业务及服务进行评价，感谢您的支持！【中国移通】";
}

while (true) {
    output "欢迎使用移通服务中心。请选择服务（余额查询 / 流量查询 / 套餐更换 / 充值 / 故障报告 / 转人工客服）：";
    let cmd = "";
    input cmd;
    branch (cmd ~= "余额查询") check_balance();
    else branch (cmd ~= "流量查询") check_data_usage();
    else branch (cmd ~= "套餐更换") change_plan();
    else branch (cmd ~= "充值") recharge();
    else branch (cmd ~= "故障报告") report_issue();
    else branch (cmd ~= "转人工客服") get_support();
    else output "无法识别命令，请重新输入。";
}
```

../examples/test4.dszl

```
let destination = "";
let booking_number = ""; // something strange unicode character that cannot be input
let travel_date = "20231220";
let number_of_people = 2;
let special_requests = "";

state choose_destination() {
    output "请问您计划去哪里旅游？";
    input destination;
    output "好的，您选择了" + destination + "作为旅行目的地。";
}

state gen_new_booking_number(date num) {
    booking_number = "BK" + date + "0" + num;
}

state book_trip() {
    branch (destination == "")
        output "请先选择旅行目的地。";
    else {
        output "正在为您预订" + destination + "之行。请稍等…";
        sleep 2;
        gen_new_booking_number(travel_date number_of_people);
        output "预订成功！您的预订号为：" + booking_number + "。";
    }
}

state view_booking() {
    branch (booking_number == "")
        output "您还没有预订任何旅行。";
    else
        output "您的预订信息如下：[[LF]]旅行目的地：" + destination + "[[LF]]旅行日期：" + travel_date + "[[LF]]人数：" + number_of_people + "[[LF]]特别要求：" + special_requests;
}

state change_booking() {
    output "请输入您的预订号：";
    let booking_number_i = "";
    input booking_number_i;
    branch (booking_number_i != booking_number) {
```

```
        output "预订号不匹配！";
    } else {
        output "您想要更改什么信息？（旅行日期 / 人数 / 特别要求）";
        let change_request = "";
        input change_request;
        branch (change_request == "旅行日期") {
            output "请输入新的旅行日期：";
            input travel_date;
        } else branch (change_request == "人数") {
            output "请输入新的人数：";
            input_num number_of_people;
        } else branch (change_request == "特别要求") {
            output "请描述您的特别要求：";
            input special_requests;
        }
        gen_new_booking_number(travel_date number_of_people);
        output "您的预订信息已更新。预定号为 " + booking_number + "。";
    }
}

state cancel_booking() {
    output "请输入您要取消的预订号：";
    let booking_number_i = "";
    input booking_number_i;
    branch (booking_number_i != booking_number)
        output "预订号不匹配！";
    else {
        booking_number = ""; // something strange unicode character that cannot
be input
        output "预订已取消。";
    }
}

state get_travel_advice() {
    output "请输入您想要咨询的旅行目的地：";
    input destination;
    output "关于" + destination + "的旅游信息如下 ... [[LF]]……[[LF]]……[[LF]]……";
}

while (true) {
    output "欢迎使用旅游咨询服务。请选择服务：（目的地选择 / 预订旅行 / 查看预订 / 更改预订 /
取消预订 / 获取旅游建议）";
```

```
    let cmd = "";
    input cmd;
    branch (cmd ~= "目的地选择") choose_destination();
    else branch (cmd ~= "预订旅行") book_trip();
    else branch (cmd ~= "查看预订") view_booking();
    else branch (cmd ~= "更改预订") change_booking();
    else branch (cmd ~= "取消预订") cancel_booking();
    else branch (cmd ~= "获取旅游建议") get_travel_advice();
    else output "无法识别命令，请重新输入。";
}
```

../examples/test5.dszl

```
let balance = 200;
let grades = "3.92";
let billed = false;

state wait_for_notification() {
    sleep 1;
    output "等通知。";
    sleep 2;
}

state query_balance() {
    output "您的校园卡余额是 " + balance + " 元。";
}

state pay() {
    output "请输入您的充值金额：";
    let amount = 0;
    input_num amount;
    branch (amount > 0) {
        balance = balance + amount;
        output "充值成功，您的当前余额为 " + balance + " 元。";
    } else {
        output "充值金额必须为正数。";
    }
}

state grade() {
    output "您的绩点是 " + grades + "。";
}
```

```
state billing() {
    output "计算机科学与技术专业的学费是 5500 元。";
    branch (billed) {
        output "您已经缴费。";
    } else {
        branch (balance >= 5500) {
            output "校园卡余额充足。是否现在缴费?（是 / 否）";
            let cmd = "";
            input cmd;
            branch (cmd == "是") {
                balance = balance - 5500;
                billed = true;
                output "缴费成功，您的当前余额为 " + balance + " 元。";
            } else {
                output "您已取消缴费。";
            }
        } else {
            output "您的校园卡余额不足。";
        }
    }
}

state unroll() {
    output "已为您提交退学申请。再见。";
    exit;
}

state service() {
    output "人工客服当前不在线，请留言。";
    let msg = "";
    input msg;
    output "您的留言已记录，感谢您的反馈。";
}

state default() {
    output "抱歉，我不明白您的问题。请尝试其他问题或联系人工客服。";
}

while (114 < 514) {
    output "欢迎使用学校客服系统，请向我描述您需要的帮助。[[LF]]（查询 ... 通知 / 查询校园卡余额 / 校园卡充值 / 查询成绩 / 查询学费 / 退学 / 人工）";

    let cmd = "";
```

```
    input cmd timeout(15) {
       output "[[LF]]您似乎很久没有说话了。[[LF]]一段时间后若无响应，客服会话将自动关闭。";
    };
    branch (cmd == "")
    input cmd timeout(5) {
        output "[[LF]]客服会话即将关闭，欢迎再次使用客服机器人，再见。";
        exit;
    };

    branch (cmd ~= "通知") wait_for_notification();
    else branch (cmd ~= "余额") query_balance();
    else branch (cmd ~= "充值") pay();
    else branch (cmd ~= "成绩") grade();
    else branch (cmd ~= "学费") billing();
    else branch (cmd ~= "退学") unroll();
    else branch (cmd ~= "人工") service();
    else branch (cmd ~= "查询") wait_for_notification();
    else default();
}
```

# 附录 A DSZL 配套设施

## 附录 A.1 代码配套提供的超多文档!

Rust 的包管理器 cargo 十分现代，只要运行 `cargo doc` 命令，就会生成整个项目以及所有依赖的第三方库的文档。不过我们不需要第三方库的文档，所以可以使用 `cargo doc --no-deps` 命令，只生成项目自身的文档。

要查看文档，只需要打开文件目录中 `doc/autogen/dszl/index.html` 点点点即可。以下是一些截图：

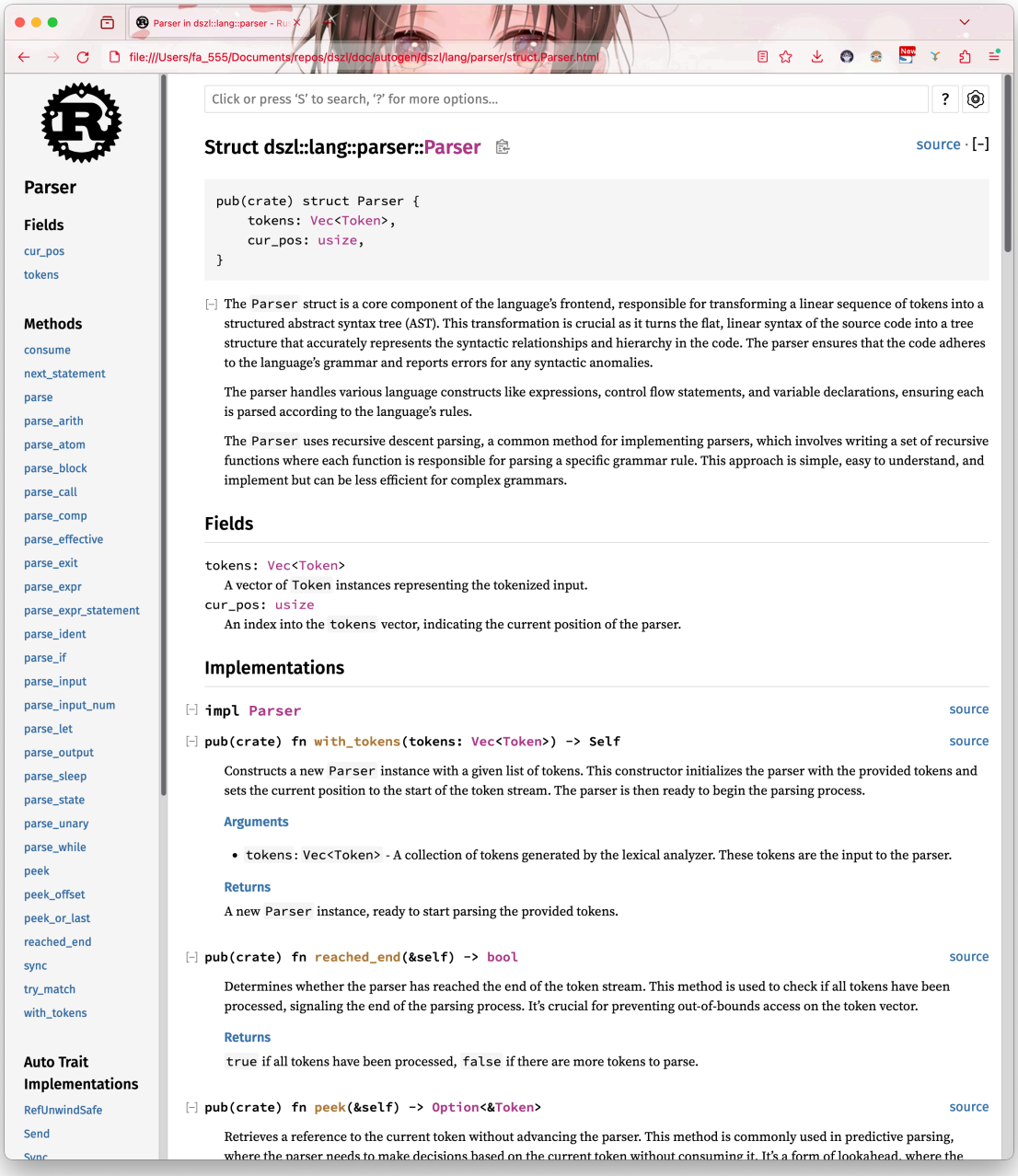


图 5.1 解析器文档：`dszl/lang/parser/struct.Parser.html`

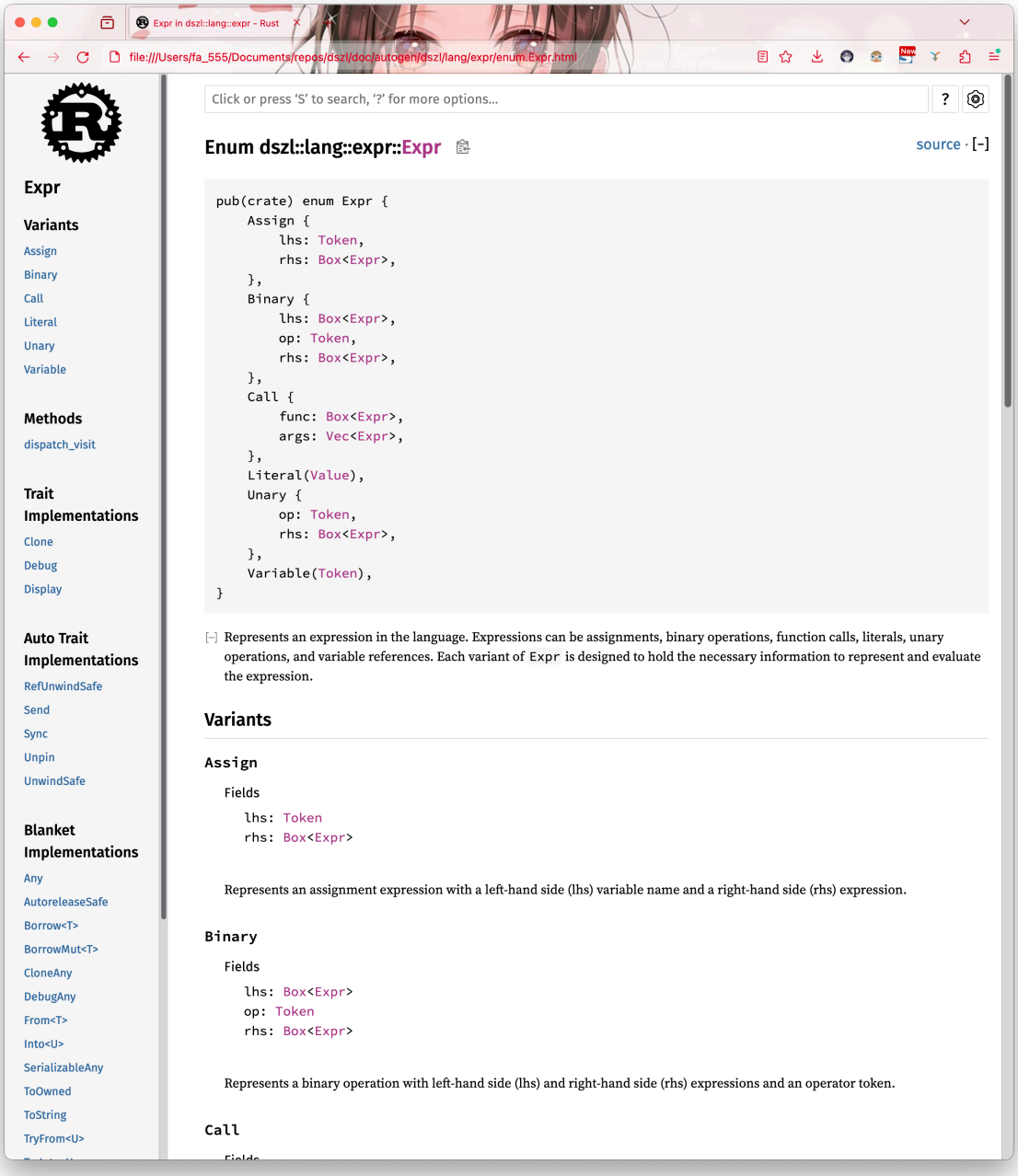


图 5.2 表达式文档：`dszl/lang/expr/enum.Expr.html`

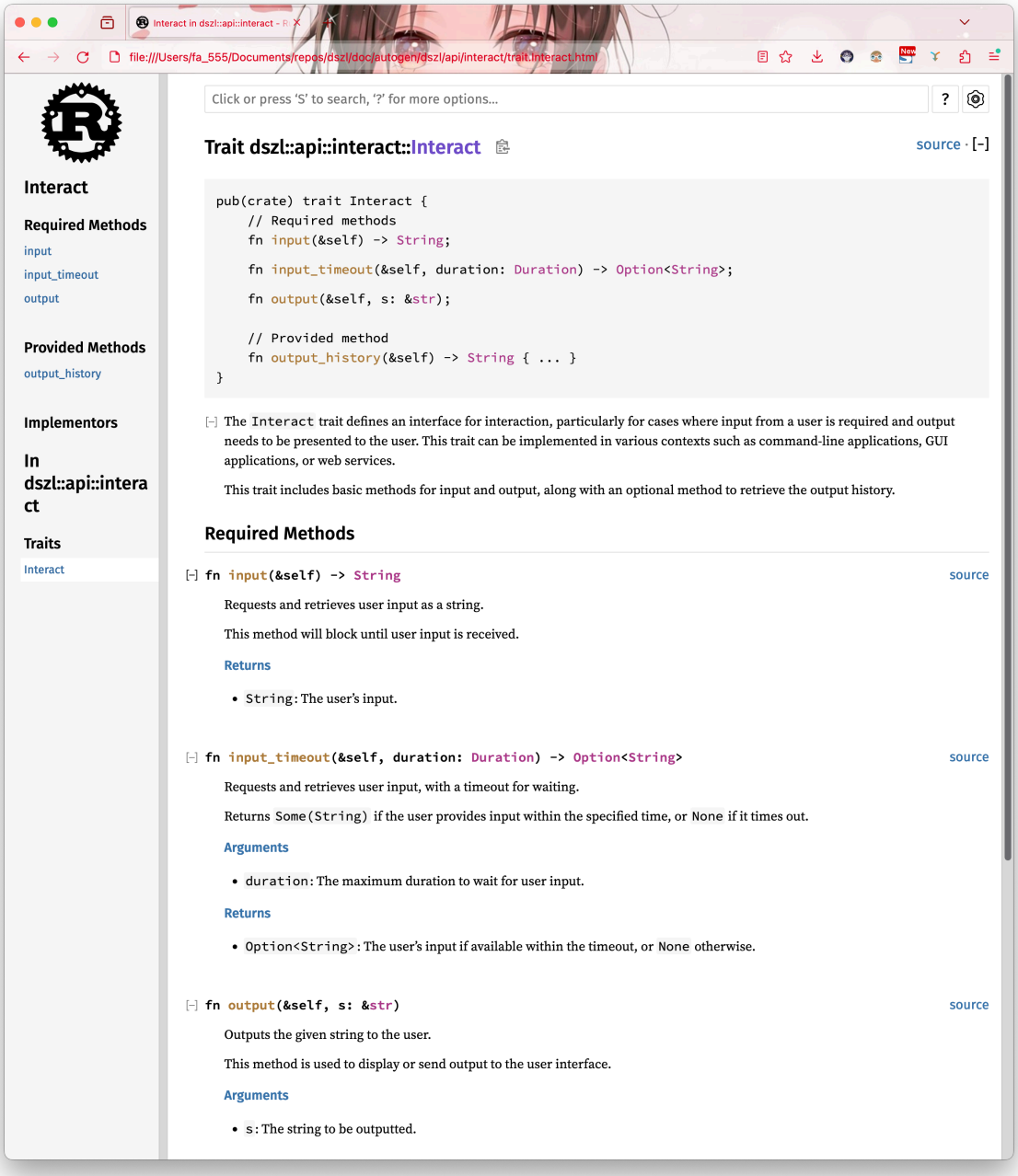


图 5.3　前后端交互接口文档：`dszl/api/interact/trait.Interact.html`

## 附录 A.2 语法高亮支持：`dszl.sublime-syntax`

`.sublime-syntax` 是一种通用的语法高亮文件格式。

DSZL 语言提供了高亮文件 `dszl.sublime-syntax`，可以在 Sublime Text 编辑器、Typst 排版工具或者其他支持该高亮格式的场合中使用。

本文档中 DSZL 语言的代码有漂亮的语法高亮，就是因为用了这个文件。

```
%YAML 1.2
---
```

```yaml
name: DSZL
file_extensions:
  - dszl
scope: source.dszl

contexts:
  main:
    - include: comments
    - include: block-comments
    - include: keywords
    - include: strings
    - include: numbers
    - include: operators
    - include: expressions

  comments:
    - match: '//.*'
      scope: comment.line.double-slash.dszl

  block-comments:
    - match: '/\*'
      scope: punctuation.definition.comment.begin.mylang
      push:
        - meta_scope: comment.block.mylang
        - match: '\*/'
          scope: punctuation.definition.comment.end.mylang
          pop: true
        - match: '(?=\*/)'
          fail: block-comments
  keywords:
    - match: '\b(branch|else|while|state|let|exit|output|sleep|input|input_num|
timeout|nil|true|false|分支|否则|循环|状态|定义|退出|输出|等待|输入|输入数字|超时|空|
真|假)\b'
      scope: keyword.control.dszl

  strings:
    - match: '"[^"]*"'
      scope: string.quoted.double.dszl

  numbers:
    - match: '\b[\.\d][\.\deE]*\b'
      scope: constant.numeric.dszl

  operators:
```

```
    - match: '\+|-|!|~|==|!=|<|<=|>|>=|&&|\||\||||==|\(|\)|{|}|;'
      scope: keyword.operator.dszl

  expressions:
    - match: '\b(?<!\d)[A-Za-z_]\w*\b(?!-keyword)'
      scope: variable.other.dszl
```