编译器设计专题实验一：词法分析

计算机 2101 田濡豪 2203113234

1 目录

2 环境配置

本人选择使用 Visual Studio Code 配合阿里云 PAI-DSW 完成实验，在阿里云中创建的 DSW 实例如下：

配置完成后，使用 Visual Studio Code SSH 远程连接时会显示实例 ID，作为本人凭据。

3 实验内容：模拟 DFA 运行实现过程

3.1 实验要求（需求分析）

- DFA 输入包括“字符集”、“状态集”、“开始状态”、“接受状态集”、“状态转换表；

- 上述输入的五元组保存在一个文本文件中，

- 对 DFA 进行检查，开始状态是否唯一，包含在状态集中，接受状态集是否为空，包含在状态集中，

- DFA 的输出，输入待显示的字符串的最大长度 N，输出以上定义的 DFA 的语言集中长度≤N 的所有规则字符串；

- DFA 的规则判定，输入（或用字符集随机生成）一个字符串，模拟 DFA 识别字符串的过程判定该字符串是否是规则字符串（属于 DFA 的语言集）；

3.2 架构设计

采用依赖倒置原则自顶向下设计。

实验的顶层需求包括：

- 加载 DFA 配置文件

- 模拟单一字符串

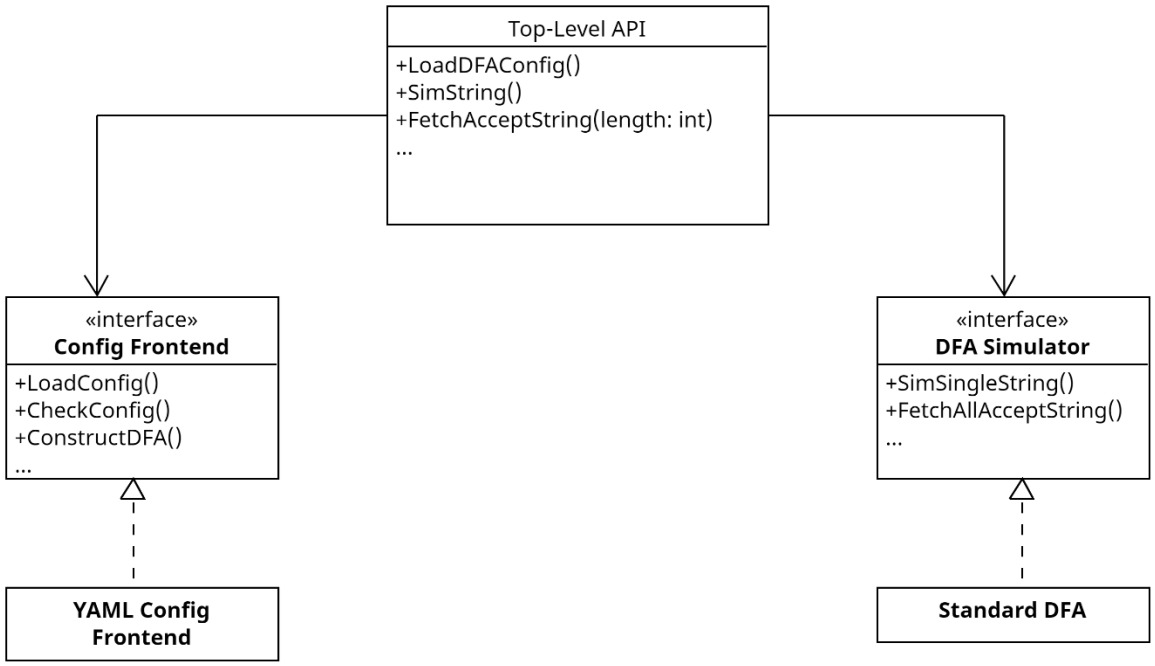
- 寻找所有符合规则的字符串

根据需求定义两个单一职责接口：

- Config Frontend：加载 DFA 配置文件，检查合法性并构造 DFA 数据模型

- DFA Simulator：模拟 DFA 的状态转换，检查字符串是否符合规则，寻找所有符合规则的字符串

UML 依赖关系图如下：



3.3 实现细节

3.3.1 DFA 模型设计

约定每个字符使用 char 实现，因此字符串可使用 std::string。

允许状态有多个字符组成，因此状态使用 std::string 描述。

状态集不允许状态重复且不作排序要求，因此使用 std::unordered_set。

状态转移表要求同上，故使用 std::unordered_map。

综上，DFA 数据模型定义如下：

```
// DFA 数据结构定义
struct DFA {
```

```
    std::unordered_set<char> character_set;                          // 字
符集
    std::unordered_set<std::string> states_set;                      // 状
态集
    std::string initial_state;                                       // 开
始状态
    std::unordered_set<std::string> accepting_states;                // 接
受状态集
    std::unordered_map<std::string, std::unordered_map<char, std::string>>
 transitions; // 状态转换表
};
```

3.3.2 配置加载及验证接口

选用 YAML 文件存储 DFA 配置，C++提供 YAML 处理标准库 yaml-cpp，因此不再详述文件加载过程，聚焦配置合法性检查。

由于 unordered_set 保证唯一性，仅检查开始状态是否包含在状态集中。采用简单的集合查找实现。

```
    if (dfa.initial_state.empty()) {
        std::cerr << "Error: Initial state is not defined" << std::endl;
        return false;
    }

    if (dfa.states_set.find(dfa.initial_state) == dfa.states_set.end()) {
        std::cerr << "Error: Initial state '" << dfa.initial_state
                  << "' is not in the states set" << std::endl;
        return false;
    }
```

其余合法性检查实现大同小异。

```
    // 检查接受状态集是否为空并包含在状态集中
    if (dfa.accepting_states.empty()) {
        std::cerr << "Error: Accepting states set is empty" << std::endl;
        return false;
    }

    for (const auto& state : dfa.accepting_states) {
        if (dfa.states_set.find(state) == dfa.states_set.end()) {
            std::cerr << "Error: Accepting state '" << state
                      << "' is not in the states set" << std::endl;
            return false;
        }
    }

    // 检查转换表中的状态是否都存在于状态集中
    for (const auto& [from_state, transitions] : dfa.transitions) {
        if (dfa.states_set.find(from_state) == dfa.states_set.end()) {
```

```
            std::cerr << "Error: Transition from state '" << from_state
                      << "' which is not in the states set" << std::endl;
            return false;
        }

        for (const auto& [input_char, to_state] : transitions) {
            if (dfa.character_set.find(input_char) == dfa.character_set.en
d()) {
                std::cerr << "Error: Transition on character '" << input_c
har
                          << "' which is not in the character set" << std:
:endl;
                return false;
            }

            if (dfa.states_set.find(to_state) == dfa.states_set.end()) {
                std::cerr << "Error: Transition to state '" << to_state
                          << "' which is not in the states set" << std::en
dl;
                return false;
            }
        }
    }
```

3.3.3 DFA 模拟器

3.3.3.1 检查单一字符串

基本检查流程设计如下：

- 初始化信息，包括一个字符串用于记录转移日志。
- 遍历字符串并进行状态转移，若无法转移则提前退出。
- 如果字符串遍历结束，检查最终状态是否可以结束。

实现如下：

```
    std::string current_state = dfa.initial_state;
    simulation_log = "Start single string simulation\n";
    simulation_log += "Target string: " + input + "\n";
    simulation_log = "Initial state: " + current_state + "\n";
    // 遍历输入字符串的每个字符
    for (const char &input_char : input) {

        // 当前字符
        simulation_log += "Current character: '" + std::string(1, input_ch
ar) + "'\n";

        // 检查是否可以转移
        if (!CheckSingleCharInSingleState(current_state, input_char)) {
            // 注意这不是程序错误，只是输入字符串不符合DFA
            // 将情况记录到日志中
```

```
            simulation_log += "No transition from state " + current_state
+ " on input '" + input_char + "'\n";
            return false;
        }
        // 进行单步模拟
        current_state = SingleStepSimulate(current_state, input_char);
        simulation_log += "Transition to state: " + current_state + "\n";
    }
    // 检查最终状态是否为接受状态
    simulation_log += "Final state: " + current_state + "\n";
    if (IsAcceptState(current_state)) {
        simulation_log += "Accepted\n";
        return true;
    } else {
        simulation_log += "Rejected\n";
        return false;
    }
```

其中几个辅助函数均通过简单的集合查找实现。

3.3.3.2 生成一定长度内符合要求的所有的字符串

使用 std::set 维护接受串集合，从而无需考虑串的重复性。

采用递归程序，对一个生成串及其当前状态：

- 如果串过长，不接受。
- 如果当前状态是结束状态，添加当前串到接收串集合。
- 如果当前状态可以转移，递归检查转移后的串。

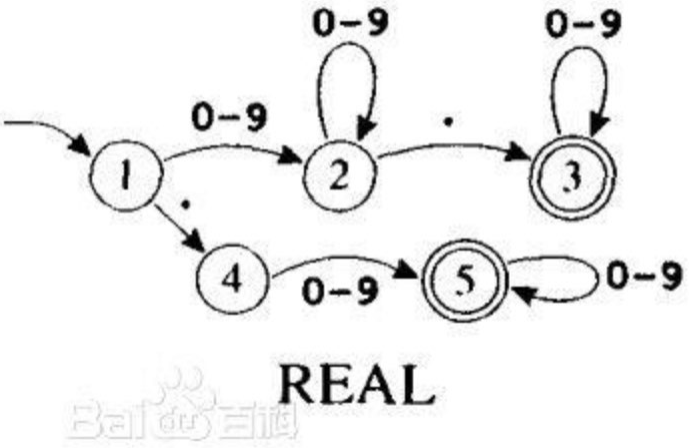
核心实现如下：

```
void StandardDFASimulator::GenerateAcceptedStrings(const std::string &curr
ent_string, const std::string &current_state, int max_length, std::set<std
::string> &accepted_strings) const {
    // 如果当前字符串长度超过最大长度，返回
    if (current_string.length() > max_length) {
        return;
    }
    // 如果当前状态是接受状态，添加到结果集中
    if (IsAcceptState(current_state)) {
        accepted_strings.insert(current_string);
    }
    // 遍历所有可能的输入字符
    for (const char &input_char : dfa.character_set) {
        // 检查是否可以转移
        if (CheckSingleCharInSingleState(current_state, input_char)) {
            // 进行单步模拟
            std::string next_state = SingleStepSimulate(current_state, inp
ut_char);
            GenerateAcceptedStrings(current_string + input_char, next_stat
e, max_length, accepted_strings);
        }
```

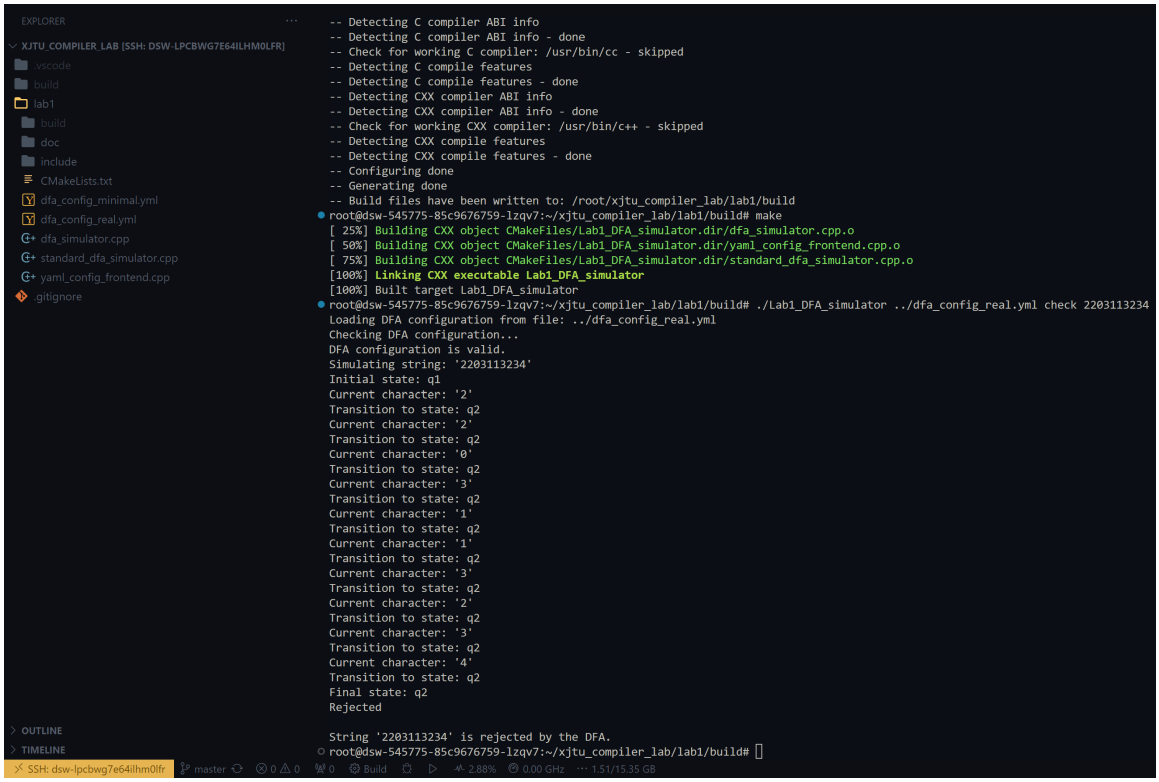
```
        // 否则字符串必然不符合规则
        // 不用继续递归
    }
```

4 实验结果

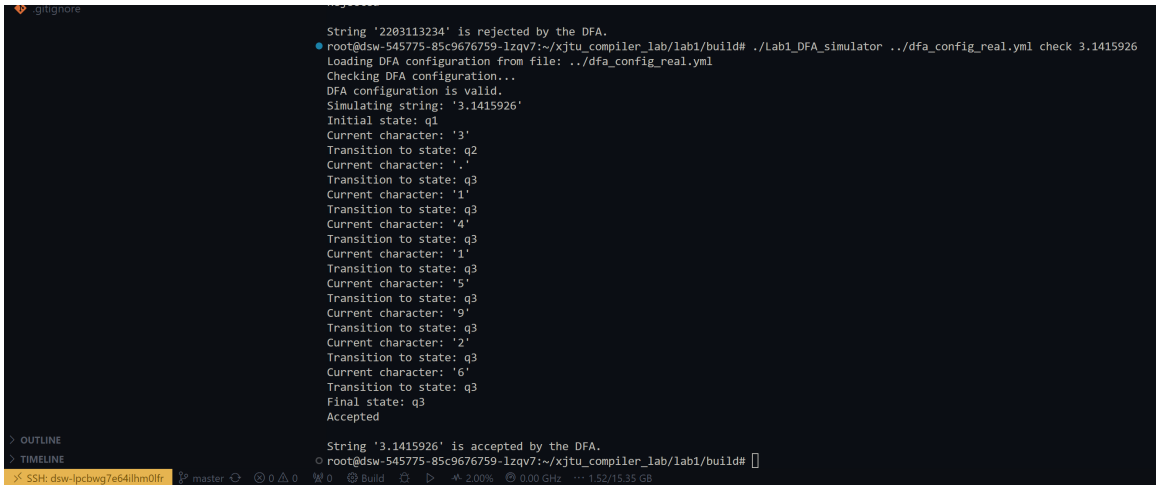
选取一个状态机，其接受“带小数点的浮点字面量”，常作为词法分析器的一部分。



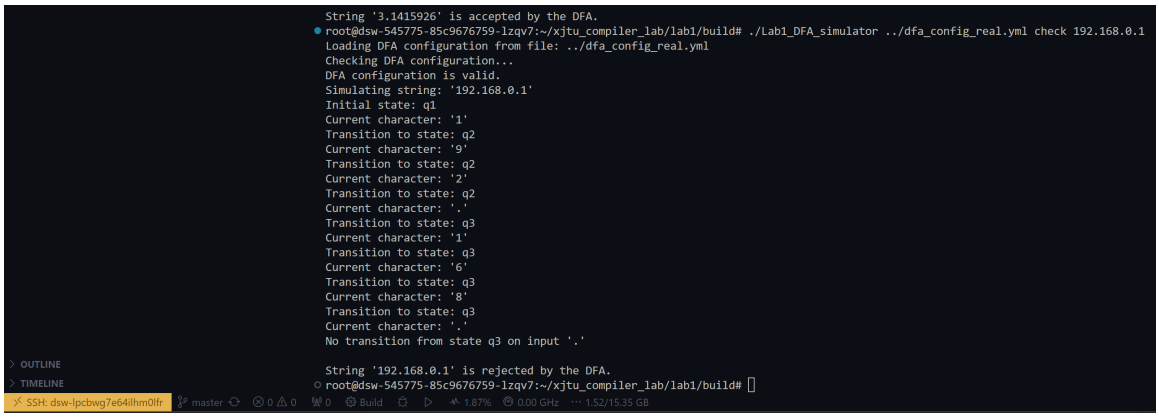
首先以本人学号 2203113234 作为测试用例。由于不含小数点，其无法被接受，最终停在状态 2：



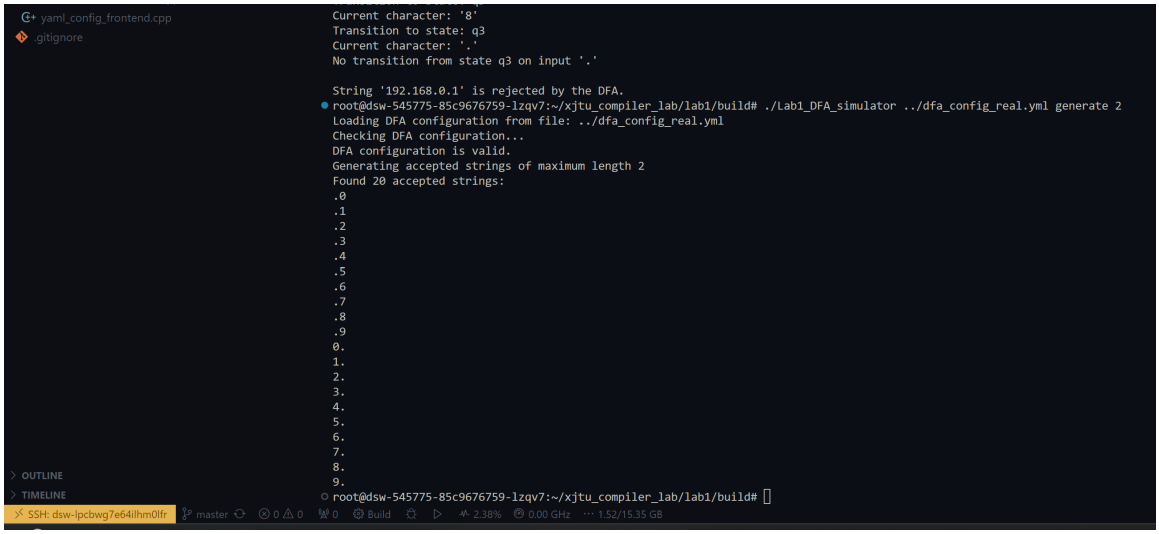
尝试 3.1415926，字符串于状态 3 被接受：



尝试 IP 地址 192.168.0.1，因有多个小数点而无法在状态 3 转移，被拒绝：



生成长度不大于 2 的所有符合条件字符串：



注释掉配置文件中的所有接受状态，可见在加载配置时报错，并且指出是接受状态错误：

把开始状态设置为状态集合之外，同样报错，可见配置检查工作正常：