形式语言与自动机 第二次实验 正则表达式的编译 实验文档

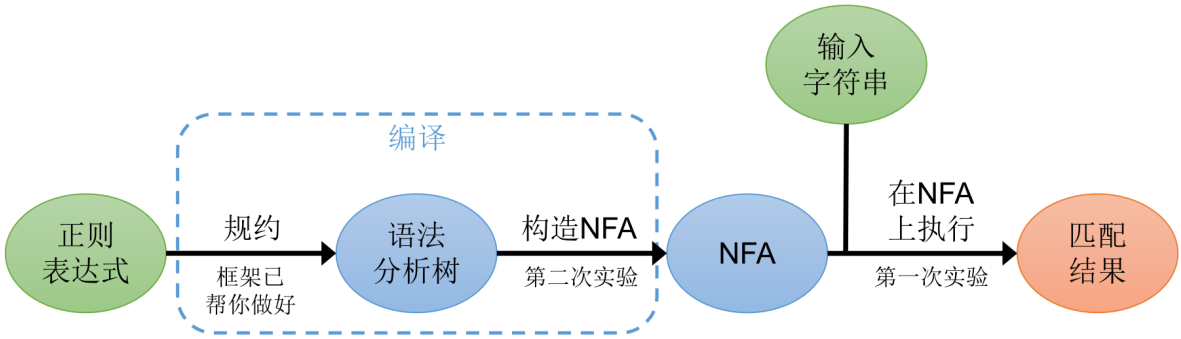
实验概述

本实验需要大家以编程的方式完成，目标是编写一个能够将正则表达式字符串转换为一个NFA的编译器，再结合第一次实验中的NFA执行器，实现用正则表达式匹配字符串。

> 2023.5.7: 本次发布的实验材料尚未包含judge，测试样例也暂时只包含了3个，非常抱歉。缺失的内容将在三天内补充完整，并在群内发布公告。

实验思路

为了实现一个正则表达式引擎(Regular Expression Engine)，我们整体的思路如下图所示：



对输入的正则表达式，我们首先：

- 将输入的正则表达式，通过类似于课上讲的规约的过程，得到语法分析树。
  - 虽然我们课上讲了规约的定义和例子，但如何程序化地进行规约，实际上是之后的《编译原理》课程的重要内容，而非本门课程的要求。
  - 因此，这个过程已经借助ANTLR帮你实现好，你无须自己实现。
- 利用语法分析树的分析结果，构造NFA。
  - 这个过程关联的知识点有：第七讲的语法分析树，和第四讲的由正则表示构造NFA。
  - 这是你本次实验实现的重点。
- 将输入字符串在构造好的NFA上执行，得到执行结果。
  - 这是上一次实验完成的内容。
  - 但是请注意，你将需要对之前完成的代码**进行必要的修改**，详见后文的需要对NFA进行的修改部分。
  - 此外，对于涉及返回分组的情况，还需要在构造NFA时和拿到NFA的执行结果后进行一些特殊处理。（本次实验暂不涉及）

编程语言

本实验需要大家在**C++**或**Python**语言中任选一种进行完成。
对于每种语言，我们都提供了一套SDK（半完成的代码框架），大家只需按要求完成相应的函数即可。

关于外部依赖，在全部的实验中，**不允许大家使用任何的外部依赖**。
即C++语言只能使用标准库，Python语言只能使用系统库，不可以引入任何第三方库（无论是以源代码复制、pip、cmake或是任何其他形式都不可以）。此外，标准库或系统库中与正则表达式有关的库（如C++的std::regex，Python的re）也不可以使用。

实验框架内容概述

你收到的实验框架中包含的内容和含义描述如下表：

目录名	描述
cpp	C++语言的编程框架。具体的用法参见实验具体说明的C++语言部分。
python	Python语言的编程框架。具体的用法参见实验具体说明的Python语言部分。
antlr	包含正则表达式的文法定义`regex.g4`和ANTLR的jar文件等。具体的用法参见实验具体说明的ANTLR部分。
cases	存放测试样例的文件夹。每个测试样例是一个txt文件。
judge	内含不同平台上的评测器的可执行文件。命名均为judge-操作系统-架构，请自行选择合适的评测器运行。如果没有你合适的平台的可执行文件，请直接联系助教。

实验要求

本次实验采取全自动评测的形式。
自动评测的含义是：你的程序将被多次调用、输入多个不同的测试样例进行测试。对每个测试样例，若你的程序在规定时限内正常返回并给出正确的结果，则该测例得分，否则不得分。
本次实验**不需要提交文档**。

提交方法

请在网络学堂的DDL前，将你的作业提交到**网络学堂**。提交的内容为一个压缩包（建议最好是`.zip`格式），内容包含：

- 若你是使用C++语言完成，请提交：
  - `cpp`文件夹
    - 请确保删除了`cpp`文件夹内的中间产物文件夹如`build`、`cmake-build-*`等。
    - 请勿删除`CMakeLists.txt`、`lib`文件夹和`parser`文件夹，否则你的代码将无法编译！
  - 在你编程的平台上编译出的可执行文件。Windows请命名为`regex.exe`，Mac和Linux请命名为`regex`（即CMake的默认编译输出名）
    - 此项用于在助教无法亲自编译你的代码的情况下作为参考。如果助教能够成功编译你的代码，则不会使用你提交的可执行文件，此种情况下即使你未提交可执行文件也不会扣分。
  - 一个简短的TXT文件，命名为`README.txt`，说明你的编程平台（如操作系统、架构、编译器等）。如果有其他想给助教留言的也可以写在这里。
  - 请不要提交除上述内容外的其他东西。特别是注意不要提交`python`文件夹，否则可能造成评测器混淆你的编程语言，严重影响评测分数。
- 若你是使用Python语言完成，请提交：
  - `python`文件夹。
    - 请勿删除`parser`文件夹。
  - 一个简短的TXT文件，命名为`README.txt`，说明你的编程平台（如操作系统、架构、Python版本等）。如果有其他想给助教留言的也可以写在这里。

    - 请不要提交除上述内容外的其他东西。特别是注意不要提交`cpp`文件夹，否则可能造成评测器混淆你的编程语言，严重影响评测分数。
- 此外，虽然我们要求原则上不得修改文法文件，但如果你确实由于某些原因修改了文法，则请额外提交`regex.g4`文件备查。

评分规则

本次实验占据实验部分总成绩的30%。

本次实验的评分规则为：

- 自动评测 100%
    - 公开测例 60% (20个，每个测例分值均等)
        - 已包含在本次下发的实验框架中，同时提供了自我测评用的评测器。
        - **使用评测器得到的自测得分仅供参考**，不作为得分的直接依据。
    - 隐藏测例 40% (每个测例分值均等)
        - 不会公开给同学。将在DDL后由助教进行测试。
    - 所有的测例，无论是公开还是隐藏，均将**由助教在DDL后统一运行你的代码进行测试并计算得分**。得分以助教评测的结果为准。
- 减分项：如你存在下列问题，可能会被额外进行惩罚性的减分。
    - 抄袭：**本实验和其他的所有实验均严禁抄袭**。抄袭者最严重将被处以所有实验全部0分的惩罚。不能给出合理解释的代码高度雷同也被视为抄袭。
    - 攻击评测机：禁止用任何方式攻击评测机，包括但不限于尝试访问、修改与自己的实验无关的文件、执行恶意代码、尝试提权等行为。违反者视情节，最严重将被处以所有实验全部0分的惩罚，如涉及违纪违法还将上报学校等有关部门处理。
    - 使用非正常手段通过测例：包括但不限于针对特定的输入直接匹配输出，通过联网、调用评测机等手段从外部来源获取答案等。违反者将被扣除所有以非正常手段通过的测例的得分。
    - 未按要求提交：请严格按照上面的提交要求提交作业。
        - 若未按要求提交引起评测器无法自动评测分数的（如同时提交cpp和python文件夹等行为），将被至多扣除本次实验总分的20%。
        - 其他情况，至多扣除本次实验总分的10%。

迟交政策

- 每迟交一天，分数扣减5%，至多扣减50%。
- 扣减是在正常方法计算的应得分的基础上按比例扣减的。
    - 例如，迟交3天，正常计算的应得分为90分，则实际最终得分为90*(1-5%*3)=76.5分（分数舍入到小数点后一位）。

其他

- 第二次实验，保证无论是正则表达式(pattern)字符串，还是待匹配文本字符串中，都只包含ASCII字符（字节值0~127），且不会包含NULL字符`\0`和换行符`\r` `\n`。
- 程序“正常返回”的定义：你的程序正常结束执行，进程返回码为0。
- 助教评分时使用的评测环境：
    - Ubuntu 22.04 LTS (in docker)
    - Intel i7-12700K (5.0GHz)

    - Python 3.11
- 关于程序限时：本实验给出的限时可以说是非常宽松的。
    - 对于C++语言，每个测例限时5s；对于Python语言，每个测例限时10s。
    - 限时一律以程序在助教的评测机上的用时为准，均包括IO时间。
    - 请注意，提供给你的自测用评测器**只实现了统计限时，但并未实现超时杀死进程的机制**。
        - 即，无论你的程序运行多久，评测器都会一直等待直到你的进程退出，然后计算你的程序执行的用时是否超时。
        - 超时的测例在评测器中会分类在“发生异常”类别中。

实验具体说明

关于ANTLR

ANTLR(ANother Tool for Language Recognition)是一款强大的语法解析器生成器，可以用于读取、处理、执行或翻译结构化的文本文件或二进制文件，在许多编程语言、工具和框架中被广泛应用。
给定一个文法，ANTLR可以生成一个针对该文法的解析器(parser)，利用该解析器，可以对任意符合文法的输入字符串生成语法分析树，并支持方便地在树上进行遍历。
ANTLR所使用的技术是Adaptive LL(*)，分为词法分析、语法分析两个部分。词法分析是由输入文本生成token（即文法的终结符）序列，语法分析则是由词法分析的结果(终结符的序列)，进行规约过程、生成语法分析树。

ANTLR由Java语言编写而成，本质可以视为是一个代码生成器(codegen)，由给定的文法（在本实验中是`antlr/regex.g4`），生成可以用于解析该文法的字符串的parser代码（在本实验中是`cpp`、`python`等目录下的`parser`文件夹），生成的代码可以是许多种不同的语言。
生成代码的过程依赖ANTLR的主程序，因此必须使用Java解释器；而生成好的代码则不需要Java，只需要对应语言的ANTLR运行环境(runtime)，就可以运行。

正则表达式的文法定义文件是`antlr/regex.g4`。你不得修改此文件，如对文件内的内容有疑问，请联系助教。
同时，考虑到不是所有同学都会使用Java，我们已经提前完成好了生成parser的过程，并为你生成了三种不同语言的parser代码，包含在作业框架内：

- `cpp/parser`文件夹：是C++语言的parser。调用该parser的过程也已写成`Regex::parse`函数。
- `python/antlr_parser`：是Python语言的parser。调用该parser的过程也已写成`Regex`类下的`parse`函数。
    - 之所以Python不叫`parser`，是因为在Python3.9或以下的版本中，有一个系统库名叫`parser`，会导致命名冲突。
- `antlr/parser`：是Java语言的parser。此parser是供大家自愿使用TestRig可视化查看语法分析树时使用的，与实验本身无关。

文法定义

为了完成根据语法分析树生成NFA的过程，你必须了解语法分析树内会有哪些类型的节点，和正则表达式的文法定义，因此，**你必须阅读并理解位于`antlr`文件夹下的文法定义文件`regex.g4`**。
这个文件的本质就是一个上下文无关文法的定义文件，由许多的产生式和终结符的定义构成。

关于antlr的文法定义格式，你需要了解：

- 在antlr的文法里，**终结符的定义与课上有所不同**。
    - 课上的文法，推导出的是以字符为单位的字符串，因此终结符都是单个的字符。

    - 然而在我们的实验中，文法推导出的是连续的**token**构成的序列，终结符是token，即具有特定含义的字符串的最小单元。
    - 在正则表达式里，确实多数情况下，单个字符就是一个token，但也有很多例外：
        - 类似`\d` `\w`这种元字符，它们表示一个整体的意义。单独拆开讨论`\`和`d`是没有任何意义的，因此，`\d`整体是一个token。
        - 正则表达式中存在转义字符。如果你想匹配一个`(`，则你的正则表达式必须写成`\(`。此时，`\(`是一个token。
        - 有一些特殊的写法，本身就是有多个字符组成的。在我们要求的文法里，这样的例子只有非捕获分组`?:`一个。
- 每一条规则都形如`xx : ... ;`，即以符号的名字加上冒号`:`开头，以分号`;`结尾。
- 根据antlr的规定，文法定义中，**非终结符必须以小写字母开头，终结符必须以大写字母开头**。
    - 形如`aa : bb cc;`这样的式子，表示的是一条产生式，非终结符`aa`可以生成非终结符`bb`连接上非终结符`cc`。
    - 形如`Hat : '^';`这样的式子，表示的是一个终结符的定义，终结符`Hat`的定义是单个字符`^`。
        - 或者，终结符的定义的右侧可以使用中括号字符组，如`Digit : [0-9];`表示的是终结符`Digit`的定义是任何单个数字。
- 当一个（一串）原始输入可以匹配到多个终结符的定义时，**优先级首先按照匹配的长度，其次按照它们在文法中出现顺序排序**。
    - 例如，有定义`Hat : '^'; Digit : [0-9]; Char: . ;`。那么，一个字符`^`总会被解析为`Hat`类型的token，而永远不会被解析为`Digit`或`Char`类型；
    - 类似地，一个字符`0`总会被解析为`Digit`类型的token，而永远不会被解析为`Char`类型。
    - 换言之，由于`Digit`的存在，你可以认为`Char`的定义实际上是`~[0-9]`，即数字永远都不会被匹配为Char类型。

下面是`regex.g4`文件的片段（经过修改和简化），我们在其中以注释的形式做了一些具体的解释说明。

```
grammar regex; // 这行是固定格式的文件头，表示定义了一个名为regex的文法。

// 这是一个产生式。表示一个正则表达式(regex)至少包含一个expression，并可通过`|`符号连接更多的expression。
// 例如，`aa|bb|cc`是一个有效的正则表达式，它由`aa`、`bb`、`cc`三个expression或起来构成。
regex : expression ('|' expression)* ;

// 这个产生式表示一个expression由许多个expressionItem构成。
// 例如，`abc[A-Z](de|fg)h+i`这个expression，是由`a`、`b`、`c`、`[A-Z]`、`(de|fg)`、`h+`、`i`共7个expressionItem构成的。
expression : expressionItem+;

// 这个产生式表示一个expressionItems是由一个normalItem，加上一个可选的quantifier限定符(即*、+、?)构成的。
// 例如，`h+`expressionItem，其中`h`normalItem，`+`是quantifier`。
expressionItem : normalItem quantifier? ;

// 所有的规则都以分号结尾，因此，以下四行是一条产生式
// 表示normalItem要么是一个(能匹配一个字符的Item，如`a`、`\d`、`[A-Z]`)，要么是一个括号分组(如`(de|fg)`)。
normalItem
```

- 对文法中的**每个形如`xxx : yyy ...`的产生式**：
  - 当产生式生成的`yyy`类子节点**至多只有一个时**（即没有*、+之类的修饰符修饰时），在`XxxContext`类上会有函数：
    - `YyyContext* XxxContext::yyy();`
      - 返回当前节点上的`yyy`子节点。如果没有此类节点，则返回空指针。
  - 当产生式生成的`yyy`类子节点**可能有多个时**（通常是因为有*、+之类的修饰符修饰），在`XxxContext`类上会有**两个**函数：
    - `std::vector<YyyContext*> XxxContext::yyy();`
      - 返回当前`xxx`节点上的所有`yyy`节点构成的**列表**。如果没有此类节点，则返回长度为0的空列表。
    - `std::vector<YyyContext*> XxxContext::yyy(size_t i);`
      - 返回当前`xxx`节点上的第`i`个`yyy`节点（下标从0开始）。如果没有第`i`个节点，则返回空指针。
  - **这将是你最常用的函数**。理论上，只通过这组函数，你就足以访问到语法分析树上的所有你需要用到的节点了。
  - **注意**：特殊地，在C++语言中，由于`char`是关键字、无法用作函数名，用于访问`char`类型子节点的函数名为`char_()`。Python语言不受此影响，还是叫`char()`。
  - 例如：对字符串`ab+`，和文法

```
regex : expression ('|' expression)* ;
expression : expressionItem+ ;
expressionItem : normalItem quantifier?
```

可使用如下代码进行解析：

```
RegexContext *tree; // 假设你已经拿到了一个RegexContext节点。（PS：这就是整个语法分析树的根节点，parse函数直接返回的）
ExpressionContext *e0 = tree->expression(0); // 获得第0个expression，本例中是`ab+`（一共只有一个expression）
// tree->expression(1)将返回nullptr

ExpressionItemContext *i0 = e0->expressionItem(0); // 获得第0个expressionItem，即`a`
NormalItemContext *i0n = i0->normalItem(); // 获得对应的NormalItemContext对象，可供继续往下处理
// i0->quantifier()将返回nullptr，因为本例中`a`并没有修饰符

ExpressionItemContext *i1 = e0->expressionItem(1); // 获得第1个expressionItem，即`b+`
NormalItemContext *i1n = i0->normalItem(); // 获得对应的NormalItemContext对象，即`b`
QuantifierContext *i1q = i0->quantifier(); // 获得对应的QuantifierContext对象，即那个修饰b的`+`
// 继续往下还可以有 i1n->single(); i1q->quantifierType(); 等
// e0->expressionItem(2)将返回nullptr，因为`ab+`这个expression只有2个expressionItem
```

除此之外，我们还将介绍一些各个类上的常用函数：

- `ParseTreeType ParseTree::getTreeType()`

    - 获得节点的类型。
    - `antlr4::tree::ParseTreeType`是一个枚举：

```
enum class ParseTreeType : size_t {
  TERMINAL = 1,
  ERROR = 2,
  RULE = 3,
};
```

    - 注意，Python语言没有这个函数。在Python中判断节点的类型，建议使用isinstance判断：

```
from antlr4 import RuleContext, TerminalNode
isinstance(node, RuleContext) # 当node是rule节点（非终结符、非叶子节点）时返回true
isinstance(node, TerminalNode) # 当node是终结符节点（叶子节点）时返回true
```

- `std::vector<ParseTree*> ParseTree::getText();`
    - 获得对应于该节点的字符串，即从该节点往下推导出的字符串。
    - 常用用途是访问到叶子节点后，获得叶子节点的具体字符内容。
- `std::vector<ParseTree*> ParseTree::children();`
    - 获得一个节点的所有子节点。
    - 当你想要按顺序获得所有子节点、而不在乎它们的类型时，可以使用本函数。
- `size_t RuleContext::getRuleIndex();`
    - 获得一个非终结符(rule)的编号。
    - 用于确定一个非终结符节点(RuleContext)的具体类型。用途通常是与`ParseTree::children();`函数合用。
    - 所有rule的编号是通过regexParser下的一个匿名枚举定义的，可参考`cpp/parser/regexParser.h`文件的第24行左右的位置。
    - 使用例：

```
ExpressionContext* e0;
for (auto child : e0->children) {
    if (child->getTreeType() == antlr4::tree::ParseTreeType::RULE) { // 如果是规则节点（非叶子节点）的话
        auto ruleNode = (antlr4::RuleContext*)child;
// 上面两行的写法是根据getTreeType()判断类型，再强制转换。或者，写成动态类型转换也是可以的，如下面两行：
//  auto ruleNode = dynamic_cast<antlr4::RuleContext*>(child);
//  if (ruleNode) {

        if (ruleNode->getRuleIndex() == regexParser::RuleExpressionItem) { // 如果是expressionItem类型的节点的话
            auto itemNode = (regexParser::ExpressionItemContext*)ruleNode;
            // ...确认了节点的类型，进一步进行操作
        }
    }
}
```

利用上面介绍的API，你应当已经足以完成本次实验。

当然，除了我们介绍的API之外，ANTLR还有许多API你可自行了解。你可参考ANTLR的API文档自行了解和学习。一些常用类的快速索引：ParseTree TerminalNode RuleContext ParserRuleContext

使用TestRig可视化查看语法分析树

对于一个给定的字符串，你可能会想要可视化地查看它规约得到的语法分析树的结果。
针对此需求，我们为你准备了`testRig`脚本，它利用ANTLR Java主程序包中自带的TestRig功能，可视化一个语法分析树。
该功能并非完成本次实验所必需的，但如果想要使用该功能，则必须安装Java运行环境。如你想要安装一个，我们推荐使用Adoptium OpenJDK，你可自行点击链接进入其官网下载安装。

使用方法：

- 在任意工作目录下执行`testRig`脚本，该脚本位于`antlr`目录下。
  - 类Unix系统(Linux、MacOS): 请使用`testRig.sh`
  - Windows系统：请使用`testRig.bat`
- 按照提示，输入要解析的字符串。输入完后，请直接关闭输入流：
  - 类Unix系统下，按两次Ctrl+D。
  - Windows下，先回车，再按Ctrl+Z（屏幕上出现`^Z`），再回车。
    - 这种方法的缺点是，总是会在输入的字符串末尾加上换行符`\r\n`。目前尚没找到方法规避这个问题，如你有好的建议，欢迎提出！
- 终端上将打印词法分析的结果(token序列)，同时，将打开一个新的窗口，可以直接可视化地看到语法分析树。

需要对NFA进行的修改

在第一次实验中，我们的NFA输出接受状态，要求必须是当整个字符串都输入完后自动机停在终态。这也符合我们课上的定义。
然而，当你想要进行正则表达式的匹配时，所使用的自动机必须进行修改：它不再是要求必须输完整个串，而是只要到达终态，就说明找到了一个匹配，无论串是否已经修改。
因此，你需要对你第一次实验的NFA代码加以修改，具体而言，在自动机执行(DFS)的过程中，只要到达终态，就立即返回Path。

其他提示

- 本次作业的重点在于，你要理解和学会在ANTLR产生的语法分析树上进行遍历的过程，和不同种类的产生式应当如何构造\整合自动机。
  - 因此，你需要阅读并理解关于ANTLR部分的内容，和`regex.g4`文法定义。
  - 不同种类的产生式应当如何构造\整合自动机，需要你自己思考。
    - 一些提示性的思路是，你应当自上而下的遍历树的节点。
    - 每个子树实际上都可对应一个自动机，而父节点处的自动机就是把所有子树对应的自动机以某种方式组合一下即可。
    - 为了完成这种组合，你可能需要一定的技巧，甚至考虑构建一些新的数据结构。
    - 课上的讲解当中有更详细的思路提示。
  - 在自动机上执行字符串，当到达某个终态时，说明当前已经输入的字符串与自动机对应的pattern相匹配。
    - 然而，直接用整个输入字符串这样去做，似乎还不够完整；这样只能找到文本开头就与pattern相匹配的结果，而不能实现从文本的中部匹配。
    - 对于如何用自动机实现从任意位置匹配，你可能有很多思考。我们鼓励你思考效率更高的方法，但实际上，即使是以每个字符为开头执行一次的方法，也是能满足我们的要求的。

- 推荐使用regexr.com。这个网站可以使你在线地执行正则表达式、可视化地查看执行的结果，还能看到模式串例每个字符的含义。

代码框架公共说明——对所有的语言都适用

- cases中的测试样例均为文本文件，内含正则表达式的pattern字符串和输入字符串，文件的格式是很容易理解的，如有需要，你也可以在其基础上进行修改/编写自己的测例进行测试。
    - 如你需要自行修改/构造测试样例，**建议复制出来改动而不是在测例上直接修改**，否则如果你忘改回去了可能会造成自测分数与最终分数不一致
- 你所需要做的是完成`Regex`类的`compile`和`match`两个函数。在代码中已经使用TODO注释为你标记好。
    - 你的程序并不需要亲自从stdin中读取输入，也不需要亲自向stdout中写入结果。
    - 框架已经实现好了读取并解析文本输入，构造`Regex`类的对象并调用compile方法，再用输入字符串调用match方法，最后将match的结果输出到stdout的逻辑。
        - 如你感兴趣，可看`main-regex.cpp`或`regex.py`文件最下面的`if __name__ == '__main__'`部分。
    - 你的程序**尽量不要在stdout中打印输出**，如果确实需要打印，请**尽量打印在stderr中**。
        - 至少，请确保你打印的内容结尾有换行符。评测器的逻辑是读取stdout的最后一行作为评测的依据。
- 程序的调用方法分为两种：
    - 若不传入任何参数，则程序将从stdin中读取输入。（也是评测器评测时所用的方式）
    - 或者，你可以传入一个参数，是输入文件的路径。此时程序将改为从你指定的文件路径中读取输入。（当你想要具体的在某个测例上执行和调试时，这种方法更为方便些）
- **仔细阅读框架代码的注释！** 很多问题，包括类的含义、函数的含义、返回值的方式等，都可以在框架代码的注释中可以找到答案。
    - 框架中已经定义好了一些和函数，类内也已经定义好了一些成员变量和方法。不建议大家修改这些已经定义好的东西。
    - 但是，你可以自由地增加新的函数、类等，包括可以在已经定义好的类自由地添加新的成员变量和方法。如果你确实需要，也同样可以增加新的文件（但C++语言请注意将新增的文件加到`CMakeLists.txt`的名为`regex`的target里）。

C++语言

编译执行方法

本框架的C++语言部分使用CMake作为构建的工具。

IDE使用提示

请参见第一次实验文档。

直接在命令行中编译运行

或者，若你想直接使用命令行进行编译，方法如下：

```
cd cpp
mkdir build # 作为编译结果（可执行文件）和各类编译中间产物存储的文件夹
cd build
cmake .. # 意思是去找上级目录（此时你在build中，上级目录就是cpp）中的CMakeLists.txt文件，据此在当前目录(build)中进行中间产物的生成。这步cmake会帮你生成好一个Makefile。
cmake --build . # 执行编译
```

执行文件的方法：（注意windows平台上是regex.exe）

```
./regex # 程序会从stdin中读取数据，请自行使用输入重定向 < 、管道 | 等手段为它提供输入
./regex ../../cases/01.txt # 程序会从指定的路径读取输入。此处假定你在cpp/build文件夹下，故测试样例的相对路径应如同这个样子
```

代码结构具体描述

- `Regex`类：`regex.h` `regex.cpp`
  - 包括`Regex`类的定义。
  - 你需要实现的是`Regex::compile`和`Regex::match`函数，其参数和返回值含义均在注释上。请在`regex.cpp`中完成其实现。
- 入口点文件：`main-nfa.cpp`
  - 你应该不需要去管这个文件。这个仅包含`main`函数的实现，其中会构造`Regex`类的对象和调用`exec`、`match`等方法。

Python语言

运行方法

需要的Python版本应>=3.8，否则可能会无法运行。
首次执行代码前，务必先安装依赖:

```
pip install -r requirements.txt
```

本次实验的入口点文件为`regex.py`。

```
python regex.py # 程序会从stdin中读取数据，请自行使用输入重定向 < 、管道 | 等手段为它提供输入
python regex.py ../cases/01.txt # 程序会从指定的路径读取输入。此处假定你在python文件夹下，故测试样例的相对路径应如同这个样子
# 或者，如果你使用类Unix系统(Linux、Mac)，由于nfa文件中包含了Shebang，也可以不必输入python、直接执行regex.py
./regex.py
```

代码具体描述：`regex.py`

- 包括`Regex`类的定义。
- 你需要实现的是`Regex`类中的`compile`和`match`函数，其参数和返回值含义均在注释上。

评测器的使用方法

请参见第一次实验文档。

此外，输入`./judge-xxx --help`可以查看程序内置的使用帮助。