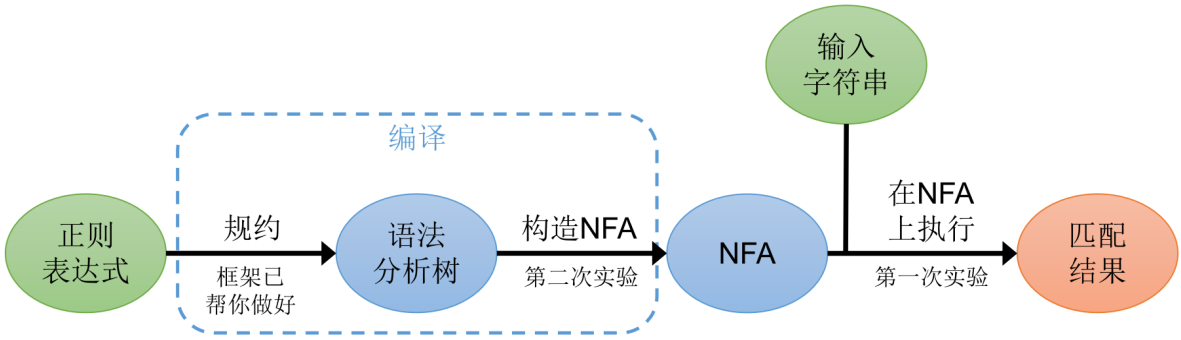
形式语言与自动机 第二次实验 正则表达式的编译 实验文档

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。

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。

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。

代码框架内容概述

代码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。你不应修改此文件夹下的任何内容。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。

提交方法

请提交到本课程的OJ上（须在清华校园网环境下访问，校外请用SSLVPN客户端）。登录OJ所用的账号密码已经通过网络学堂下发给大家。

请以 **.zip格式**的压缩包进行提交，压缩包中的内容请遵守以下规则：

- 若你是使用C++语言完成，请仅提交`src`文件夹下的内容。
  - 不必提交`lib`和`parser`文件夹。完成实验过程中也请勿对这两个文件夹内的内容做任何修改。
    - 即使你提交上来了，OJ评测时也会将它们还原到原始的版本。
  - 请确保删除了编译产物文件夹如`build`、`cmake-build-*`等。
  - 请勿删除`CMakeLists.txt`，否则你的代码将无法编译！
- 若你是使用Python语言完成，请提交`python`文件夹下的内容。
  - 其中，`python`文件夹下的`antlr_parser`子文件夹可不必提交上来。完成实验过程中也请勿对这个文件夹内的内容做任何修改。
    - 即使你提交上来了，OJ评测时也会将它们还原到原始的版本。
- 不需要提交文档。

提交时请注意，不要使用OJ中的“在线编程模式”，而是**点击“递交”**，然后通过 **“Or upload a file”按钮**上传你的压缩包，如下图所示。

评分规则

本次实验占据实验部分总成绩的40%。

本次实验的评分规则如下：

- 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。在DDL前你提交到OJ时，OJ显示的分数即为本部分的分数。
  - 最终的成绩以DDL后使用全部公开和隐藏测例重测得到的成绩为准。
- 隐藏测例 40% (每个测例分值均等)
  - 不会公开给同学。将在DDL一段时间后，与全部公开测例一起进行重测，重测后显示的得分即为最终总得分。
- 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。
  - 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。

迟交政策如下：

- 每迟交一天，分数扣减5%，至多扣减60%。
  - 接受补交的最晚日期另行通知。
- 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。
- 如你迟交较多、OJ上的作业窗口已经关闭，则你可以先通过“题库”进行公开测例的自助测试。然后在你准备好后，单独联系助教进行最终的十成测。
  - 在此种情况下，你只有一次十成测的机会。

其他

- 第二次实验中：
  - 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`。
  - 额外地，pattern字符串中保证不会含有换行符`\r` `\n`(即pattern只有一行)。
  - 但**待匹配文本字符串中可能含有换行符**(即文本可能具有多行)。这是与第一次实验不同的。
- 数据约定：对全部的测例，正则表达式长度<=500，且保证存在一种NFA的构造，满足NFA状态数<=500，总转移规则条数<=1000；输入字符串长度<=10000。
  - 事实上，你只需要用普通的思路，不太需要什么特殊的优化技巧，就足以完成
- 时空限制：时间：C++ 4s，Python 8s；内存512MiB。
- OJ评测机评测环境：
  - Ubuntu 22.04.3 LTS (in docker)
  - Intel i7-12700K (5.0GHz)
  - Python 3.11.8
  - GCC 13.2.0

实验具体说明

任务说明（必读）

- **本次实验中，你只需要完成两个函数**（在代码中已经使用**TODO注释**为你标记好）：
  - `Regex`类的`compile`函数。
  - `Regex`类的`match`函数。
- 你只需要完成上述的函数即可，不需要自己处理标准输入输出相关的问题。
  - 你的代码**不应在stdout中打印任何额外的输出**，如果确实需要打印，请**打印在stderr中**。
- 你所需要支持的**正则表达式的特性为`antlr/regex.g4`文件**中的且未被注释的所有部分。
  - 具体而言，主要包括以下内容：
    - 基本的正则表达式语法(`regex`、`expresstion`、`expressionItem`)
    - 三种限定词(`quantifier`)：`*` `+` `?`
    - 括号表达式(`group`)
    - 三种单字符匹配（单个字符`char`、字符类`characterClass`、中括号字符组`characterGroup`(`[]`)）。
  - 凡未在本次实验的regex.g4中出现的，或虽然出现但是被注释了的，都是本次实验不要求支持的内容。如rangeQuantifier(`{1,3}`)，anchor(`^`、`$`)等等。
  - 此外，你还需要支持正则表达式的**flags功能**，详见关于正则表达式的flags的具体说明。本次实验只要求支持一种flag：single line(`s`)。
  - 本次实验不需要捕获分组功能，你只需要按match函数的注释的要求，返回**仅有一个元素的数组**即可，无须返回分组内容。就算你返回了，评测器也会忽略不计。
- cases中的测试样例均为文本文件，内含正则表达式的pattern和输入字符串等。这些内容是人类可读的，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。
- 运行程序时，请将**单个测试样例文件**作为**唯一的参数**传入。
  - 具体的方法见对应语言的框架说明：C++框架说明 Python框架说明
  - 或者，若不传入任何参数，则程序将从stdin中读取输入。如果你懂得如何把测试样例通过stdin输入进去（如重定向`<`、`cat |`等），也可采取此方法。
- **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`add_executable(regex ...)`里）。

关于正则表达式

- 课上讲解的PPT中有比较详细的叙述，可从网络学堂查阅。
- 强烈推荐使用regexr.com。这是一个用于学习、构建和测试正则表达式的在线工具。
  - 它不但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，还会为你解释表达式里每个字符的含义，是学习正则表达式的利器。
  - 此外，恰当地使用该网站，还可能有助于你完成本次实验过程中进行debug。在你未来的学习工作中，也可以使用该网站进行正则表达式相关的调试。
- 此外，还可参考菜鸟教程的正则表达式 - 教程。

关于flags

正则表达式的flags是一种附加于正则表达式上的修饰符，可以用于改变正则表达式的行为。更多信息可以参阅这份文档。
本次实验中，暂时仅要求支持一种flag：single line(`s`)。

- 正常情况（无`s`flag的情况）下，表示任意字符的圆点`.`只能匹配**除了`\r``\n`外的所有字符**。但在有`s`flag时，`.`也同样**可以匹配`\r``\n`**，真正实现匹配任何字符。

在我们的实验中，正则表达式的flags会通过`compile`函数的`flags`参数传入。

- 如果`flags`参数的值是字符串`"s"`，说明该正则表达式具有`s`flag。当没有任何flags时，`flags`参数的值为空串`""`。
- 虽然本次实验只要求大家支持一种(`s`)flag，但实际上一个正则表达式可能同时具有多个flag。在下次实验中才会遇到这种情况。

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，本身就是由多个字符组成的。在我们要求的文法里，这样的例子只有非捕获分组`?:`一个（第三次实验才涉及）。
- 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。
        - 例如，终结符`EscapedChar`的定义是`EscapedChar : '\\' ~[0-9] | '\\x' [0-9a-fA-F][0-9a-fA-F];`，表示一个`EscapedChar`的构成有两种情况：
            - 由一个斜杠`\`，加上一个非数字的单个字符构成，共2个字符。例如`\n`。
            - 或是，由两个字符`\x`，加上两个数字或大小写A-F字符构成，共四个字符。例如`\x0a`。
- 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`ZeroOrMoreQuantifier: '*'; Digit : [0-9]; Char: . ;`。那么，一个字符`*`总会被解析为`ZeroOrMoreQuantifier`类型的token，而永远不会被解析为`Char`类型，尽管从`Char`的定义上看，任何单个字符都可以是`Char`；
    - 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+;
```

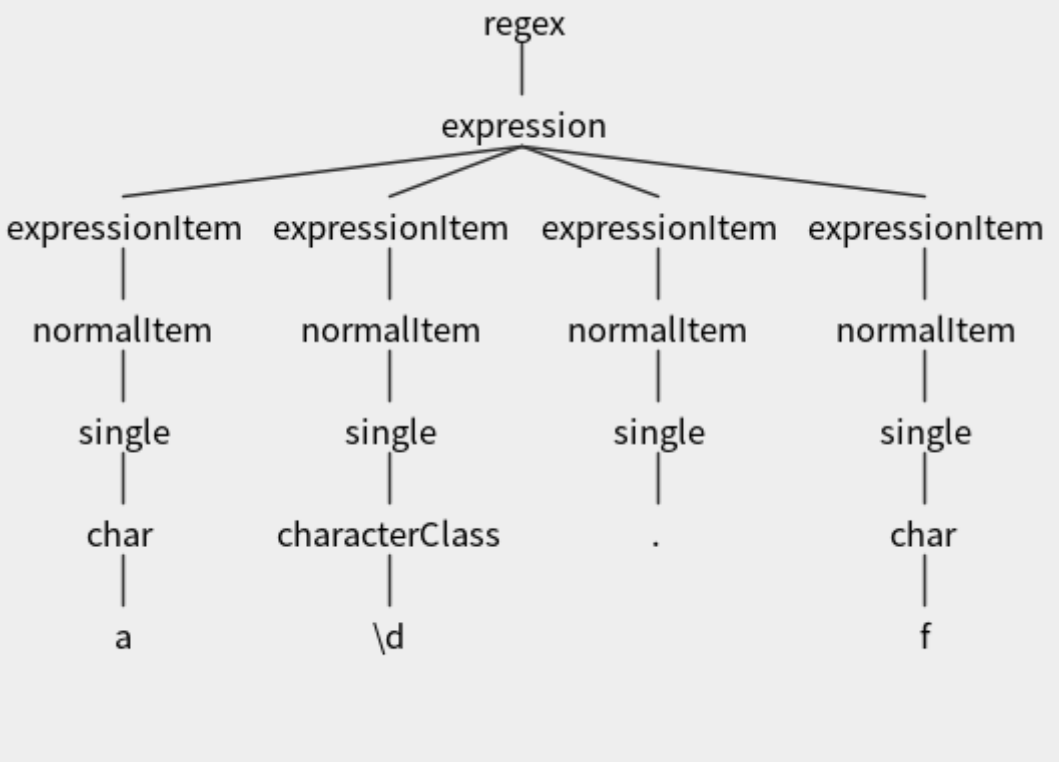
```
// 这个产生式表示一个expressionItems是由一个normalItem，加上一个可选的quantifier限定符(即*、+、?)构成的。
// 例如，`h+`expressionItem，其中`h`normalItem，`+`是quantifier`。
expressionItem : normalItem quantifier? ;

// 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
    : single // 能匹配一个字符的Item，包括普通的单字符、元字符、字符区间等
    | group // 括号分组
    ;

// 这是一个终结符的定义，表示ZeroOrMoreQuantifier的定义是单个字符`*`
ZeroOrMoreQuantifier : '*';

// 这是一个终结符的定义，表示EscapedChar的定义是一个字符`\`加上任意除了数字以外的字符
// 这里写成了`\\`，是因为antlr本身也有\n这种转义规则，因此匹配单个字符`\`的时候必须写成`\\`才可以被antlr理解)
EscapedChar : '\\' ~[0-9];

// 这是一个终结符的定义，表示Digit的定义是任意数字
Digit : [0-9];

// 这是一个终结符的定义，表示Char的定义是任意字符
// 由于优先级规则的存在，尽管Char被定义为了任意字符，但一个数字永远只会被看作Digit而不会被看作Char。
Char: . ;
```

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。安装方法建议参考网络学堂的课程文件中的“自动机绘制工具JFLAP”的相关文档。

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。
- 提示：TestRig中，对于叶子节点（终结符），**不会输出终结符的名字**，而是直接输出字符串的内容，如下图所示。

- 

语法分析树的访问

通过语法分析的过程（`Regex`类下的`parse`函数），得到的是语法分析树对象`RegexContext`。为了构造NFA，我们必须对这棵语法分析树进行遍历，在遍历的过程中进行分析，恰当地生成构造NFA所需的数据结构，或直接生成NFA。
遍历的方法可以采取：

- 直接自行访问语法分析树的每个节点。即直接使用ANTLR的相关API遍历树的各个节点，通常是自顶向下、从根节点起用DFS的方式遍历每个节点的子节点。
  - 具体的方法，请见下节语法分析树的API。
- 使用ANTLR内置的Listener机制进行遍历。这种方法需要你学习ANTLR的Listener机制。
  - 参考资料：ANTLR官方文档、一个使用Listener机制的例子。更多资料你也可以自行上网搜索。
  - 这种方法需要你额外学习一些东西，但生成的代码可能会稍微简单一些，建议感兴趣的同学自行学习和采用此方法。

语法分析树的API

下面以C++语言为例，介绍ANTLR关于语法分析树所提供的API。**Python语言的API名字与之完全相同，故不再分别赘述。**

首先需要明确，类的包含关系：

- `antlr4::tree::ParseTree`: 表示语法分析树上的一个节点。包括两个子类：
  - `antlr4::tree::TerminalNode`: 表示语法分析树的**叶子节点**，对应于文法的一个**终结符**。
  - `antlr4::RuleContext`: 表示语法分析树的**非叶子节点**的基类，对应于文法的一个**非终结符**。
    - `regex::XxxContext`，其中`Xxx`是文法中的非终结符的名字: 是**特定于某类型**的非终结符的**子类**，提供了比基类`antlr4::RuleContext`更丰富的方法，便于更方便地访问语法分析树。**将是大家在编程过程中最常用到的类**。

下面介绍，如何访问一个节点的子节点，这是在树上进行遍历的基础：（以下为了简便，我们省去类名前面的命名空间前缀）：

- 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。
  - 返回值是一个枚举：

```
enum class ParseTreeType : size_t {
  TERMINAL = 1, // 终结符
  ERROR = 2, // 错误节点，只有当输入的表达式无法被规约时才会产生。保证大家编程过程中不会遇到此类节点。
  RULE = 3, // 非终结符
};
```

  - 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::string 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

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，就立即返回接受的路径Path，而不在乎输入串是否已经读完。

转义字符

本实验中，所有的转义字符都会被解析成一个`EscapedChar`类型的终结符。它们实际上都对应于唯一一个字符，应当按照单个字符的方式去处理和构造NFA。
具体而言，分为三种情况：

- 按照C语言等语言的惯例，用特定的转义字符匹配特定的ASCII非打印字符。
  - 要求支持的此类字符共有5个：`\f` `\n` `\r` `\t` `\v`，其含义及对应ASCII码可参考 https://www.dotcpp.com/qa/17 。
- 用`\x`加两位16进制数的方式，表示任意ASCII字符。两位16进制数即是对应的ASCII码。
  - 例如`\x0a`，即对应ASCII码13(查表知其实就是`\n`)；`\x40`，即对应ASCII码64(查表知其实就是`@`)。
- 对某个字符在正则表达式中有特殊含义的情况，转义的内容就是斜杠`\`后面的内容。
  - 如`\(`匹配的是单个`(`，`\\`匹配的是单个`\`等。
  - 当你获得了一个`EscapedChar`类型的终结符，而经过判断又不是前面两种情况，则就是这种情况。

思路提示

- 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，可以参考课上的PPT讲解（可从网络学堂下载），和第四讲的课程内容。当然，更重要的是需要你自己的思考。
    - 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NFA，而父节点处的自动机就是把所有子树对应的NFA以某种方式组合一下即可。
      - 组合的方式是特定于产生式的类型的。
      - 例如，`expression`节点处的操作应是把每个`expressionItem`子树对应的NFA给“连接”起来，即从前一个`expressionItem`的终态通往下一个`expressionItem`的终态；
      - `regex`节点处的操作则应是把每个`expression`子树对应的NFA“或”起来，也就是新建一个新的初态，ε转移接到到每个子NFA的初态上。
      - `expressionItem`节点处的操作则应是对`normalItem`子树对应的NFA做一些“包装”。具体“包装”的方法，可以在第四讲课件中获得提示。

        - `normalItem`分为几种情况，其中`single`的情况，基本就对应着一个最简单的、由一个初态通过单个字符直接转移到一个终态的自动机了。
        - `single`向下分析时，可能会遇到一种情况：带有`characterGroupNegativeModifier`的`characterGroup`。这该如何处理呢？
            - 有很多种方法，但在NFA实现中新增一种转移规则的类型(RuleType)，可能是一种还不错的解决方案。
    - 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。
        - 比如，一个比较直接的方法就是实现“merge”函数，输入两个自动机，合并为一个自动机；
        - 或者注意到，上面函数的本质就是调整自动机的状态编号（包括转移规则中引用的dst状态编号），使得两者不会发生冲突。
            - 那么，有没有可能在生成最小的自动机时，就确保状态编号的全局唯一性呢？
        - 直接的“连接”/“或”之类的方法，可能会引入大量的冗余状态和ε转移。
            - 或许可以，对每个处理具体类型的节点的函数，并不是直接返回NFA类的对象，而是返回一些特制的数据结构；
            - 然后，在整棵树自上而下遍历完后，再通过一个函数，根据上述数据结构一次性生成最终的NFA类对象。
    - 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、依次执行挨个尝试的方法，也是能满足我们的要求的。

调试提示

- 你可以使用附带的`exportJFLAP`函数，来将NFA类的对象导出为JFLAP格式的文件，再用JFLAP打开。
    - 只需在合适的地方（推荐位置是Regex::compile函数的最后，因为这时刚刚完成完整NFA的构建）加入对该函数的调用：`exportJFLAP(nfa)`即可。更多用法详见函数注释。
    - **注意：** 该函数仅用于大家本地进行测试使用。**提交OJ的版本中切勿包含此函数的调用**，否则会造成OJ无法通过测试点（如PE 输出格式错误）！

C++语言框架说明

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。

此外，当你刚使用IDE打开项目，发现类似 **“找不到antlr-rumtime.h”、找不到ANTLR相关类**这种报错的时候，将项目**编译一遍**即可。编译的过程就包含了生成这些文件的过程，因此你只要编译一遍就可以找到这些文件了。

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../src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 . --target regex # 执行编译
```

执行文件的方法：（注意windows平台上是regex.exe）

```
./regex ../../cases/01.in # 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
# 或者，也可以不带参数执行
./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
```

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-regex.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`compile`、`match`等方法。

Python语言框架说明

运行方法

建议Python版本>=3.8。
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 ../cases/01.in # 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，也可以不带参数执行
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
```

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。