

AQL Subset Compiler

组员	学号
葛剑航	13331057
关宏朗	13331058
韩龙粤	13331067
杨振杰	13331311

项目目录

```
.
├── Readme.md
├── dataset
│   ├── Cplusplus.aql
│   ├── CplusplusInput
│   │   ├── cppfile1.input
│   │   └── cppfile2.input
│   ├── CplusplusOutput
│   │   ├── cppfile1.output
│   │   └── cppfile2.output
│   ├── PerLoc.aql
│   ├── PerLocInput
│   │   ├── A.input
│   │   └── PerLoc.input
│   ├── PerLocOutput
│   │   ├── A.output
│   │   └── PerLoc.output
│   ├── Revenue.aql
│   ├── RevenueInput
│   │   └── Revenue.input
│   └── RevenueOutput
│       └── Revenue.output
└── src
    ├── Lexer.cpp
    ├── Lexer.h
    ├── Makefile
    ├── Parser.cpp
    ├── Parser.h
    └── Tokenizer.cpp
```

```
|— Tokenizer.h  
|— View.cpp  
|— View.h  
|— main.cpp  
|— regex.cpp
```

8 directories, 25 files

代码运行环境

- Mac OS X 10.11

代码运行方法

1. 进入项目所在文件夹的 `src` 文件夹，执行 `make` 命令编译

```
$ cd <项目所在文件夹>/src  
$ make  
$ make cleanoutput
```

2. 运行 `main`，后面需要带3个参数：

- 指定输入的 `aql` 文件的路径
- `*.input` 文件的路径（或所在文件夹的路径）
- `*.output` 文件的路径（或指定输出的文件夹的路径）。

格式如下：

```
$ ./main <aql文件路径> <.input文件路径> <.output指定输出文件路径>
```

或者

```
$ ./main <aql文件路径> <.input文件夹路径> <.output输出文件夹路径>
```

举例如下：

```
$ ./main ../dataset/PerLoc.aql ../dataset/PerLocInput/ \  
  ../dataset/PerLocOutput/
```

即编译 dataset 文件夹中的 PerLoc.aql 文件，分析的文本是 dataset 文件夹中 PerLocInput 文件夹的所有 xxx.input 文件，每个文件对应的输出文件 xxx.output 放进 PerLocOutput 文件夹中。

3. 进入 dataset 文件夹以及相应的 xxxOutput 文件夹，即可看到相关的输出文件。

注意事项：

- 如果是路径参数是一个文件夹的话，最后必须加上 / 作为结尾；
- 由于文件输出是追加输出，所以在跑每个样例之前都要执行 make cleanoutput ；
- 相关运行语句如下（方便TA运行）

```
$ ./main ../dataset/PerLoc.aql ../dataset/PerLocInput/ \
  ../dataset/PerLocOutput/
$ ./main ../dataset/Revenue.aql ../dataset/RevenueInput/ \
  ../dataset/RevenueOutput/
$ ./main ../dataset/Cplusplus.aql ../dataset/CplusplusInput/ \
  ../dataset/CplusplusOutput/
```

部分思路分析

Lexer与Tokenizer

Lexer 和 Tokenizer 实现的方法类似。Tokenizer 将输入的 .input 文件分割成一个一个的 word 。Lexer 将 .aql 文件分割成一个个的 token 。遇到空格、\r、\t、\n 则作为分词的根据，另外遇到特殊符号也应该分词，其它单词、整数、正则表达式则作为一个整体。总的来说也是对字符串进行处理。处理之后得到的结果则储存在 vector 之后供后续使用。

Parser

语法分析器的基本思想是将词法分析器分析得到的 Token 向量逐个向后分析。每读取一个 Token ，则分析该 Token 的属性，然后根据语法产生式调用相关子函数进行处理，最后将处理后得到的结果返回。通过这种递归调用的方式，就能得到逻辑处理需要的参数，然后构建相应的 View 。

Select分支

每个 View 都有一个存放类 View_col 的向量，以表示存放view表格的每一列。而本部分对 select_stmt 分支的实现，本质上就是根据 select 的定义要求，构造一

个 `vector<View_col>`（存放新的view里面的所有列），返回给上一级的 `creat_stmt`，因而此分支的返回值是一个装有 `view` 列的向量。

实现过程：根据语法规则，可知 `select` 的操作对象来源于已经构造好的 `view` 表，而来源对象可以通过 `from_list` 里面的 `from_item` 来锁定，具体获取过程是利用本小组提供的接口，根据 `from_item` 里面的 ID 锁定来源对象 `view`，根据 `select_item` 里面的 ID 锁定对象 `view` 的相应列，锁定完成后进行对目标列的提取，从要求来看可能提取多列，返回这个提取结果即可（以 `vector<View_col>` 存放）。

Regex匹配分支

正则匹配分支中，主要是需要提取正则表达式，别名和列名、新 `View_column` 的命名、本名与别名。通过本名获取对应的 `View`（即 `Document`），列名获取对应的 `View_column`（即 `text`），然后调用正则引擎去分析输入文本。将得到的捕获块范围抽取出来，得到在原文中对应的子串，然后再组成 `Span`，在加入到以新 `View_column` 的命名创建的 `View_column` 之中，最后将其加入到 `View` 之中。

Pattern匹配分支

完成模式匹配的前提是先要建立一种结构体 `Block`，该结构体实例用于记录 `Pattern` 关键字后出现的每一个关键块。

能够匹配的关键块有5种，包括 `<ID.ID>`，`REG`，`(<ID.ID>)`，`(REG)` 和 `<Token>` `{NUM, NUM}`。其余情况如类似 `(<ID.ID> REG)` 这种在一个括号内包含多个关键块的则暂时不予考虑。

如此 `Group 1` 至 `Group N` 都是只有一个关键块描述的分组，`Group 0` 则是在 `Pattern` 关键字后出现的所有关键块组合而成的一个大关键块描述的分组。另外关键块中会有该关键块的组号（若被捕获）。

由于 `Group 0` 中的每个 `Span` 都是多个 `Span` 组合而成的大 `Span`，因此使用 `vector` 代表 `Group 0` 中的每一个 `Span`，而 `Group 0` 就是一个大 `Span` 的列表，其结构为 `vector< vector >`，接下来的主要算法是通过一个递归的过程逐步建立完整的 `Group 0`。

递归过程需要遍历关键块集合，过程如下：

1. 取出当前的关键块
2. 当前关键块为 `<Token>{NUM, NUM}` 时，取出 `NUM` 与 `NUM` 作为Token出现的最次数 `min` 与最高次数 `max` 后作为参数传递给下一次递归
3. 当前关键块为另外四种时，尝试对每个在当前 `Group 0` 中 `vector` 的最

后一个 `Span` 与当前关键块对应的 `Span` 列表（例如 `<P.Per>` 对应的 `Span` 列表）进行匹配：若 `min` 与 `max` 都不为 `0`，说明两个要匹配的 `Spans` 之间存在有 `<Token>{NUM, NUM}` 这种关键块，此时当且仅当前后两个 `Spans` 之间的 `Token` 数目（文档中的词数）在 `min` 与 `max` 的范围内进行匹配；

若 `min` 与 `max` 都为 `0`，则当且仅当前后两个 `Spans` 之间的 `Token` 数目（文档中的词数）为 `0` 时进行匹配。匹配时会先建立一个空的 `vector<vector>` 实例，成功进行一次匹配就会往一个空的 `vector` 实例中加入所有相应的 `Spans`，然后把这个 `vector` 实例添加到 `vector<vector>` 实例中，完成所有匹配后会将该 `vector<vector>` 实例、置 `0` 的 `min` 与 `max` 一同作为参数传递给下一次递归

4. 关键块集合遍历完全时结束递归，向上返回最终组合而成的 `Group 0`（一个 `vector<vector>` 实例）

完成递归过程获得 `Group 0` 后，遍历该 `vector<vector>` 对象，从中提取出 `Group 1` 至 `Group N` 的内容，然后在将 `Group 0` 每一个大 `Span` 重新整合成一个 `Span`，整合时通过判断组号是否为 `-1` 的方式将属于非捕获关键块的 `spans` 剔除。如何便可以提取出 `Group 0` 到 `Group N` 的内容，每一个 `Group` 都用一个 `vector<View_col>` 的对象表示，而这些 `Group` 可以用于构造需要的 `View`。

附加的数据集

实现了一个 `Cplusplus.aql`，用于分析提取大型 `C++` 文件中的类名、方法名、参数、返回值等信息。

在 `dataset/CplusplusInput` 提供了两个 `C++` 文件（`cppfile1.input` 和 `cppfile2.input`），提取后的结果列表放在 `dataset/CplusplusOutput` 中。

PS:

项目目录中的 `dataset` 中的每个 `xxxOutput` 的文件夹中已经放进了对应的处理后的结果 `xxx.output` 文件，可供TA参考。如果要实际运行，请先在 `src` 文件中执行 `make clean`，再重复之前的运行步骤。