

# 实验二: 语法分析器

张配天-2018202180

2021 年 5 月 15 日

## 目录

<b>1 实验内容</b>	<b>2</b>
<b>2 程序设计原理与方法</b>	<b>2</b>
2.1 原理 . . . . .	2
2.2 方法 . . . . .	2
<b>3 程序设计流程</b>	<b>5</b>
3.1 和词法分析器的串联 . . . . .	6
<b>4 程序设计清单</b>	<b>6</b>
<b>5 运行结果</b>	<b>6</b>
<b>6 程序使用说明</b>	<b>7</b>
<b>7 总结与完善</b>	<b>8</b>

## 1 实验内容

用 C++ 实现对给定文法的语法分析器。

## 2 程序设计原理与方法

### 2.1 原理

使用 LL(1) 分析器

- 对输入字符串进行词法分析, 将其解析为句子
- 根据给定文法求各个符号的 First, Follow 集
- 根据 First, Follow 集求每个文法的 Select 集
- 构造分析表
- 进行 LL(1) 分析

### 2.2 方法

明确了原理后, 参照课本和老师给的例子, 进行一步步实现。语法分析器的逻辑甚至比词法分析器还简单, 就是按照上述一步步做就可以了;

由于老师给的文法较为简单, 我认为更加挑战的在于如何写出一个优雅的语法分析器, 我使用类来实现, 目的有两个:

1. 更好地抽象出**符号, 文法, 分析器**的概念, 方便自己的理解, 以及后续能够方便地引入到别的文件中;
2. 尽可能少地使用空间, **多使用指针**;

具体来说, 我分别定义了如下 5 个类:

```
1 // 符号类
2 class Symbol{
3     public:
4         char symbol;           // 具体符号
5         int index;             // 符号的序号, 对应在最
                                // 终的分析表中的行号/列号
6         bool produce_e;        // 是否能产生 epsilon
7         bool is_terminal;      // 是否是终结符
8
9         Symbol();
10        Symbol(char, int, bool);
```

```
11
12     set<Symbol*> first;           // First集
13     set<Symbol*> follow;         // Follow集
14
15     int append_first(Symbol*,int);
16     int append_follow(Symbol*, int);
17
18     void printFirst();
19     void printFollow();
20 };
21
22 // 符号集
23 class Symbols{
24     public:
25         Symbols();
26         int len();
27         void info();               // 打印符号集中所有符号
                                   // 的相关信息
28
29         void append(Symbol*);      // 给符号集添加新的符号
30         Symbol* find(char);        // 根据符号的值查找某一
                                   // 个符号
31
32         map<char,Symbol*> symbols; // 用字典保存符号集中符
                                   // 号的指针
33 };
34
35 // 文法类
36 class Grammer{
37     public:
38         Grammer();
39         Symbol* head;              // 文法的左式，直接对应
                                   // 到相应的符号
40         string tail;               // 文法的右式，由于不止
                                   // 一个符号，因此还是用字符串保存符号的值
41
42         set<Symbol*> select;        // 文法的select集，同样
                                   // 是符号的指针
43
```

```

44         void printSelect();
45     };
46
47     // 文法集类
48     class Grammers{
49     public:
50         Grammers();
51         Grammers(int);           // 创建给定个数的新
                                   文法
52         int len();               // 有效的文法长度
53         void info();
54
55         Grammer& operator[](int); // 重载[]使得该操作
                                   符能够直接访问其grammer成员
56
57         void getGrammer(istream&, Symbols&); // 读入并加载文法,
                                   同时加载非终结符到Symbols中
58         bool _produceE(char);       // 判断某一个符号能
                                   否导出epsilon的子函数
59         void getProduceE(Symbols&); // 递归判断每一个符
                                   号能否导出epsilon
60         void getSelect(Symbols&,Symbols&); // 根据first集和
                                   follow集计算select集, 存在各个文法中
61
62         Grammer * grammers;       // 用动态数组保存文
                                   法
63
64     private:
65         int length;
66     };
67
68     // 分析器类
69     class Parser{
70     public:
71         Parser(Grammers&,Symbols&,Symbols&,Symbols&); // 构造函数
                                   , 将各个成员赋值
72         void getSymbols();         // 根据读取
                                   的文法, 更新终结符
73         void initialize(const string&); // 进行一系

```

```

    列初始化操作，包括构建三个符号集，读取文法等
74     void getFirst(); // 计算每一
        个符号的first集
75     void getFollow(); // 计算每一
        个符号的follow集
76     void getTable(); // 根据
        select集，构建分析表
77     void parse(string,string); // 从给定输
        入文件读取输入并分词，然后写入给定输出文件中，之后进行语
        法分析
78     void error(); // 出错的响
        应函数
79
80     Grammer *** table; // 语法分析
        表，表的单元是语法类的指针
81 private:
82     Grammers grammers;
83     Symbols symbols;
84     Symbols terminals;
85     Symbols nterminals;
86
87     int getLex(vector<deque<Symbol*>>&, string&, string&); //
        进行词法分析的函数
88 };

```

### 3 程序设计流程

根据上一个 section, 在完善了各个类的定义后, 整个语法分析的流程被极大地简化, 我在主程序中只需要构造符号, 文法, 和分析器的实例, 然后调用两个分析器的方法即可完成语法分析:

```
1  #include "parser.cpp"
2  #include "grammer.cpp"
3  #include "symbol.cpp"
4
5  int main(){
6      Grammers grammers(GRAMNUM);
7      Symbols symbols, terminals, nterminals;
8
9      Parser parser(grammers,symbols,terminals,nterminals);
10     parser.initialize("data/grammers.txt");
11     parser.parse("data/test.in");
12     return 0;
13 }
```

### 3.1 和词法分析器的串联

需要注意的是, 实验中让设计的语法分析器是对给定的四则运算的, 那个 `id` 根据我的理解就代表着 *identifier*, 也就是任何整数, 小数, 变量; 而将输入的字符串解析为有意义的符号集, 即句型, 是靠词法分析器的, 因此我将实验一的词法分析器整合到实验二中, 对每一个输入字符串, 都是先词法分析, 再读取中间文件, 然后语法分析。

这里不得不说, 老师给的文法还是相当简单的, 如果要完成整个 c- 的文法, 其实应该会更复杂。

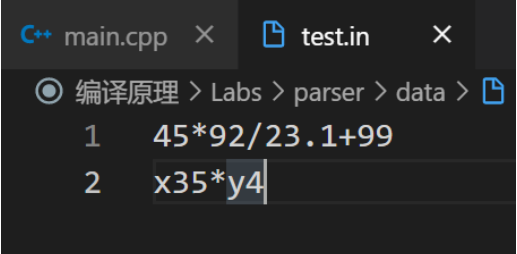
## 4 程序设计清单

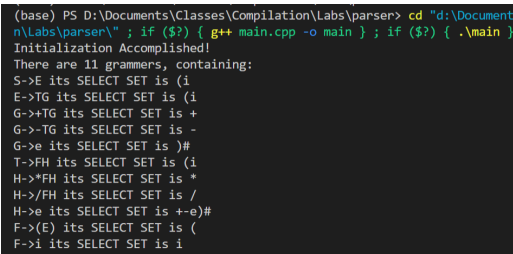
如上, 我们要设计实现五个类, 其相应的成员函数, 同时要把词法分析器的内容做一些修改, 同时, 实验一我图懒省事, 所有代码放在一个 .cpp 文件中, 而且 `lexer` 也不是一个类, 使用函数封装的, 在整合时我就感受到了很多不方便之处, 这也是我在 `parser` 使用类的一个重要原因。

更多地, 在本实验中, 我将代码组织为经典的 .h 声明, .cpp 定义的结构, 最后在 `main.cpp` 中运行, 很优雅。

## 5 运行结果

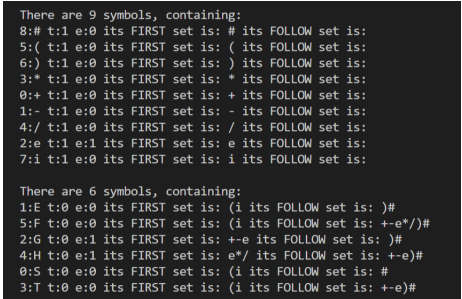
将多个运算表达式写入 `test.in`, 其中包含数字, 小数和 `identifier`, 进行语法分析, 得到结果部分展示如图 1a ~ 图 1f:

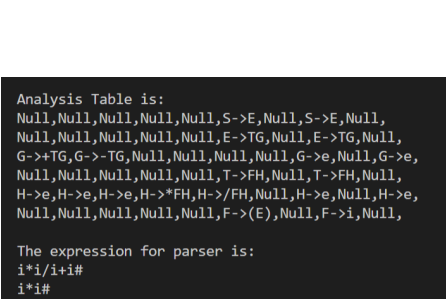




(a) input

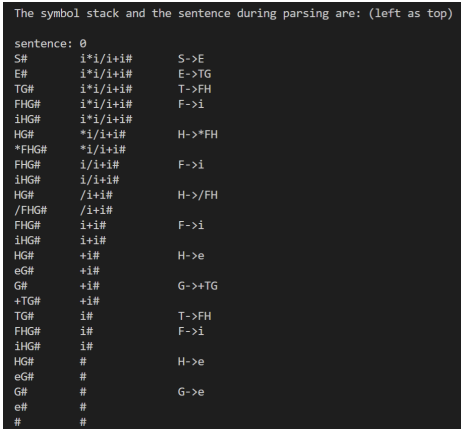
(b) grammers

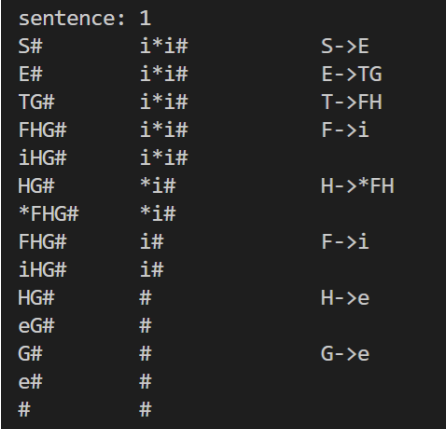




(c) symbols

(d) analysis table





(e) result for the first sentence

(f) result for the second sentence

图 1: 运行结果

## 6 程序使用说明

- 在 data/test.in 中修改要解析的字符串；
- 在 data/grammers.txt 中修改给定的文法；
- 在 data/KeyWords.txt 中修改预设关键词；
- 在 data/Operators.txt 中修改预设操作符；
- 在 data/Separators.txt 中修改预设分隔符；

- 语法分析的结果默认会在终端显示；

## 7 总结与完善

通过这次实验, 回忆了 c++ 的类写法, 从头到尾搞明白了 LL(1) 的分析逻辑, 自己完成了各项算法的设计; 同时也遗留了几个问题:

- 在求某一个符号能否产生  $\epsilon$  时使用递归, 但是递归会将部分符号计算多次, 可以使用**动态规划**优化;
- 在求 First, Follow 集时使用迭代的逻辑, 求 First 要迭代 4 次, 求 Follow 要迭代 2 次, 其实有很多冗余操作, 可以考虑使用递归;