实验三: SLR 分析器

张配天-2018202180

2021 年 5 月 30 日

目录

1 实验内容 **2**

2 程序设计原理与方法 **2**

2.1 原理 2

2.2 方法 2

3 程序设计流程 **7**

3.1 和词法分析器的串联 8

4 程序设计清单 **8**

5 运行结果 **8**

6 程序使用说明 **8**

7 总结与完善 **10**

1 实验内容

用 C++ 实现对给定文法的 SLR 语法分析器。

2 程序设计原理与方法

2.1 原理

使用 **SLR 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
- 根据给定文法创建一系列项目
- 求项目的闭包 Closure 以及转移函数 Go
- 构造 SLR 分析表
- 进行 LR 分析

2.2 方法

明确了原理后, 参照课本, 进行一步步实现。不得不说, SLR 分析器比 LL(1) 难写太多了; 和上一个实验一样, 我使用**类**来实现, 目的有三个:

1. **最少化所需的对应关系表的数量**。比如用 map 存某个 symbol 对应的 first 集这种, 统统不需要;
2. **最大化程序的灵活程度**。由于所有操作对象都是指针, 我们可以轻松地完成对元素的访问, 比如通过文法类访问其左端的符号, 再比如通过项目类访问对应的文法, 通过闭包访问其中某个项目的某个文法的某个右端符号的序号等;
3. **最小化占用的空间**;
4. **更好地抽象出符号, 文法, 闭包, 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;

具体来说, 我分别定义了如下 8 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 is_terminal;                    // 是否是终结符
7
8         Symbol();
9         Symbol(char,int,bool);
10
11        set<Symbol*> first;                  // First集
12        set<Symbol*> follow;                 // Follow集
13
14        int append_first(Symbol*,int);
15        int append_follow(Symbol*, int);
16
17        void printFirst();
18        void printFollow();
19    };
20
21    // 符号集
22    class Symbols{
23        public:
24            Symbols();
25            void info();                     // 打印符号集中所有符号
                  的相关信息
26
27            void append(Symbol*);            // 给符号集添加新的符号
28            Symbol* find(char);              // 根据符号的值查找某一
                  个符号
29
30            set<Symbol*> symbols;            // 用集合保存符号集中符
                  号的指针
31
32            int length;                      // 符号个数
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            vector<Symbol*> tail;            // 文法的右式，用vector
```

```
            按顺序保存右端的所有符号
41
42          void info();                            // 输出文法信息
43  };
44
45  // 项目类
46  class Item{
47      public:
48          Item();
49          Item(Grammer*, int, int);
50
51          Symbol* get_dot();                      // 获取.之后的那个符号
                的指针
52          Symbol* get_head();                     // 获取项目头符号的指针
53
54          void info();
55
56          Grammer* grammer;                       // 保存该项目对应的文法
                的指针
57          Item* next;                             // 保存下一个项目的指针
58
59          int dot;                                // .的位置
60          int index;                              // 项目序号
61          int length;                             // 项目长度(对应文法的
                右部的长度)
62  };
63
64  // 项目集类(闭包类)
65  class Items{
66      public:
67          Items();
68          Items(int);
69
70          Items* find_route(char);                // 根据路由表查询某一个
                字符在DFA上对应的下一个闭包
71
72          bool append(Item *, bool=false);        // 往项目集中添加新项目
73          void info();
74          vector<Item*> items;                    // 用动态数组保存项目
```

```
75          set<int> indices;                    // 维护一个项目的id的集
                合，用来判断某两个项目集是否相等
76          map<char, Items*> route;             // 路由表
77          int index;                           // 闭包序号(状态号)
78  };
79
80  // 闭包集类
81  class Closures{
82      public:
83          Closures();
84          Closures(int);
85          vector<Items*> closures;             //用动态数组保存闭包的
                指针
86
87          Items* find(Items*);                 // 在数组中查询某一个闭
                包
88          void get_closure_and_go(map<char, vector<Items*>>&, map<
                char, Symbol*>&);                // 计算所有闭包和对应的
                go函数
89          void info();
90
91          int length;                          // 闭包的个数
92  };
93
94  // 文法集类
95  class Grammers{
96      public:
97          Grammers();
98          Grammers(int);                           // 创建给定个数的新
                文法
99          int len();                               // 有效的文法长度
100         void info();
101
102         Grammer& operator[](int);                // 重载[]使得该操作
                符能够直接访问其grammer成员
103         vector<Items*>& operator[](char);        // 重载[]使得能直接
                访问其items成员
104
105         void get_grammer(ifstream&, map<char, Symbol*>&, Symbols&);
```

```
                // 读入并加载文法，为所有独特的符号创建一个实例
106
107         vector<Grammer*> grammers;              // 用动态数组保存文
                法
108         map<char, vector<Items*>> items;        // 用map保存某一个
                符号开头的项目集的数组
109
110         int length;
111 };
112
113 // 分析器类
114 class Parser{
115     public:
116         Parser(Grammers&,Symbols&,Symbols&,Symbols&);   // 构造函数
                ，将各个成员赋值
117         void get_symbols();                              // 根据读
                取的文法，创建两个符号集，对应终结符和非终结符
118         void initialize(const string&);                  // 进行一系
                列初始化操作，包括构建符号集，加载文法，计算闭包等
119         void get_first();                                // 计算每
                一个符号的first集
120         void get_follow();                               // 计算每
                一个符号的follow集
121         void get_table();                                // 根据闭
                包和Go，构建分析表
122         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
123         void error();                                    // 出错的响
                应函数
124
125         Grammer *** action;                              // action表
                ，表的单元是语法类的指针
126         int ** goTo;                                     // goto表，
                表的单元是状态序号
127     private:
128         Grammers grammers;
129         map<char, Symbol*> symbols;                      // 总的单词
                表
```

```
130         Symbols terminals;
131         Symbols nterminals;
132 
133         Closures closures;                                  // 所有闭包
134 
135         int get_lex(vector<deque<Symbol*>>&, string&, string&);  //
                进行词法分析的函数
136         int _id(char);
137 };
```

3 程序设计流程

根据上一个 section, 在完善了各个类的定义后, 我需要说明一下代码的逻辑:

- 文法集类实例 parser.grammers 调用 grammers.get_grammer 读取老师给的文法, 在此函数中完成两件事:
  - 为所有 symbol 创建实例, 将指针保存在分析器实例的属性 parser.symbols 中
  - 为所有文法构建对应的项目实例, 注意**项目实例不需要保存文法的头尾, 只需要保留一个文法的指针, 然后用一个 int 来指明. 的位置即可**。之后, 从同一个文法中构建的项目形成一个**项目集**, 进一步, 以同一个非终结符开头的所有项目集被保存在一个 vector 中, 存在文法集实例的属性 items 中。由于是按顺序保存各个项目, 所以每个项目集的第一个都是.XX, 这有助于我们之后构建闭包;
- 分析器调用 parser.get_symbol 将 parser.symbols 中的符号划分为 terminals 和 nterminals, 然后调用 parser.get_first, parser.get_follow 分别计算每个符号的 first 集和 follow 集;
- 闭包集类实例 parser.closures 调用 closures.get_closure_and_go 计算 closure 集和 go 函数, 这里主要采用两个方法:
  - 通过**维护一个 set** 来求某一个 item 的闭包;
  - 通过**广度优先搜索**来计算所有 go 函数;
- 分析器调用 parser.get_table 构建 action 表和 goto 表;
- 分析器调用 parser.parse() 进行分析。

由上, 整个语法分析的流程被极大地简化, 我在主程序中只需要实例化分析器, 然后调用两个分析器的方法即可完成语法分析。值得一提的是, 由于使用了私有变量保存符号类等, 类外是无法干扰编译程序运行的。

编译原理 > Labs > parser_LR > main.cpp > main()

```cpp
#include "parser.cpp"

int main(){
    Parser parser;
    parser.initialize("data/grammers.txt");
    parser.parse("data/test.in");
    return 0;
}
```

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。

除此之外, 我的分析器支持**多行分析**。

4 程序设计清单

如上, 我们要设计实现八个类, 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。

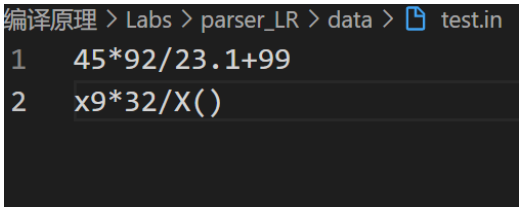
更多地, 在本实验中, 我将代码组织为经典的.h 声明, .cpp 定义的结构, 最后在 main.cpp 中运行, 很优雅。

5 运行结果

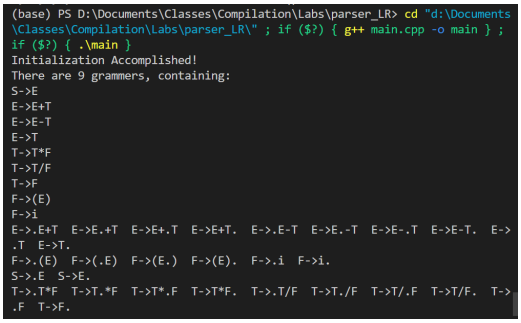
将多个运算表达式写入 test.in, 其中包含数字, 小数和 identifier, 进行语法分析, 得到结果部分展示如图 1a ∼ 图**??** :

6 程序使用说明

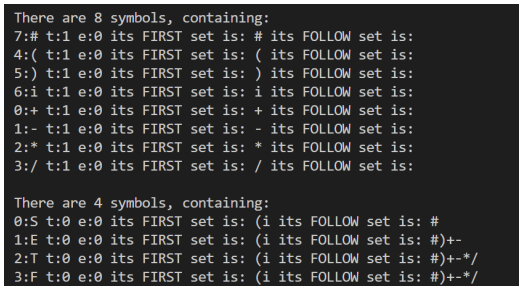
- 在 data/test.in 中修改要解析的字符串;
- 在 data/grammers.txt 中修改给定的文法;



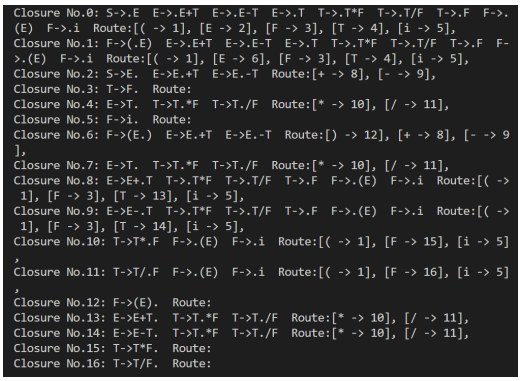
(a) input



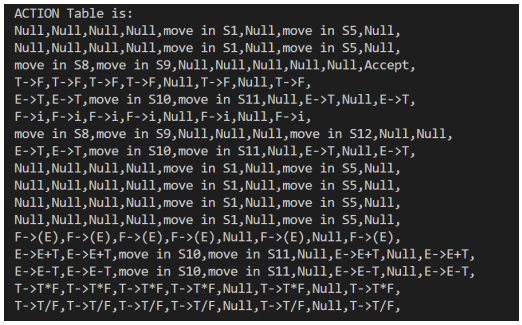
(b) grammers and items



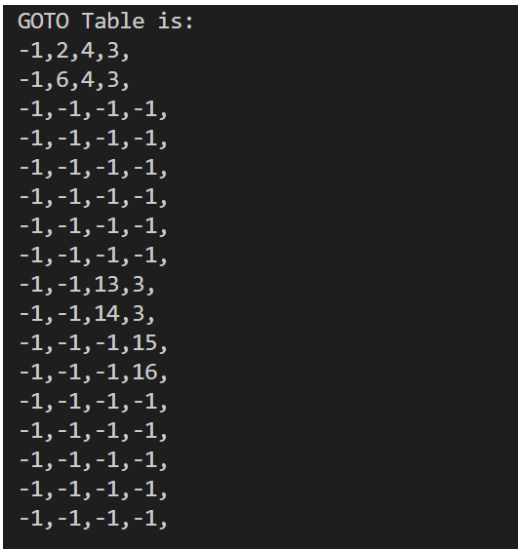
(c) symbols



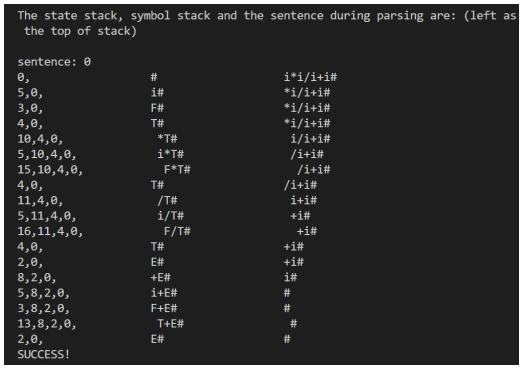
(d) closures



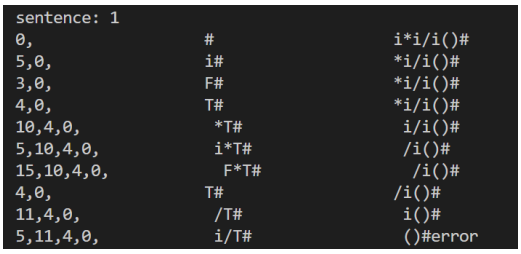
(e) action table



(f) closures



(g) result for the first sentence



(h) result for the second sentence

图 1: 运行结果

- 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 SLR 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 Closure 时应该可以更快, 而且感觉我一直不是很擅长写“到不增大为止”这种条件的代码。