北京郵電大學

Beijing University of Posts and Telecommunications

编译原理语法分析实验报告

[C++&fLex]

学院：计算机学院
2020211376 马天成
2022 年 11 月 18 日

北京邮电大学《编译原理》课程实验报告

<table>
<tr><td>实验
名称</td><td colspan="2">语法分析</td><td>学
院</td><td>计算机</td><td>指导教师</td><td>刘辰</td></tr>
<tr><td>班 级</td><td>班内序
号</td><td colspan="2">学 号</td><td>学生姓名</td><td colspan="2">成绩</td></tr>
<tr><td>2020211305</td><td>2</td><td colspan="2">2020211376</td><td>马天成</td><td colspan="2"></td></tr>
<tr><td>实
验
内
容</td><td colspan="6"></td></tr>
<tr><td>学生
实验
报告</td><td colspan="6">(详见“实验报告和源程序”册)</td></tr>
<tr><td>课
程
设
计
成
绩
评
定</td><td colspan="6">评语:

成绩:

指导教师签名:

年 月 日</td></tr>
</table>

注：评语要体现每个学生的工作情况，可以加页。

目录

1. 实验目的和要求

1.1 实验目的

语法分析程序的设计与实现

1.2 实验要求

编写语法分析程序，实现对算术表达式的语法分析。要求所分析的算术表达式 由如下的文法产生： 在本实验中，我们需要用 LL(1)和 LR 两种方法来对输入串进行分析 对于 LL(1)方法，要求如下：

（1）为给定文法自动构造预测分析表

（2）构造 LL(1)预测分析程序 对 LR 方法，要求如下：

（1）构造识别该文法所有活前缀的 DFA

（2）构造该文法的 LR 分析表

（3）构造 LR 分析程序

构造 YACC 程序自动生成文法分析程序

2. LL1 程序

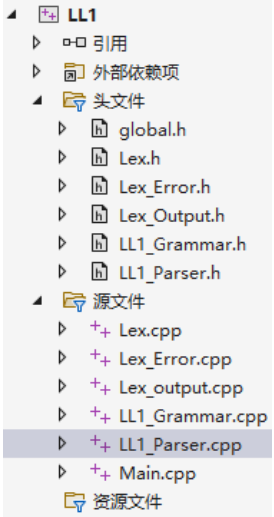
2.1 在词法分析程序上扩展程序

首先是词法分析。词法分析的结果是把输入文件变为符号和符号类型以及坐标信息的元组。所以我们直接在 main 函数里进行词法分析后，再进行语法分析。

词法分析这里就不再过多赘述。

但词法分析结果一开始是打算读取输出文件的，但是输出文件是系统调用，后面执行的时候还没有对文件进行修改，所以我们只能用全局变量来解决。

2.2 模块设计



lex, output, error 有对应.h&.c

1. global.h：规定数据类型
2. Main.c：主函数，运行
3. Lex：词法分析模块
4. Lex_Output：输出正确词法模块
5. Lex_Error：输出错误词法模块
6. LL1_Grammar：分析文法，输出分析表
7. LL1_Parser：分析输入表达式，根据分析表构造分析过程

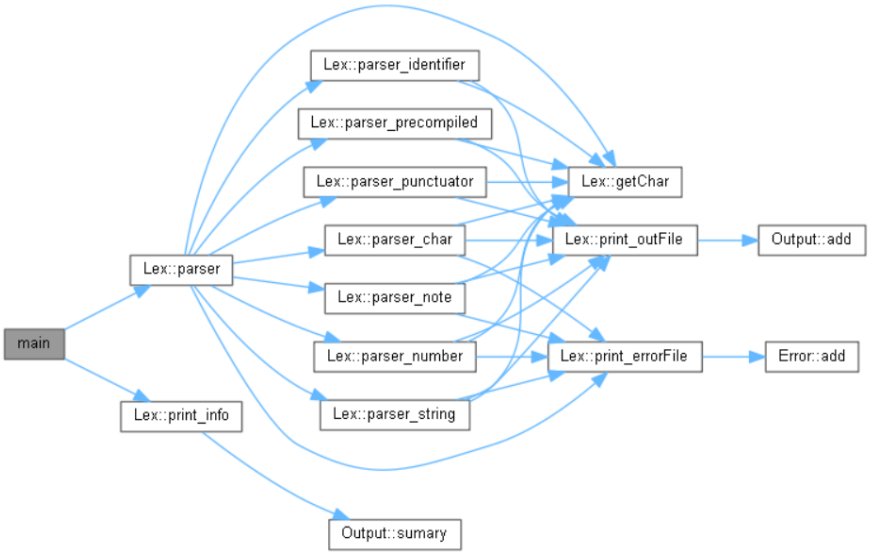
2.3 Main 函数

Main 函数的主要功能：

```cpp
#include <iostream>
#include "lex.h"
#include "LL1_Parser.h"
#include <string>

using namespace std;

vector<pair<string, string>> lex_outcome;

int main(int argc, char const* argv[])
{
    // 词法分析部分
    if (argc != 5) {
        cout << "Usage: LL1 [infile_name] [outfile_name] [errorfile_name] [gramfile_name]" << endl;
        exit(1);
    }

    Lex myLex(argv[1], argv[2], argv[3]);
    myLex.parser();     // 执行语法分析
    myLex.print_info(); // 输出结果再terminal和outFile

    // 语法分析部分
    Parser myParser(argv[4]);

    return 0;
}
```

1. 获得输入参数，分别为[infile_name] [outfile_name] [errorfile_name] [grammarfile_name] 若格式不正确则报错 exit(1)。
2. 运行 lex 类分析相应的 inFile。
3. 用 lex 统计输出的函数输出有关的所有统计信息（同时打印在 out 和 terminal 里）。
4. 用 Parser 来分析程序（Parser 内包含 Grammar 类进行文法分析表的构造，然后进行表达式分析）

生成的拓扑图：



2.4 Lex-识别进入各类词法函数

Lex 就不进行详细介绍了。

2.5 Grammar 类详解

```
13  class Grammar {
14  public:
15      // 待分析的生成式集合
16      map<string, vector<string>> expressions;
17      // 终结符集合
18      set<string> term;
19      // 非终结符集合
20      set<string> not_term;
21      // 起始符号
22      string start;
23
24      // 非终结符的First集
25      map<string, set<string>> First;
26      // 非终结符的Follow集
27      map<string, set<string>> Follow;
28
29      // 预测分析表
30      map<vector<string>, string> Table;
31
32      Grammar(string file_name);
33      Grammar() {};
34
35      // 分割字符串
36      vector<string> split(const string& str, const string& splits);
37
38  private:
39      // 文件读出非终结符，终结符，开始符号，表达式
40      void init(string file_name);
41
42      // 消除左公因子
43      void Eliminate_left_common_factor();
44      // 消除左递归
45      void Eliminate_left_recur();
46      // 生成fisrt集
47      void create_first();
48      // 找first
49      set<string> find_first(string symbol);
50      // 生成follow集
51      void create_follow();
52      // 找follow
53      set<string> find_follow(string symbol);
54      set<string> find_first_list(string symbol_list);
55
56      // first和follow 计算预测分析表
57      void predict_table();
58      void print_predict_table();
59      void print();
60
61      // 是终结符
62      bool is_terminal(string str);
63      bool is_not_terminal(string str);
64  };
65
66  #endif
```

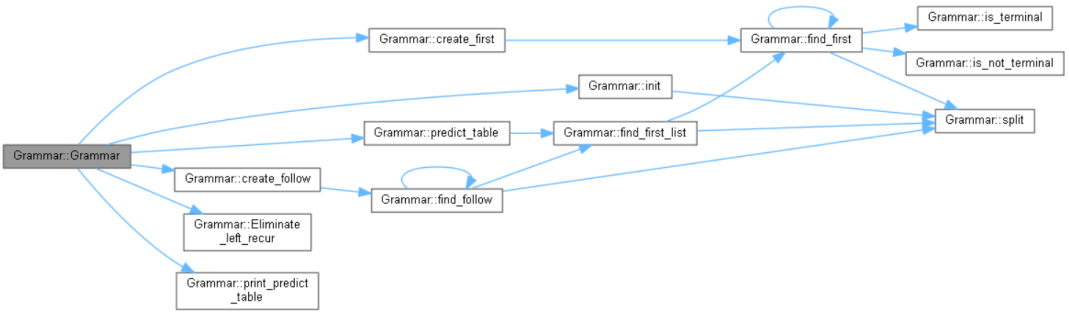
我觉得还是主要重点分析数据类型、我们用 **set 存符号，用 map 存 firstfollow，用 map 存预测分析表**。

尤其是预测分析表的构造。我们用**前一个 vector 所确定非终结符遇到的终结符**，来寻找到对应的须要调用的产生式。这是一个非常巧妙的做法。

根据输入语法来进行语法分析表的生成。其实主要的步骤就是这些：

```
Parser::Parser(string grammar_file) {
    // 初始化语法分析表
    G = Grammar(grammar_file);

    // 栈放入$符号
    Symbol.push_back(END);
    // 栈放入起始符号
    Symbol.push_back(G.start);

    // 输入暂存符号为$
    input.push_back(make_pair("PUNCTUATOR", END));
    // 将词法分析的文件结果转化到输入中
    Lex_Input();

    // 参照Grammar对栈和输入进行分析
    Analysis();
}
```

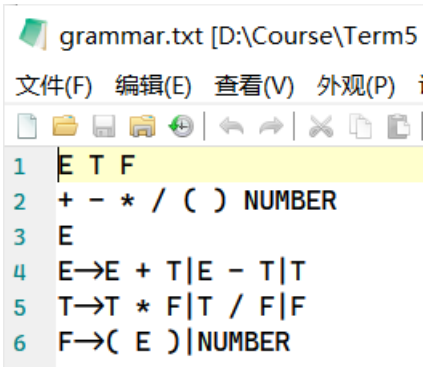
函数调用如下：



2.5.1 Grammar-获得文法输入信息

这里规定文法的输入是这样的：

- 第一行是非终结符
- 第二行是终结符
- 第三行是起始符
- 后面是文法表达式，但是文法生成式各符号间一定是空格隔开（正则匹配太离谱了）




```
void Grammar::init(string file_name) {
    ifstream file;
    file.open(file_name);
    if (!file.is_open()) {
        cout << "Fail to open grammar file!" << endl;
        return;
    }

    string line;

    // get not terminals
    getline(file, line);
    vector<string> non_t = split(line, " ");
    for (string item : non_t)
        not_term.insert(item);

    // get not terminals
    getline(file, line);
    vector<string> t = split(line, " ");
    for (string item : t)
        term.insert(item);
    term.insert(END);

    // get start symbol
    getline(file, line);
    start = line;

    // get grammar
    while (getline(file, line) && line != "") {
        vector<string> infos = split(line, "->");
        vector<string> info = split(infos[1], "|");
        expressions.insert(make_pair(infos[0], info));
    }
}
```

2.5.2 Grammar-消除左公因子和左递归

```
void Grammar::Eliminate_left_common_factor() {
}
```

消除左递归的算法如下所示：

$$A \to A\alpha_1|\dots A\alpha_n|\beta_1\dots|\beta_n$$

转化为

$$A \to \beta_1 A'|\dots|\beta_n A'$$
$$A' \to \alpha_1 A'|\dots|\alpha_n A'|\epsilon$$

```
void Grammar::Eliminate_left_recur() {
    // 遍历每行的表达式
    int flag = 0;
    for (auto item : expressions) {
        flag = 0;

        // 遍历右边的生成式
        for (string g : item.second) {
            // 左边的符号等于右边的首符，存在左递归
            if (g.substr(0, item.first.length()) == item.first) {
                flag = 1;
                break;
            }
        }

        // 当前表达式需要处理，后面添加 #
        vector<string> new_production;   // 原符号新表达式
        vector<string> new_symbol_production; // 新符号新表达式
        if (flag) {
            not_term.insert(item.first + "#");
            for (auto g : item.second) {
                string new_generative = "";
                if (g.substr(0, item.first.length()) == item.first) {
                    new_generative = g.substr(item.first.length()) + " " + item.first + "#";
                    new_symbol_production.push_back(new_generative);
                }
                else{
                    new_generative = g + " " + item.first + "#";
                    new_production.push_back(new_generative);
                }
            }
            // 加入空串
            new_symbol_production.push_back(EPSILON);
            expressions[item.first] = new_production;
            expressions.insert(make_pair(item.first + "#", new_symbol_production));
        }
    }
}
```

实质就是找到左公因子，提取；找到左递归，加入新符号，生成文法表达式。

我们这里新符号就是在原符号后面加入#（如 E 变成 E#）

2.5.3 Grammar-寻找 First 和 Follow

这里就是按照书上给的递归算法来实现了。我们这里写了对于每一个生成式的递归调用。

```
void Grammar::create_first() {
    // 对于所有的非终结符产生式
    for (auto item : expressions)
        find_first(item.first);

    // print First
    cout << "\nFirst:\n";
    for (auto item : First) {
        cout << item.first << "\n";
        for (auto f : item.second)
            cout << "\t" << f;
        cout << "\n";
    }
}
```

```
void Grammar::create_follow() {
    for (auto item : expressions)
        set<string> temp = find_follow(item.first);

    // print Follow
    cout << "\nFollow:\n";
    for (auto item : Follow) {
        cout << item.first << "\n";
        for (auto f : item.second)
            cout << "\t" << f;
        cout << "\n";
    }
}
```

```
set<string> Grammar::find_first(string symbol) {
    set<string> ans;

    // $符
    if (symbol == EPSILON)
        ans.insert(EPSILON);
    // 终结符
    else if (is_terminal(symbol))
        ans.insert(symbol);
    // 非终结符
    else if (is_not_terminal(symbol)){ ... }

    return ans;
}
```

```
set<string> Grammar::find_follow(string symbol) {
    set<string> ans;

    // 起始符号
    if (symbol == start)
        ans.insert(END);
    // 遍历所有产生式，symbol是follow，不用再找
    if (Follow.find(symbol) != Follow.end()) {
        ans = Follow[symbol];
    }
    // 需要计算follow
    else{ ... }
    return ans;
}
```

```
// 对字符序列求first
set<string> Grammar::find_first_list(string symbol_list) {
    set<string> ans;
    int flag_e = 1;
    auto outcome = split(symbol_list, " ");

    for (auto tmp_symbol : outcome)
    {
        if (tmp_symbol == "")
            continue;
        set<string> temp = find_first(tmp_symbol);
        set<string> swap;

        if (temp.find(EPSILON) == temp.end()) {
            set_union(ans.begin(), ans.end(), temp.begin(), temp.end(), insert_iterator<set<string> >(swap, swap.begin()));
            ans = swap;
            flag_e = 0;
            break;
        }
        else {
            /*cout << "erase" << endl;*/
            set_union(ans.begin(), ans.end(), temp.begin(), temp.end(), insert_iterator<set<string> >(swap, swap.begin()));
            ans = swap;
            ans.erase(EPSILON);
        }
    }
    if (flag_e == 1)
        ans.insert(EPSILON);

    return ans;
}
```

这里要注意的是，first 的寻找多了一个函数，是为了计算符号串的 first。

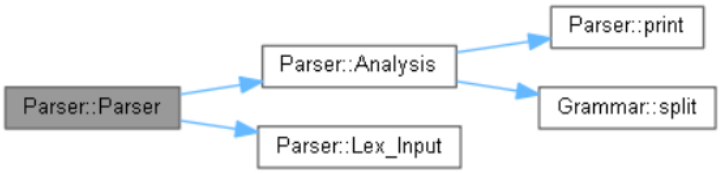
2.5.4 Grammar-构造文法分析表

```
void Grammar::predict_table() {
    for (auto item : expressions) {
        for (auto exp : item.second) {
            set<string> first_symbol = find_first_list(exp);
            for (auto str : first_symbol) {
                if (str != EPSILON) {
                    vector<string> temp;
                    temp.push_back(item.first);
                    temp.push_back(str);

                    Table.insert(make_pair(temp, exp));
                }
                else {
                    // 放置左侧符号的follow集
                    for (auto f_str : Follow[item.first]) {
                        vector<string> temp;
                        temp.push_back(item.first);
                        temp.push_back(f_str);

                        Table.insert(make_pair(temp, exp));
                    }
                }
            }
        }
    }

    for (auto item : expressions) {
        for (auto follow : Follow[item.first]) {
            vector<string> temp;
            temp.push_back(item.first);
            temp.push_back(follow);

            if (Table.find(temp) == Table.end())
                Table[temp] = "Synch";
        }
    }
}
```

我们**预测分析表的构造**还是挺明确的。就是根据题目列出的算法来实现。这也是**本次实验的重点之一。**

在这个中间，我们也加入了错误处理。但很遗憾没有太多时间进行错误测试。

2.6 Parser 类详解




```
class Parser {
private:
    Grammar G;

    // 符号栈
    vector<string> Symbol;
    // 输入
    vector<pair<string, string>> input;

    void Lex_Input();
    void Analysis();
    void print(string generative);

public:
    Parser(string grammar_file);
    Parser() {};
};
```

parser 就显得比较简单，其实就是把词法分析结果放到栈里面，边分析边输出。

2.6.1 Parser-读入词法分析结果到栈

```
void Parser::Lex_Input() {
    // 把东西放到栈中
    for (int i=lex_outcome.size()-1; i >= 0 ; i--)
        input.push_back(lex_outcome[i]);
}
```

非常简单，把词法分析 pair 的 vector 倒序插入栈。

2.6.2 Parser-分析过程

很简单的根据当前字符和输入找文法的过程。

只不过要注意符号序列消除的问题。

```
string top1;
pair<string, string> top2;
while (1) {

    // 获得当前两个栈栈顶
    top1 = Symbol.back();
    top2 = input.back();

    string index;   // 当前保存的符号
    // 符号
    if (top2.first == "PUNCTUATOR")
        index = top2.second;
    // 数据
    else if (top2.first == "NUMBER")
        index = "NUMBER";


    // 统计出错数量
    int fail = 0;
    // 结果
    string final_str = "";
    // 保存生成信息
    string generative = "";
    vector<string> search_index;

    // 两栈栈顶一样
    if (top1 == index) { ... }
    // 栈顶不一样
    else { ... }

    // 输出行信息
    print(generative);

    // 输出结束信息
    if (Symbol.size() == 1) { ... }
}
```

2.6.3 Parser-输出

我觉得还是很简单

就是填满相应的空格大小输出就可以。

2.7 运行结果

First 和 Follow 集，以及预测分析表

分析过程

样例程序可以运行完毕。（没有过多测试失败情况和左递归情况）

3. YACC

3.1 Lex 部分

比较简单，根据提供的文法只需识别 +、-、*、/、（、）这几个终结符和 num。

文本编辑器

打开(O)

```
%{
    #include<stdio.h>
    #include "y.tab.h"
%}

digit       ([0-9])
number      ({digit}+)

%%

{number}   {
    return number;
}

\+   {
    return '+';
}
\-   {
    return '-';
}
\*   {
    return '*';
}
\/   {
    return '/';
}
\(   {

    return '(';
}
\)   {

    return ')';
}
\n  {
    return '\n';
}
.   {
    printf("存在不匹配字符:%s\n",yytext);
    return;
}

%%
```

3.2 YACC 部分

最主要的部分就是非终结符产生式部分。

文本编辑器

打开(O)

```
%{
        #include<stdio.h>
        extern int yylex();
        extern int yyparse();
%}

%token number

%%


line : E'\n'    {printf("Expression has been read ended.\n");return;}
;
E : E'+'T       {printf("Use : E->E+T\n");}
    | E'-'T     {printf("Use : E->E-T\n");}
    | T         {printf("Use : E->T\n");}
    ;
T : T'*'F       {printf("Use : T->T*F\n");}
    | T'/'F     {printf("Use : T->T/F\n");}
    | F         {printf("Use : T->F\n");}
    ;
F : '('E')'     {printf("Use : F->(E)\n");}
    | num       {printf("Use : F->number\n");}
    ;


%%


int main(){
    yyparse();
    return 0;
}
void yyerror(char *s)
{
    printf("%s\n",s);
    system("pause");
}
```

3.3 运行结果

```
spike@spike-ubuntu:~/programs/compile$ bison -d y.y
spike@spike-ubuntu:~/programs/compile$ flex l.l
spike@spike-ubuntu:~/programs/compile$ gcc y.tab.c lex.yy.c -lfl
y.y: In function ‘yyparse’:
```

```
spike@spike-ubuntu:~/programs/compile$ ./a.out
(1+2)*3/4
Use : F->number
Use : T->F
Use : E->T
Use : F->number
Use : T->F
Use : E->E+T
Use : F->(E)
Use : T->F
Use : F->number
Use : T->T*F
Use : F->number
Use : T->T/F
Use : E->T
Success
spike@spike-ubuntu:~/programs/compile$
```

4. 问题&解决

4.1 Lex->Parser

本来想着直接用文件来读取。因为 lex 最终把词法分析结果输出到文件。但是在写完读文件之后，一直读的是空文件。可能是因为系统调用的时间比不上运行时间，所以就改用全局变量了。

4.2 First 和 Follow 递归

在这之间我尝试了很多错误方法。。。不过还是解决了。 主要出在 First 的处理上。有时候递归不到位导致少了。还有 Follow 的$处理一开始也忘了。

4.3 栈的分析过程

栈在分析的时候还是有很多需要主义的。尤其是消除的时候，可能是多个符号序列来消除。这时候需要记录前几个。就是在这里花费了我很多时间。

5. 实验总结

实现了 LL1 和 YACC

LL1：文法输入，左递归/左公因子的消除，first/follow 集合的生成，分析表的构建等，非常难，花了很长时间。但 YACC 好像还挺简单。这两个一对比发现 YACC 好太多了。不过这 LL1 加深了我对课本知识的理解。