杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

《编译原理》课程实验报告

(2021 年)

专　　业: 计算机科学与技术

学　　号: **********

班　　级: **********

姓　　名: *******

上课时间: **********

目录

1. 课程实验概述

实验概述部分说明自己的课程实验完成情况，介绍完成了哪些实验，每个实验之间关联度。

报告正文用小四字号、宋体，每个专题写一章（如果多个实验专题在一个程序中实现的，可以写在一章中，并在标题中注明包括了哪几个实验专题）

实验材料提交要求:

① 材料整理方式: 以“学号_姓名”新建一个文件夹，将本文档命名为”学号_姓名_实验报告.docx 或 pdf“，放入该文件夹。每个程序源码一个子文件夹

学号_姓名	学号_姓名_实验报告.docx或pdf	
	source	实验一源码
		实验二源码
		每个程序一个文件夹

提交以”学号_姓名”为压缩文件名

② 截止时间: 6 月 21 日晚上 12 点

③ 提交地址: huangxx@hdu.edu.cn

④ 邮件主题: 编译原理实验报告-学号-姓名，邮件内容注明上课时间

2. 实验专题一、词法分析

2.1 实验目的与内容

能够把源代码文件分析为以给定规则为基础的单词序列。能够正确识别分隔符、标识符、关键字、常量整数并且把它们以二元组的形式输出到文本中。一旦识别到错误的输入，要能够立即报错并且提供错误所在的行号。

正确词语的输出格式：<类型，对应符号>，每个关键字单独一类，元组的第二位设置为'@'字符。

错误词语的输出格式：Error: 错误提示信息 at line 行号。错误有块注释未闭合、标识符不合法（如 2as）等。

实验的最后输出结果可供后面的递归下降子程序与 LL1 分析器使用。

2.2 程序总体设计思路和框架

通过把源代码按分隔符、空格、运算符等分成一个个词汇单元，然后交由专门的处理函数 *judgeLegality()*来分析词汇的合法性。关键词、分隔符、运算符这种有穷集合依靠集合枚举，标识符依靠单字符判别来判断合法性。注释的识别集成在词汇提取函数 *analysis()*中，该函数按行读取源代码，可以过滤注释和空白字符，然后提取词汇单元进行相应的解析。行注释通过直接舍弃本行剩余的字符实现，块注释通过 *blockCheck* 标记来实现多行与行内的屏蔽。

2.3 主要的数据结构和流程描述

基本流程是双字符同时处理，可以同时针对注释与双字符运算符这两种特殊情况。优先判断注释，若不是注释再提取单词进行判别。

下列代码描述了词汇提取函数 *analysis()*中注释处理部分，其中 *current* 是该行字符中第 *i* 个字符，*next* 是第 *i+1* 个字符。

```
if (current == '/' && next == '/')
    if (blockCheck) //如果不在块注释中，直接忽略本行后面的内容
        break;
if (current == '/' && next == '*')
{
    blockCheck = false;
    ++i; // 为了防止/*/被认为是块注释而如此设置
    continue;
}
if (current == '*' && next == '/')
{
    if (blockCheck) // 如果只有闭合块注释符号，那么它是非法的
        illegalWordError("*/", lineNo);
```

```
    blockCheck = true;
    ++i;
    continue;
}
```

如果字符不在块注释中，那么再进行词汇的提取。本程序使用几个特征来标志一个词汇的结束与另一个词汇的开始。这些特征包括：空白字符，由一般字符转换为运算符以及遇见分隔符。

程序段中字符数组 *wordCandidate* 用来暂存可能会成为一个单词的字符串，*j* 是它的下标，*word* 则是被认定可能为一个词语的字符串。

以下是空白字符处理模块，遇到这些符号可以代表一个词语的结束：

```
if (current == ' ' || current == '\n' || current == '\t'
        || current == '\0') // 遇到换行、空格、Tab 时处理前一符号串
{
    word = wordCandidate;
    if (!word.empty())
        judgeLegality(word, lineNo);

    word.clear();
    memset(wordCandidate, 0, sizeof(wordCandidate));
    j = 0;
}
```

遇到分隔符或者运算符时对前一单词的处理也是同理，但是此时需要额外针对一下双字符运算符（分隔符全部为单字符），我采用贪心的方法优先匹配双字符运算符，当前字符不是双字符运算符时才判断单字符运算符与分隔符。程序段中 *s* 为 *current* 的 *string* 类型形式，贪心处理如下：

```
string temp = s + next; // 贪心处理双符号运算符
if (isOperator(temp))
{
    judgeLegality(temp, lineNo);
    ++i; // 此时跳过下一个字符，因为这个字符已经在双字符运算符处理中解决了
}
else
    judgeLegality(s, lineNo);
```

其他情况下代表该单词没有被认为结束，此时把当前字符放入 *wordCandidate* 中，等待该单词被认为结束的时候处理。

判断程序 *judgeLegality()* 的判断顺序是关键词=>运算符=>分隔符=>常数=>标识符的顺序来判断，然后按相对应的格式进行输出。如果这些情况均不符合，则此单词会被认为非法，将会报错。

另有一类报错是块注释没有闭合，此错误处理相对方便，只需在处理完所有字符后检查如果 *blockCheck* 标记不为 *true* 则报错即可。

2.4 测试结果与说明

测试了三段代码的结果，包含正确程序与错误程序。
<*ID*>代表标识符，<*SEP*>代表分隔符，<*OP*>代表运算符，其他的大写字符代表关键字，它们自成一类，二元组的第二维用*'@'*表示。

TestSample01:

此案例演示正确输出

测试源文件 *test.sy*:

```
int a[10][10];
int main(){
    s = 2;
    return 0;
}
```

输出结果 *test_la.txt*:

```
<INT, @>
<ID, a>
<SEP, [>
<NUM, 10>
<SEP, ]>
<SEP, [>
<NUM, 10>
<SEP, ]>
<SEP, ;>
<INT, @>
<MAIN, @>
<SEP, (>
<SEP, )>
<SEP, {>
<ID, s>
<OP, =>
<NUM, 2>
<SEP, ;>
<RETURN, @>
<NUM, 0>
<SEP, ;>
<SEP, }>
```

Figure 1 Lexical Analysis Test Sample 01

TestSample02:

此案例演示块注释不闭合错误

测试源文件 *test.sy*:

```
int main(){
    int b[10];
        int c;
        int d=a*2;
    /*
    The comment block;
    int a=5;
    return 0;
}
```

输出结果 *test_la.txt*:

```
<INT, @>
<MAIN, @>
<SEP, (>
<SEP, )>
<SEP, {>
<INT, @>
<ID, b>
<SEP, [>
<NUM, 10>
<SEP, ]>
<SEP, ;>
<INT, @>
<ID, c>
<SEP, ;>
<INT, @>
<ID, d>
<OP, =>
<ID, a>
<OP, *>
<NUM, 2>
<SEP, ;>
Error: Block comment lack of '*/' error at line 9
```

Figure 2 Lexical Analysis Test Sample 02

TestSample03:

此案例演示非法字符与不合法标识符错误

测试源文件 *test.sy*:

```
int 2foo;
int main(){
    a@b = 10;
    return 0;
}
```

输出结果 *test_la.txt*:

```
<INT, @>
Error: Illegal word error '2foo' at line 1
<SEP, ;>
<INT, @>
<MAIN, @>
<SEP, (>
<SEP, )>
<SEP, {>
Error: Illegal word error 'a@b' at line 3
<OP, =>
<NUM, 10>
<SEP, ;>
<RETURN, @>
<NUM, 0>
<SEP, ;>
<SEP, }>
```

Figure 3　Lexical Analysis Test Sample 03

2.5 实验收获与反思

做这个词法分析实验借鉴了很多人的思想，但是照着别人的思想或多或少都会遇到很多问题，到头来还是得重构一遍代码。做这个实验最大的收获就是对文件操作熟练了很多，然后就是因为这个程序分支结构太多，而且很容易出现不容易预知的 bug，我的程序阅读与调试能力也有了很大的提升。最后就是真正体会到了对语言分析的难度，我可能选择了最难写的一种，写这个实验是我在写编译原理的所有实验里面耗时最长的，感觉真的就是硬分析，书上提到的算法，例如有限自动机什么的，那些思想我还是没能运用进来。应用什么的可能还是就停留在表面上吧。

3. 实验专题二、有限自动机相关算法实现

基于 MYT 算法的正规表达式到 NFA

3.1 实验目的与内容

给定一正规表达式，先分析出它的逆波兰表达式，若此步转换报错则直接结束程序。得到正规式的逆波兰表达式后，据其生成对应的 NFA，然后跟据 NFA 信息输出对应图的*.dot* 源文件，然后经 *dot -Tpng *.dot -o *.png* 命令将其转换为 *PNG* 图像。

3.2 程序总体设计思路和框架

整个程序分程两部分：求逆波兰表达式与求 NFA 转换两部分。

求逆波兰表达式就按照运算数直接输出，左括号压栈，遇到右括号集体出栈，然后比较栈顶符号与当前操作运算符的运算优先级来实现。处理正规表达式的连接操作时，在两个要连接的符号间添加*'.'*作为运算符用来方便后期统一处理。一旦遇到错误则输出一个以感叹号开始的字符串（用作错误标记）来表示错误原因，同时终止程序。若成功把正规表达式转换为逆波兰表达式，则返回这个逆波兰表达式用于后期处理。

求 NFA 时对每一个运算符（*| . *）都提供一个函数来处理（添加节点），这些函数是 *generate()*, *choose()*, *connect()*，它们可以完成对应运算符下的节点连接操作。

3.3 主要的数据结构和流程描述

主要还是模仿手工按照 MYT 算法绘制 NFA 的操作。使用一个结构体 *NodeBind* 表示每一个未连接起来的连通子图，每一个节点使用 *Node* 结构体表示，下标表示该节点的标号，该结构体里另有两个向量来表示边，分别代表有出边相连的点的标号以及该边的符号。

```
struct Node
{
    vector<int> next; // 连到下一个点的下标
    vector<char> id; // 该边的id
    void clear()
    {
        next.clear();
        id.clear();
    }
};
struct NodeBind
{
    int begin;
```

```
    int end;
};
Node nodes[100]{}; // 存放NFA图
```

NodeBind 的两个数据分别表示该连通子图的起始点与终止点下标，每次生成这样一副连通子图就把它压入栈。

遍历逆波兰表达式，当读到的字符是标识符，则开辟两个顶点并且设置一条边连接它们，这条边的值即为这个标识符，然后将这张子图压入 *NodeBind* 的栈中，相关代码如下：

```
if (isalpha(c) || isdigit(c))
{
    NodeBind temp{};
    temp.begin = ++count;
    temp.end = ++count;
    nodes[temp.begin].next.push_back(temp.end);
    nodes[temp.begin].id.push_back(c);
    edgeStack.push(temp);
}
```

如果读到的是运算符，再跟据运算符的要求弹出栈中的子图，再把它们连接到一起或者在前后添加新节点与 ε 边然后再压栈。'.'操作与'|'操作是双目运算符，需要弹出栈中的两个子图进行操作，'*'是单目的，只需要弹出栈顶的一个子图，添加节点与 ε 后再压栈。

三种运算符的操作如下：

```
if (c == '.')
{
    NodeBind temp1 = edgeStack.top();
    edgeStack.pop();
    NodeBind temp2 = edgeStack.top();
    edgeStack.pop();
    NodeBind temp = connect(temp2, temp1);
    edgeStack.push(temp);
}
else if (c == '|')
{
    NodeBind temp1 = edgeStack.top();
    edgeStack.pop();
    NodeBind temp2 = edgeStack.top();
    edgeStack.pop();
    NodeBind temp = choose(temp2, temp1);
    edgeStack.push(temp);
}
else if (c == '*')
{
```

```
    NodeBind temp1 = edgeStack.top();
    edgeStack.pop();
    NodeBind temp = generate(temp1);
    edgeStack.push(temp);
}
```

最后遍历完成就可以得到 NFA，然后，按照以下逻辑输出 dot 代码：

```
cout << "digraph nfa {" << endl;
cout << '\t' << "label = \"" << exp << "\";" << endl;\
cout << '\t' << "rankdir = LR;" << endl;
for (int i = 1; i <= count; ++i)
    for (int j = 0; j < nodes[i].next.size(); ++j)
        if (nodes[i].next[j] != 0)
            if (nodes[i].id[j] == '$') // 为了输出好看点
                printf("\t%d -> %d [shape = circle, label =
\"%s\"];\n", i, nodes[i].next[j], "ε");
            else // NOLINT
                printf("\t%d -> %d [shape = circle, label =
\"%c\"];\n", i, nodes[i].next[j], nodes[i].id[j]);
cout << "}" << endl;
```

3.4 测试结果与说明

测试了 5 个正规表达式，包括 3 个基础的 MYT 算法的图以及 2 个测试案例。

TestSample01:

测试表达式为：*r**
输出 *nfa.txt* 文件为：

```
digraph nfa {
    label = "r*";
    rankdir = LR;
    1 -> 2 [shape = circle, label = "r"];
    2 -> 1 [shape = circle, label = "ε"];
    2 -> 4 [shape = circle, label = "ε"];
    3 -> 1 [shape = circle, label = "ε"];
    3 -> 4 [shape = circle, label = "ε"];
}

```

Figure 4 Regex Parser Test Sample 01 Text

输出 NFA 图为：

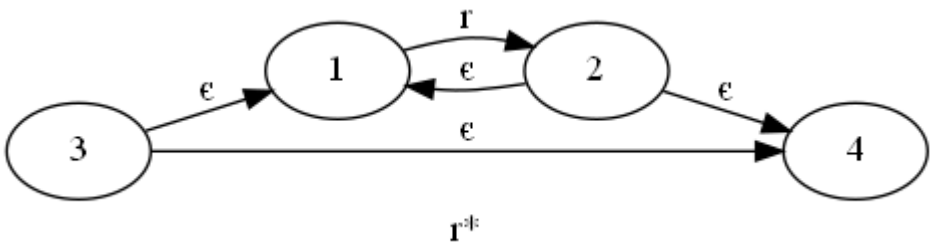

Figure 5 Regex Parser Test Sample 01 NFA

TestSample02:

测试表达式为：*r|s*
输出 *nfa.txt* 文件为：

```
digraph nfa {
    label = "r|s";
    rankdir = LR;
    1 -> 2 [shape = circle, label = "r"];
    2 -> 6 [shape = circle, label = "ε"];
    3 -> 4 [shape = circle, label = "s"];
    4 -> 6 [shape = circle, label = "ε"];
    5 -> 1 [shape = circle, label = "ε"];
    5 -> 3 [shape = circle, label = "ε"];
}

```

Figure 6 Regex Parser Test Sample 02 Text

输出 NFA 图为：

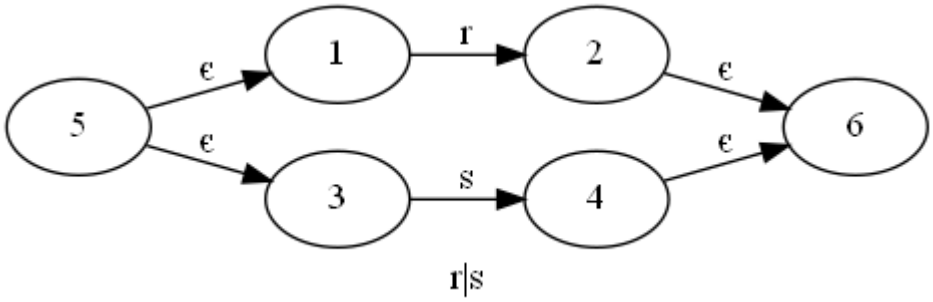

Figure 7 Regex Parser Test Sample 02 NFA

TestSample03:

测试表达式为：*rs*
输出 *nfa.txt* 文件为：

```
digraph nfa {
    label = "rs";
    rankdir = LR;
    1 -> 2 [shape = circle, label = "r"];
    2 -> 4 [shape = circle, label = "s"];
}

```

Figure 8 Regex Parser Test Sample 03 Text

输出 NFA 图为：

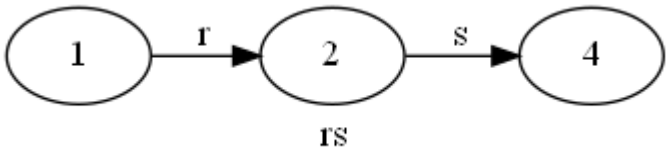

Figure 9 Regex Parser Test Sample 03 NFA

TestSample04:

测试表达式为：*(a|b)*abb*

输出 *nfa.txt* 文件为：

```
digraph nfa {
    label = "(a|b)*abb";
    rankdir = LR;
    1 -> 2 [shape = circle, label = "a"];
    2 -> 6 [shape = circle, label = "ε"];
    3 -> 4 [shape = circle, label = "b"];
    4 -> 6 [shape = circle, label = "ε"];
    5 -> 1 [shape = circle, label = "ε"];
    5 -> 3 [shape = circle, label = "ε"];
    6 -> 5 [shape = circle, label = "ε"];
    6 -> 8 [shape = circle, label = "ε"];
    7 -> 5 [shape = circle, label = "ε"];
    7 -> 8 [shape = circle, label = "ε"];
    8 -> 10 [shape = circle, label = "a"];
    10 -> 12 [shape = circle, label = "b"];
    12 -> 14 [shape = circle, label = "b"];
}

```

Figure 10 Regex Parser Test Sample 04 Text

输出 NFA 图为：

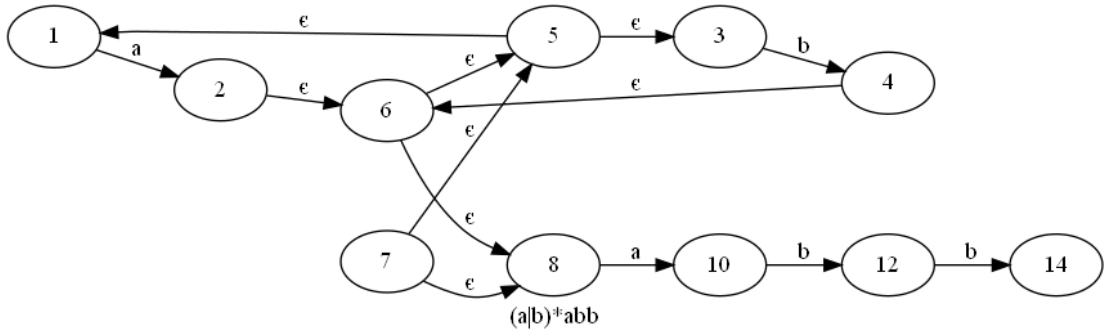

Figure 11 Regex Parser Test Sample 04 NFA

TestSample05:

测试表达式为：*((a|b)*aa)*b*
输出 *nfa.txt* 文件为：

```
digraph nfa {
    label = "((a|b)*aa)*b";
    rankdir = LR;
    1 -> 2 [shape = circle, label = "a"];
    2 -> 6 [shape = circle, label = "ε"];
    3 -> 4 [shape = circle, label = "b"];
    4 -> 6 [shape = circle, label = "ε"];
    5 -> 1 [shape = circle, label = "ε"];
    5 -> 3 [shape = circle, label = "ε"];
    6 -> 5 [shape = circle, label = "ε"];
    6 -> 8 [shape = circle, label = "ε"];
    7 -> 5 [shape = circle, label = "ε"];
    7 -> 8 [shape = circle, label = "ε"];
    8 -> 10 [shape = circle, label = "a"];
    10 -> 12 [shape = circle, label = "a"];
    12 -> 7 [shape = circle, label = "ε"];
    12 -> 14 [shape = circle, label = "ε"];
    13 -> 7 [shape = circle, label = "ε"];
    13 -> 14 [shape = circle, label = "ε"];
    14 -> 16 [shape = circle, label = "b"];
}

```

Figure 12 Regex Parser Test Sample 05 Text

输出 NFA 图为：

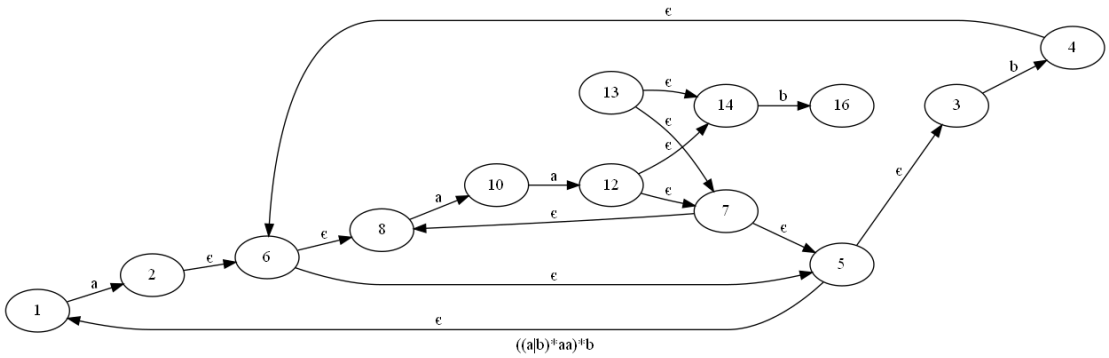

Figure 13 Regex Parser Test Sample 05 NFA

3.5 实验收获与反思

写这个实验的时候，第一步化逆波兰表达式的时候还是比较顺利的，参照大一数据结构课上的算法，得到逆波兰表达式基本没有困扰到我。但是在生成图的时候，卡了很久。邻接矩阵与邻接表在这种要处理多个子图的时候都不是那么好用，最后借鉴了我之前写哈夫曼算法的思想，数组存节点，每个节点里面设值去指向下一个节点。这个方法最后终于解决了问题。后来验收的时候老师说可以用三元组解决，我才发现我存放数据分程两个向量不是很美观，下次可以改进。做这个实验最大的收获就是学习了一个画图的好软件 dot，感觉以后画流程图会方便很多。

4. 实验专题三、递归下降子程序的语法分析技术

4.1 实验目的与内容

给定实验一得到的词语序列后，跟据特定的语法规则，通过递归下降分析法判断一个表达式是否合乎语法。要能正确按照语法规则完成句型的分析，如果遇到错误则进行错误标记，然后跳过错误识别的字符，继续处理下一个字符直至处理完成。

如果正确分析，能输出每一步使用的产生式，如果遇到错误，能在错误发生的位置输出产生式不匹配错误。

4.2 程序总体设计思路和框架

首先跟据文法得到分析表:

	i	*(*	*)*	*+*	*-*	***	*/*	*$*
E	E→TE'	E→TE'						
T	T→FT'	T→FT'						
E'			E'→ε	E'→ATE'	E'→ATE'			E'→ε
A				A→+	A→-			
F	F→i	F→(E)						
M						M→*	M→/	
T'			T'→ε	T'→ε	T'→ε	T'→MFT'	T'→MFT'	T'→ε

Table 1 Grammar Analysis Table

为每个非终结符号写一个解析函数，跟据输入递归调用该产生式右部非终结符号的解析函数。当产生式右部出现终结符号时，代表终结符号匹配成功，此时可以继续解析表达式中的下一个符号，直到解析至*'$'*代表匹配结束，然后跟据错误记号 *haveError* 下最后的结论判断该表达式是否合法。

4.3 主要的数据结构和流程描述

词法分析得到的二元组直接交给语法分析去分析过于复杂，先将其重新转化为单行字符串抽象表达式，即所有的标识符与常数都视为*'i'*，运算符原样不动，其他所有类型的单词均视为非法字符，被标记成*'@'*。以下函数用于完成此功能:

```
void readLexicalResult() // 提取词法分析得到的序列
{
    char c;
    string type, word;
    while (cin >> c)
    {
        if (' ' != c) // 过滤空格
            if (c == '<')
```

```
            {
                cin >> type >> word;
                if (type == "ID," || type == "NUM,")
                                // 立即数与标识符均视为标识符
                    expr += 'i';
                else if (type == "OP,")
                    expr += word.substr(0, word.length() - 1);
                                // 去除读到的最后一个 >
                else // 非法词汇均视为@
                    expr += '@';
            }
    }
    fclose(stdin);
    expr += '$'; // 结尾加开始符号
}
```

为每个非终结符号写的解析函数形式上都类似，这边仅展示一个非终结符号 E 的函数：

```
void E(char t)
{
    if (t == '(' || t == 'i')
    {
        cout << "E -> TE'" << endl;
        T(expr[i]);
        E_(expr[i]); // E_视为E'
    }
    else
    {
        checkError();
        ++i; // 跳过当前错误字符，处理下一个
        if (expr[i] != '$') // 读到最后一个的时候，即使错误也不再进行
            E(expr[i]);
    }
}
```

错误处理借由 *haveError* 标记，借由 *checkError* 函数输出错误信息。

4.4 测试结果与说明

测试了 5 段词汇序列，包含 3 组正确程序与 2 组错误程序。

TestSample01:

此案例演示了正确的纯识符组成的表达式的分析输出
表达式为：*i*i+i-i$*
测试源文件 *test_la.txt*：

```
<ID, i>
<OP, *>
<ID, i>
<OP, +>
<ID, i>
<OP, ->
<ID, i>
```

输出结果 *test_rp.txt*：

```
The Expression is: i*i+i-i$
E -> TE'
T -> FT'
F -> i
T'-> MFT'
M -> *
F -> i
T'-> ε
E'-> ATE'
A -> +
T -> FT'
F -> i
T'-> ε
E'-> ATE'
A -> -
T -> FT'
F -> i
T'-> ε
E'-> ε
...This Expression is CORRECT...
```

Figure 14 Recursive Parsing Test Sample 01

TestSample02:

此案例演示了正确的常数与标识符组成的表达式的分析输出
表达式为：*2+a$*
测试源文件 *test_la.txt*：

```
<NUM, 2>
<OP, +>
<ID, a>
```

输出结果 *test_rp.txt*：

```
The Expression is: i+i$
E -> TE'
T -> FT'
F -> i
T'-> ε
E'-> ATE'
A -> +
T -> FT'
F -> i
T'-> ε
E'-> ε
...This Expression is CORRECT...
```

Figure 15 Recursive Parsing Test Sample 02

TestSample03:

此案例演示了出现了运算符错误的表达式的分析输出
表达式为：*i**i++i--i$*

测试源文件 *test_la.txt*：

```
<ID, i>
<OP, *>
<OP, *>
<ID, i>
<OP, +>
<OP, +>
<ID, i>
<OP, ->
<OP, ->
<ID, i>
```

输出结果 *test_rp.txt*：

```
The Expression is: i**i++i--i$
E -> TE'
T -> FT'
F -> i
T'-> MFT'
M -> *
Error: Production NOT Match
F -> i
T'-> ε
E'-> ATE'
A -> +
Error: Production NOT Match
T -> FT'
F -> i
T'-> ε
E'-> ATE'
A -> -
Error: Production NOT Match
T -> FT'
F -> i
T'-> ε
E'-> ε
!!!This Expression is ILLEGAL!!!
```

Figure 16 Recursive Parsing Test Sample 03

TestSample04:

此案例演示了出现了错误关键字的表达式的分析输出
表达式为：*2+int+2$*

测试源文件 *test_la.txt*：

```
<NUM, 2>
<OP, +>
<INT, @>
<OP, +>
<NUM, 2>
```

输出结果 *test_rp.txt*：

```
The Expression is: i+@+i$
E -> TE'
T -> FT'
F -> i
T'-> ε
E'-> ATE'
A -> +
Error: Production NOT Match
Error: Production NOT Match
T -> FT'
F -> i
T'-> ε
E'-> ε
!!!This Expression is ILLEGAL!!!
```

Figure 17 Recursive Parsing Test Sample 04

TestSample05:

此案例演示了出现了括号时的分析输出
表达式为：*(i+i*)i$*
测试源文件 *test_la.txt*：

```
<SEP, (>
<ID, i>
<OP, +>
<ID, i>
<SEP, )>
<OP, *>
<ID, i>
```

输出结果 *test_rp.txt*：

```
The Expression is: (i+i)*i$
E -> TE'
T -> FT'
F -> (E)
E -> TE'
T -> FT'
F -> i
T'-> ε
E'-> ATE'
A -> +
T -> FT'
F -> i
T'-> ε
E'-> ε
T'-> ε
E'-> ε
...This Expression is CORRECT...
```

Figure 18 Recursive Parsing Test Sample 05

4.5 实验收获与反思

之前上课的时候其实对递归下降分析法了解的不是很清楚，因为一直以为只要学好LL(1)的那个填表方法就行了，对书上的例子也没很懂，写完这个实验后，才是对这整个算法的流程有了更清晰的认识，感觉它写起来也要比 LL(1)分析法更快捷些。

5. 实验专题四、LL(1)语法分析器的设计与实现

5.1 实验目的与内容

以另一种方法实现对表达式的语法分析检查。使用实验一得到的词语序列，使用LL(1)的自顶向下自左向右的方式分析表达式，判断它是否合乎语法。要能正确按照语法规则完成句型的分析，匹配成功则输出此次匹配的字符，匹配失败输出失败的原因，例如产生式不匹配或者识别到非法终结符号。

同时要能输出每一步的分析栈中的内容，输出形式类似课本上的自顶向下分析表。最后给出结论判断该表达式是否合法。

5.2 程序总体设计思路和框架

首先跟之前的递归下降子程序一样，构建此文法的分析表（见 *Table 1*），然后处理词法分析的二元组并把它转换为抽象表达式。然后通过一个个遍历表达式里的字符进行分支操作。

每次比对栈顶元素与表达式当前处理字符，若两者不同则弹出栈顶，将其右部重新压入处理栈，输出此次被选中的产生式，再进行下一次对比，直到栈顶与目前待处理字符均为'$'为止。若出现选择产生式为空或者无法解析字符，则立即报错并跳过该字符。

5.3 主要的数据结构和流程描述

产生式按照行号与列号映射下标，然后将右部存入一个字符串数组中。

数组的下标映射由以下实现：

```
char Vn[200] = "ETeAFMt"; // e == E', t == T'
char Vt[200] = "i()+-*/$";
int findVn(char c)
{
    for (int i = 0; i < 7; i++) //找到产生式对应行
        if (c == Vn[i])
            return i;
    return -1;
}
int findVt(char c)
{
    for (int i = 0; i < 8; i++) //找到产生式对应列
        if (c == Vt[i])
            return i;
    return -1;
}
```

依此可以通过当前处理字符与产生式左部快速找到产生式右部。

对二元组的处理同递归下降子程序，不再赘述。

以下为当栈顶为非终结符号而且成功选择到产生式时的行为：

```
if (product != "ε") // 产生式右部入栈
    for (int q = product.length() - 1; q >= 0; q--)
        procStack[stackTop++] = product[q];
else
    procStack[stackTop] = '\0';
```

以下是当选择到的产生式为空时的行为：舍弃当前处理字符，保留当前非终结符号

```
if (product == "null") // 选不到产生式，报错
{
    printf("%-5d%-10s%-15s%s\n", step, procStack, exp.c_str() + i,
                                "(ERROR: Production NOT Match)");
    step++;
    haveError = true; // 错误处理标记
    procStack[stackTop++] = ch; // 原非终结符号压回栈
    ++i; // 舍弃此终结符号
    continue;
}
```

以下是当前处理字符为非法终结符号时的行为：舍弃当前处理字符，保留当前非终结符号

```
else // 表达式下一个符号是非法终结符号
{
    printf("%-5d%-10s%-15s%s\n", step, procStack, exp.c_str() + i,
                        "(ERROR: ILLEGAL Terminate Symbol)");
    step++;
    haveError = true;
    procStack[stackTop++] = ch; // 原非终结符号压回栈
    ++i; // 舍弃此终结符号
    continue;
}
```

当处理栈顶出现非终结符号时，如果和当前处理字符一致那么匹配成功，开始处理下一个字符，处理栈弹出该非终结符号。特殊地，当栈顶为'$'时如果当前处理字符也是'$'那么输出成功匹配提醒并停止匹配。其他情况为错误情况，程序报错。处理如下：

```
if (ch == exp[i])
{
    --stackTop;
    printf("%-5d%-10s", step, procStack);
    if (ch == '$' && exp[i] == '$')
    {
        printf("%-15s%-10s\n", "$", "(Accept)");
        return;
```

```
    }
    printf("%-15s(%c %s\n", exp.c_str() + i, ch, "Matched)");
    procStack[stackTop] = '\0';
    ++i;
}
else
{
    printf("%-10s\n", "ERROR");
    ++i;
    continue;
}
```

另设 *haveError* 标记用于错误处理以及最后的结论输出。

5.4 测试结果与说明

测试了 5 段词汇序列，包含 3 组正确程序与 2 组错误程序。

TestSample01:

此案例演示了正确的纯识符组成的表达式的分析输出
表达式为：*i*i+i-i$*
测试源文件 *test_la.txt*：

```
<ID, i>
<OP, *>
<ID, i>
<OP, +>
<ID, i>
<OP, ->
<ID, i>
```

输出结果 *test_ll1.txt*：

```
The Expression is: i*i+i-i$
Step Stack|->  Expression     Production
1    $E        i*i+i-i$       E->Te
2    $eT       i*i+i-i$       T->Ft
3    $etF      i*i+i-i$       F->i
4    $eti      i*i+i-i$       (i Matched)
5    $et       *i+i-i$        t->MFt
6    $etFM     *i+i-i$        M->*
7    $etF*     *i+i-i$        (* Matched)
8    $etF      i+i-i$         F->i
9    $eti      i+i-i$         (i Matched)
10   $et       +i-i$          t->ε
11   $e        +i-i$          e->ATe
12   $eTA      +i-i$          A->+
13   $eT+      +i-i$          (+ Matched)
14   $eT       i-i$           T->Ft
15   $etF      i-i$           F->i
16   $eti      i-i$           (i Matched)
17   $et       -i$            t->ε
18   $e        -i$            e->ATe
19   $eTA      -i$            A->-
20   $eT-      -i$            (- Matched)
21   $eT       i$             T->Ft
22   $etF      i$             F->i
23   $eti      i$             (i Matched)
24   $et       $              t->ε
25   $e        $              e->ε
26   $         $              (Accept)
...This Expression is CORRECT...
```

Figure 19 LL(1) Parser Test Sample 01

TestSample02:

此案例演示了正确的常数与标识符组成的表达式的分析输出
表达式为：*2+a$*
测试源文件 *test_la.txt*：

```
<NUM, 2>
<OP, +>
<ID, a>
```

输出结果 *test_ll1.txt*：

```
The Expression is: i+i$
Step Stack|->  Expression     Production
1    $E        i+i$           E->Te
2    $eT       i+i$           T->Ft
3    $etF      i+i$           F->i
4    $eti      i+i$           (i Matched)
5    $et       +i$            t->ε
6    $e        +i$            e->ATe
7    $eTA      +i$            A->+
8    $eT+      +i$            (+ Matched)
9    $eT       i$             T->Ft
10   $etF      i$             F->i
11   $eti      i$             (i Matched)
12   $et       $              t->ε
13   $e        $              e->ε
14   $         $              (Accept)
...This Expression is CORRECT...
```

Figure 20 LL(1) Parser Test Sample 02

TestSample03:

此案例演示了出现了运算符错误的表达式的分析输出
表达式为：*i**i++i--i$*
测试源文件 *test_la.txt*：

```
<ID, i>
<OP, *>
<OP, *>
<ID, i>
<OP, +>
<OP, +>
<ID, i>
<OP, ->
<OP, ->
<ID, i>
```

输出结果 *test_ll1.txt*：

```
The Expression is: i**i++i--i$
Step Stack|->  Expression     Production
1    $E        i**i++i--i$    E->Te
2    $eT       i**i++i--i$    T->Ft
3    $etF      i**i++i--i$    F->i
4    $eti      i**i++i--i$    (i Matched)
5    $et       **i++i--i$     t->MFt
6    $etFM     **i++i--i$     M->*
7    $etF*     **i++i--i$     (* Matched)
8    $etF      *i++i--i$      (ERROR: Production NOT Match)
9    $etF      i++i--i$       F->i
10   $eti      i++i--i$       (i Matched)
11   $et       ++i--i$        t->ε
12   $e        ++i--i$        e->ATe
13   $eTA      ++i--i$        A->+
14   $eT+      ++i--i$        (+ Matched)
15   $eT       +i--i$         (ERROR: Production NOT Match)
16   $eT       i--i$          T->Ft
17   $etF      i--i$          F->i
18   $eti      i--i$          (i Matched)
19   $et       --i$           t->ε
20   $e        --i$           e->ATe
21   $eTA      --i$           A->-
22   $eT-      --i$           (- Matched)
23   $eT       -i$            (ERROR: Production NOT Match)
24   $eT       i$             T->Ft
25   $etF      i$             F->i
26   $eti      i$             (i Matched)
27   $et       $              t->ε
28   $e        $              e->ε
29   $         $              (Accept)
!!!This Expression is ILLEGAL!!!
```

Figure 21 LL(1) Parser Test Sample 03

TestSample04:

此案例演示了出现了错误关键字的表达式的分析输出
表达式为：*2+int+2$*
测试源文件 *test_la.txt*：　　输出结果 *test_ll1.txt*：

```
<NUM, 2>
<OP, +>
<INT, @>
<OP, +>
<NUM, 2>
```

```
The Expression is: i+@+i$
Step Stack|->  Expression     Production
1    $E        i+@+i$         E->Te
2    $eT       i+@+i$         T->Ft
3    $etF      i+@+i$         F->i
4    $eti      i+@+i$         (i Matched)
5    $et       +@+i$          t->ε
6    $e        +@+i$          e->ATe
7    $eTA      +@+i$          A->+
8    $eT+      +@+i$          (+ Matched)
9    $eT       @+i$           (ERROR: ILLEGAL Terminate Symbol
10   $eT       +i$            (ERROR: Production NOT Match)
11   $eT       i$             T->Ft
12   $etF      i$             F->i
13   $eti      i$             (i Matched)
14   $et       $              t->ε
15   $e        $              e->ε
16   $         $              (Accept)
!!!This Expression is ILLEGAL!!!
```

Figure 22 LL(1) Parser Test Sample 04

TestSample05:

此案例演示了出现了括号时的分析输出
表达式为：*(i+i)*i$*
测试源文件 *test_la.txt*：　　输出结果 *test_ll1.txt*：

```
<SEP, (>
<ID, i>
<OP, +>
<ID, i>
<SEP, )>
<OP, *>
<ID, i>
```

```
The Expression is: (i+i)*i$
Step Stack|->  Expression     Production
1    $E        (i+i)*i$       E->Te
2    $eT       (i+i)*i$       T->Ft
3    $etF      (i+i)*i$       F->(E)
4    $et)E(    (i+i)*i$       (( Matched)
5    $et)E     i+i)*i$        E->Te
6    $et)eT    i+i)*i$        T->Ft
7    $et)etF   i+i)*i$        F->i
8    $et)eti   i+i)*i$        (i Matched)
9    $et)et    +i)*i$         t->ε
10   $et)e     +i)*i$         e->ATe
11   $et)eTA   +i)*i$         A->+
12   $et)eT+   +i)*i$         (+ Matched)
13   $et)eT    i)*i$          T->Ft
14   $et)etF   i)*i$          F->i
15   $et)eti   i)*i$          (i Matched)
16   $et)et    )*i$           t->ε
17   $et)e     )*i$           e->ε
18   $et)      )*i$           () Matched)
19   $et       *i$            t->MFt
20   $etFM     *i$            M->*
21   $etF*     *i$            (* Matched)
22   $etF      i$             F->i
23   $eti      i$             (i Matched)
24   $et       $              t->ε
25   $e        $              e->ε
26   $         $              (Accept)
...This Expression is CORRECT...
```

Figure 23 LL(1) Parser Test Sample 05

5.5 实验收获与反思

这个算法可能是我在编译原理这门课里最熟练的一个算法了。整体写下来相比第一个词法分析的提词要舒服了不是一点半点，这个思路还是很轻松的，主要还是纸笔分析出分析表。做完这几个实验，对这几个算法熟悉了真的不是一点半点。其实剥掉这个表，算法部分对每个文法都是差不多一样的，只要给程序提供算好了的分析表，就能针对不同文法得到不同的语法分析器，所以理论上我写的这个还是有一定通用性的（可惜这边没实现）。