東南大學
SOUTHEAST UNIVERSITY

编译原理实验报告

姓名 李宗辉

学号 61522122

教师 戚晓芳

2024 年 12 月 21 日

目录

1 Lexical Analyzer Programming

1.1 实验目的

- 理解词法分析过程。
- 熟练掌握有限自动机（FA）、正则表达式（RE）和文法（Grammar）之间的转换关系。
- 自主实现词法分析程序。

1.2 实验内容

- 自主定义一些 token 对应的 RE。
- 根据状态转换图，实现一个词法分析器。
- 对于合法字符串，输出解析出来的的 token；对于不合法字符串，输出错误信息。

1.3 实验思路

首先，采用 Thompson's construction 将每个 token 对应的正则表达式（RE）转换为非确定性有限自动机（NFA），由于都是用相同的状态转换逻辑进行识别，因此可以将其合并为一个大的 NFA，接着利用 subset method 将 NFA 转换为确定性有限自动机（DFA），最后通过 partition method 将 DFA 优化为最小 DFA。

然后逐个读取输入串字符，根据合并优化后的 DFA 模拟状态转换，从而由是否到达终态判断字符串是否合法，如果合法，则输出对应的 token 类型。

1.4 设定 RE 定义

1. 基础正则表达式（RE）定义

- letter → a | b | c | ... | z | A | B | C | ... | Z
- nonzero_digit → 1 | 2 | 3 | ... | 9
- digit → 0 | nonzero_digit

2.token 对应的 RE 定义

- Identifier → letter (letter | digit)*
- Integer → nonzero_digit (digit)*
- Decimal → (0 | Integer). (digit)+

- Operator → + | - | * | / | = | < | >
- Delimiter → (|) | [|] | { | } | , | ; | ”
- Keyword → if | else | while | for | return | break | continue | void | int | double | float | char | bool | string | true | false | const

1.5 相关 FA 描述

1.5.1 Identifier DFA

根据 RE 定义，可以得到 Identifier 对应的最小 DFA 如图 1：

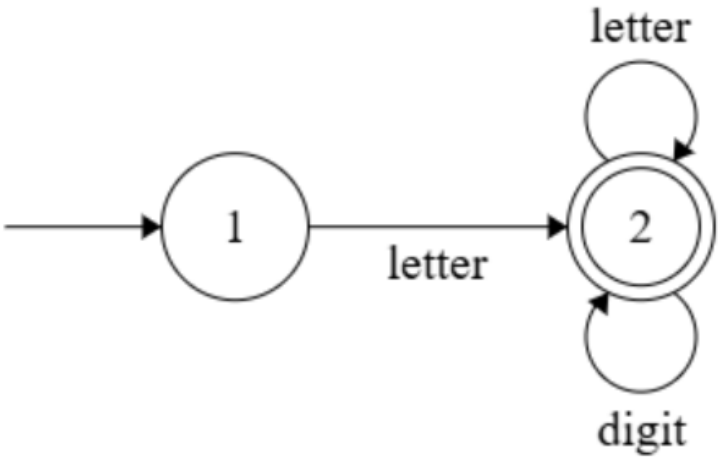

图 1: Identifier DFA

1.5.2 Integer DFA

根据 RE 定义，可以得到 Identifier 对应的最小 DFA 如图 2：

1.5.3 Decimal DFA

根据 RE 定义，可以得到 Identifier 对应的最小 DFA 如图 3：

1.5.4 Operator DFA

根据 RE 定义，可以得到 Identifier 对应的最小 DFA 如图 4：

1.5.5 Delimiter DFA

根据 RE 定义，可以得到 Identifier 对应的最小 DFA 如图 5：

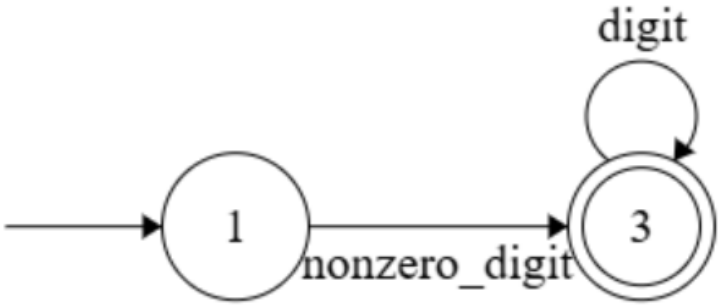

图 2: Integer DFA

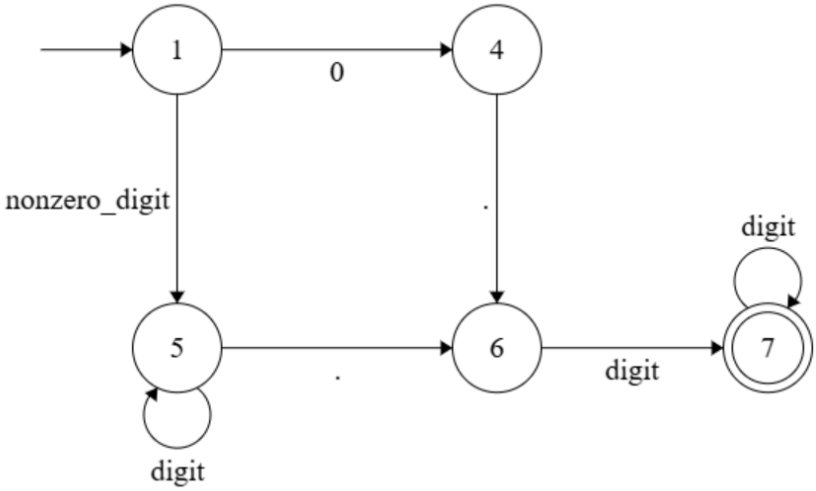

图 3: Decimal DFA

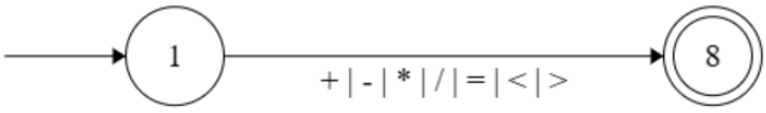

图 4: Operator DFA

图 5: Delimiter DFA

1.5.6 Keyword DFA

根据 RE 定义，可以得到 Identifier 对应的最小 DFA 如图 6：

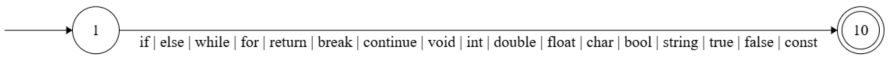

图 6: Keyword DFA

1.6 重要数据结构描述

使用 vector 容器来存储 token 和 token 所对应的字符串 values。

1.7 核心算法

1.7.1 程序框架

分析器主要使用了 if-else 和 switch-case 的控制结构，用于模拟 DFA 中 state 的转换。

类定义：定义一个名为 LexicalAnalyzer 的类来封装整个词法分析的功能。通过成员变量定义了输入字符串 input、输出的 tokens 容器、tokens 对应的字符串容器、模拟转换状态的 state 和当前分析到字符位置指针 cur 及分析下一个 token 的起始位置指针 pos。

初始化：init 方法初始化输入字符串并添加一个终结符 #，用于判断字符串是否分析完成；清空之前的 tokens 和 values，同时重置状态和位置指针。

字符分类：多个布尔函数用于判断字符的类型，例如：

- isLetter(char c): 判断字符是否为字母。
- isZero(char c): 判断字符是否为 0。
- isNonZeroDigit(char c): 判断字符是否为非 0 数字。
- isDigit(char c): 判断字符是否为数字。
- isOp(char c): 判断字符是否为运算符。
- isDlmtr(char c): 判断字符是否为分隔符。
- isKword(string s): 判断字符串是否为关键字。

分析 token：通过已经构造出的 DFA 模拟状态转换，实现多个函数来识别不同类型的 token，并将识别出的 token 及相应的字符串分别添加到 tokens 和 values 中。

- isIdentifier(string s): 识别标识符。
- isInteger(string s): 识别整数。
- isDecimal(string s): 识别小数。
- isOperator(string s): 识别运算符。
- isDelimiter(string s): 识别分隔符。
- isKeyword(string s): 识别关键字。

错误处理：analysisFail() 方法用于处理分析过程中未能识别的字符或 token，输出错误信息并终止程序。

1.7.2 核心分析设置

1. analyse 函数

该函数对输入字符串进行词法分析，直到遇到终结符 #。

在实际的分析中，由于 Integer 和 Decimal 可能具有左公共因子，也就是 Decimal 的终态是在 Integer 的终态上进行扩展的。根据最长匹配原则，分析器应该优先判断 Decimal，然后再判断 Integer。同理，Keyword 的判断也应该在 Identifier 之前。

```
1    void analyse(string s){
2        while(curChar() != '#'){ // 当没读取到终结符时，说明字符串未分析完
3 //continue 是为了避免多个 token 间无空格只识别出第一个 token 就返回的情况。
4            if(isKeyword(s))         continue;
5            else if(isIdentifier(s)) continue;
6            else if(isDecimal(s))    continue;
7            else if(isInteger(s))    continue;
8            else if(isOperator(s))   continue;
9            else if(isDelimiter(s))  continue;
10           else analysisFail();
11       }
12   }
```

2. analyseFile 函数

该函数从指定的文件中逐行读取代码内容，并对每行进行词法分析。

首先使用 ifstream 对文件进行读取，通过 getline() 函数逐行读取文件内容。对于每一行，使用 istringstream 将其转换为可逐个读取的字符串流。使用流提取操作符 (») 逐个读取每个分割出来的字符串，并且调用 analyse 函数进行词法分析。当读取到注释符

号 // 时，获取注释内容并将其视为 token 即 <Annotation>，然后存入 tokens 和 values 容器中。

```
1    void analyseFile(const string& filePath){ //分析代码文件
2        LexicalAnalyzer la;
3        string line;
4
5        ifstream inputFile(filePath);
6        if (!inputFile.is_open()) {
7            cout << "无法打开文件: " << filePath << endl;
8            return;
9        }
10
11       while (getline(inputFile, line)) { // 逐行读取
12           istringstream iss(line);
13           string word;
14
15           // 逐个读取 word，包括空格，对每个 word 进行词法分析
16           while (iss >> word) {
17               if(word == "//"){ //如果读取到注释符号，将注释部分赋给 word，
   因为字符流会默认跳过注释内容
18                   size_t commentPos = line.find("//");
19                   word = string(line.substr(commentPos));
20                   la.tokens.clear();
21                   la.values.clear();
22                   la.tokens.push_back("<Annotation>");
23                   la.values.push_back(word);
24                   la.printTokens();
25                   break;
26               }
27               else{
28               // 处理 token，调用词法分析器进行分析
29                   la.init(word);
30                   la.analyse(word);
31                   la.printTokens();
32               }
33           }
34       }
35       inputFile.close();
36   }
```

1.8 实验验证

使用的测试样例代码存放在 test.txt, 具体内容如下：

```
int x = 10;
int x = 0;
float a=0.0;
int y = 5.28;
int function = y >x; // This is annotation test;
string str = "Hello world";
for (int i = 0; i < 10; i=i+1){
    x = x+1;
}
cout<<"Hello world"<<endl;
// annotation 注释
```

在 main 函数中使用类 LexicalAnalyzer 调用 analyseFile 函数进行词法分析，输出结果如下：

```
<Keyword> int
<Identifier> x
<Operator> =
<Integer> 10
<Delimiter> ;
<Keyword> int
<Identifier> x
<Operator> =
<Integer> 0
<Delimiter> ;
<Keyword> float
<Identifier> a
<Operator> =
<Decimal> 0.0
<Delimiter> ;
<Keyword> int
<Identifier> y
<Operator> =
```

```
    <Decimal> 5.28
    <Delimiter> ;
    <Keyword> int
    <Identifier> function
    <Operator> =
    <Identifier> y
    <Operator> >
    <Identifier> x
    <Delimiter> ;
    <Annotation> // This is annotation test;
    <Keyword> string
    <Identifier> str
    <Operator> =
    <Delimiter> "
    <Identifier> Hello
    <Identifier> world
    <Delimiter> "
    <Delimiter> ;
    <Keyword> for
    <Delimiter> (
    <Identifier> int
    <Identifier> i
    <Operator> =
    <Integer> 0
    <Delimiter> ;
    <Identifier> i
    <Operator> <
    <Integer> 10
    <Delimiter> ;
    <Identifier> i
    <Operator> =
    <Identifier> i
    <Operator> +
    <Integer> 1
    <Delimiter> )
    <Delimiter> {
    <Identifier> x
    <Operator> =
    <Identifier> x
    <Operator> +
```

```
59      <Integer> 1
60      <Delimiter> ;
61      <Delimiter> }
62      <Identifier> cout
63      <Operator> <
64      <Operator> <
65      <Delimiter> "
66      <Identifier> Hello
67      <Identifier> world
68      <Delimiter> "
69      <Operator> <
70      <Operator> <
71      <Identifier> endl
72      <Delimiter> ;
73      <Annotation> // annotation 注释
```

可以看出，词法分析器成功识别出了所有的 token，并输出了其对应的字符串。结果符合预期。

1.9 问题与解决

本实验中遇到的主要问题是 RE 的定义和相应 DFA 的构造，以及状态转换图的绘制。由于前期考虑的不够细致，定义 RE 的规则不够严谨，导致 DFA 的构造出现了不少问题。但之后重新梳理并思考，并结合 Thompson's construction 的原理，逐步完善了 RE 的定义，并成功构造出了相应的 DFA。

1.10 实验体会

在定义好的 RE 和构造正确的 DFA 的基础上，实现词法分析器的思路也比较清晰，通过模拟状态转换，识别出不同的 token 即可。这次实验让我对词法分析的原理、构造、分析过程等有了更深入的理解。

2 Syntax Parser Programming

2.1 实验目的

- 理解语法分析的整体过程。
- 对 LL 语法分析要熟练掌握消除左递归、消除左公共因子、得到 First 和 Follow 集合以及构造解析表的全过程。
- 对 LR 语法分析要熟练掌握构造 NFA、转换为 DFA，以及构造解析表的过程。
- 实现 LL 或 LR 语法分析器。

2.2 实验内容

- 定义使用的文法。
- 输入字符串；如果是自上而下分析，输出系列推导式；如果是自下而上分析，输出系列归约式。
- 实现语法分析过程的错误处理。

2.3 实验思路

本实验我选择构造 LL(1) 语法分析器。首先要自己定义一段文法，消除二义性、左递归、左公共因子后，得到 First 和 Follow 集合，构造解析表 parsing table。然后就是基于 parsing table 进行编程。

2.4 设定文法

本实验设定消除二义性、左递归、左公共因子后的文法如下，可以分析四则运算表达式。

```
1    E -> T M
2    M -> + T M | - T M | ϵ
3    T -> F N
4    N -> * F N | / F N | ϵ
5    F -> ( E ) | id
```

其中非终结符包括：E、M、T、N、F；终结符包括：id、+、-、*、/、(、)、ϵ。

2.5 构造 First 集、Follow 集及分析表

2.5.1 First 集和 Follow 集

由 First 集和 Follow 集构造算法可以得到表 1：

	First 集	Follow 集
`E -> T M`	(, id	) , $
`M -> + T M`	+	) , $
`M -> - T M`	-	) , $
`M -> ϵ`	ϵ	) , $
`T -> F N`	(, id	+ , - ,) , $
`N -> * F N`	*	+ , - ,) , $
`N -> / F N`	/	+ , - ,) , $
`N -> ϵ`	ϵ	+ , - ,) , $
`F -> ( E )`	(	+ , - , * , / ,) , $
`F -> id`	id	+ , - , * , / ,) , $

表 1: 文法的 First 集和 Follow 集

2.5.2 分析表

由表 1，可以构造分析表 parsing table 如下：

	id	+	-	*	/	(	)	$
E	T M					T M		
M		+ T M	- T M				ϵ	ϵ
T	F N					F N		
N		ϵ	ϵ	* F N	/ F N		ϵ	ϵ
F	id					(E)		

表 2: parsing table

2.6 重要数据结构描述

主要使用栈进行自上而下分析，初始化先后推入 $ 终结符和非终结符 E，然后比较栈顶和输入字符，根据分析表进行弹出和推入等操作。

终结符和非终结符采用一维字符数组存储，分析表采用二维字符串数组存储，如图 7：

```
char VN[M] = {'E', 'M', 'T', 'N', 'F'}; //非终结符
char VT[N] = {'i', '+', '-', '*', '/', '(', ')', '$'};//终结符
stack<char> analyse_stack; //分析栈
string input; //输入串,设定为全局变量
int i ; //输入串指针
string PT[M][N] = { //构建分析表 parsing table
    {"E->TM", "Error", "Error", "Error", "Error", "E->TM", "Error", "Error"},
    {"Error", "M->+TM", "M->-TM", "Error", "Error", "Error", "M->NULL","M->NULL"},
    {"T->FN", "Error", "Error", "Error", "Error", "T->FN", "Error","Error"},
    {"Error", "N->NULL","N->NULL", "N->*FN", "N->/FN", "Error", "N->NULL","N->NULL"},
    {"F->i", "Error", "Error", "Error","Error", "F->(E)", "Error","Error"}
};
```

图 7: 数据结构

2.7 核心算法

核心算法是将栈顶符号与输入字符串当前分析字符进行比较，如果匹配成功，则弹出栈顶符号，字符串指针前进一位；如果匹配失败（也就是栈顶符号是非终结符），则根据分析表获取推导的产生式，将产生式的右部符号从右至左压入栈中，重新与输入字符串比较。直到栈底的 $ 与输入字符串的末尾符号 $ 匹配成功，则说明语义解析成功；否则中间出现错误，报错。

```
1  void LL1_parser(){
2      init(); //初始化栈、指针
3      //具体分析过程
4      for(int n=1; i < input.length(); n++) { //n 代表当前的分析轮数
5          char top = analyse_stack.top(); //栈顶符号
6          char cur_input = input[i]; //当前输入符号
7          if(top == cur_input){
8              analyse_stack.pop(); //匹配成功，栈顶符号出栈
9              i++;
10             cout << setw(10) << n;
11             cout << setw(20) << printStack(analyse_stack);
12             cout << setw(20) << printInput(input, i);
13             cout << setw(20) << "" << top << "匹配成功" << endl;
14         }
15         else if(findVN(top)!=-1){
16             int vn_index = findVN(top); //栈顶非终结符的索引
17             int vt_index = findVT(cur_input); //当前输入符号的索引
18             // 处理特殊情况，无法识别的终结符
19             if (vt_index == -1) {
20                 cout << "Error: Unexpected character '" << cur_input << "'" <<
   endl;
21                 return;
22             }
```

```
23              string production = PT[vn_index][vt_index]; //根据分析表获取产生式
24              if (production == "Error") { //产生式不存在无法分析
25                  cout << "Error: No production found for VN '" << top << "' and VT '" << cur_input << "'" << endl;
26                  return;
27              }
28              if(production.substr(3) == "NULL"){  //产生式右部为空
29                  //cout<<production<<endl;
30                  analyse_stack.pop(); //因为是空产生式，直接弹出栈顶的非终结符
31
32                  cout << setw(10) << n;
33                  cout << setw(20) << printStack(analyse_stack);
34                  cout << setw(20) << printInput(input, i);
35                  cout << setw(20) << production << "推导空串" << endl;
36                  continue;
37              }
38              else{
39              //否则有对应的产生式
40                  analyse_stack.pop(); //栈顶非终结符出栈
41
42                  // 从产生式的右部开始（跳过左部和箭头，长度为 3）
43                  for (int j = production.length() - 1; j >= 3; j--) {
44                      analyse_stack.push(production[j]);
45                  }
46                  cout << setw(10) << n;
47                  cout << setw(20) << printStack(analyse_stack);
48                  cout << setw(20) << printInput(input, i);
49                  cout << setw(20) << production << "推导" << endl;
50              }
51          }
52          else {
53              cout << "Error: Unexpected stack top '" << top << "'" << endl;
54              return;
55          }
56      }
57      cout<<"分析成功！"<<endl; //分析结束
58  }
```

2.8 实验结果

编译源文件后在终端进行测试，结果如图 8：

```
PS D:\VScode\C_Project\VSCODE\Compile_lab\lab2> ./LL1.exe
请输入待分析的表达式：(i-i)/i+i*i
输入的表达式为(i-i)/i+i*i
--------------------------------分析过程--------------------------------------
步骤      栈内              输入串              产生式          动作
------------------------------------------------------------------------------
初态      $ E               (i-i)/i+i*i$                        初始化
1         $ M T             (i-i)/i+i*i$        E->TM           推导
2         $ M N F           (i-i)/i+i*i$        T->FN           推导
3         $ M N ) E (       (i-i)/i+i*i$        F->(E)          推导
4         $ M N ) E         i-i)/i+i*i$                         (匹配成功
5         $ M N ) M T       i-i)/i+i*i$         E->TM           推导
6         $ M N ) M N F     i-i)/i+i*i$         T->FN           推导
7         $ M N ) M N i     i-i)/i+i*i$         F->i            推导
8         $ M N ) M N       -i)/i+i*i$                          i匹配成功
9         $ M N ) M         -i)/i+i*i$          N->NULL         推导空串
10        $ M N ) M T -     -i)/i+i*i$          M->-TM          推导
11        $ M N ) M T       i)/i+i*i$                           -匹配成功
12        $ M N ) M N F     i)/i+i*i$           T->FN           推导
13        $ M N ) M N i     i)/i+i*i$           F->i            推导
14        $ M N ) M N       )/i+i*i$                            i匹配成功
15        $ M N ) M         )/i+i*i$            N->NULL         推导空串
16        $ M N )           )/i+i*i$            M->NULL         推导空串
17        $ M N             /i+i*i$                             )匹配成功
18        $ M N F /         /i+i*i$             N->/FN          推导
19        $ M N F           i+i*i$                              /匹配成功
20        $ M N i           i+i*i$              F->i            推导
21        $ M N             +i*i$                               i匹配成功
22        $ M               +i*i$               N->NULL         推导空串
23        $ M T +           +i*i$               M->+TM          推导
24        $ M T             i*i$                                +匹配成功
25        $ M N F           i*i$                T->FN           推导
26        $ M N i           i*i$                F->i            推导
27        $ M N             *i$                                 i匹配成功
28        $ M N F *         *i$                 N->*FN          推导
29        $ M N F           i$                                  *匹配成功
30        $ M N i           i$                  F->i            推导
31        $ M N             $                                   i匹配成功
32        $ M               $                   N->NULL         推导空串
33        $                 $                   M->NULL         推导空串
34                                                              $匹配成功
分析成功！
```

图 8: LL(1) 分析结果

可见，LL(1) 分析器成功识别出四则运算表达式，并输出了系列推导式，结果符合预期。而对于不符合文法的输入，LL(1) 分析器会报错，如图 9：

```
PS D:\VScode\C_Project\VSCODE\Compile_lab\lab2> ./LL1.exe
请输入待分析的表达式：i++i
输入的表达式为i++i
--------------------------------分析过程--------------------------------------
步骤      栈内              输入串              产生式          动作
------------------------------------------------------------------------------
初态      $ E               i++i$                               初始化
1         $ M T             i++i$               E->TM           推导
2         $ M N F           i++i$               T->FN           推导
3         $ M N i           i++i$               F->i            推导
4         $ M N             ++i$                                i匹配成功
5         $ M               ++i$                N->NULL         推导空串
6         $ M T +           ++i$                M->+TM          推导
7         $ M T             +i$                                 +匹配成功
Error: No production found for VN 'T' and VT '+'
PS D:\VScode\C_Project\VSCODE\Compile_lab\lab2> ./LL1.exe
请输入待分析的表达式：i>=i
输入的表达式为i>=i
--------------------------------分析过程--------------------------------------
步骤      栈内              输入串              产生式          动作
------------------------------------------------------------------------------
初态      $ E               i>=i$                               初始化
1         $ M T             i>=i$               E->TM           推导
2         $ M N F           i>=i$               T->FN           推导
3         $ M N i           i>=i$               F->i            推导
4         $ M N             >=i$                                i匹配成功
Error: Unexpected character '>'
```

图 9: LL(1) 分析器报错

2.9 问题与解决

主要问题集中在文法定义，以及构造 First 集、Follow 集及分析表的过程。在这些完成后通过解析表可以很好地模拟 LL(1) 分析过程中栈的变化，从而实现 LL(1) 分析器的功能。

2.10 实验体会

本次实验主要是动手实现了 LL(1) 分析器，让我对语法分析尤其是 LL(1) 分析的原理、First 集、Follow 集和分析表构造、分析过程等有了更深入的理解。