Compilers Principles

编译原理实验报告

作者：09020334 黄锦峰

时间：November, 2022

东南大学计算机科学与工程学院

实验一：词法分析器实验报告

1 实验目的

程序的编译是从输入源程序到输出目标代码，其中，词法分析是编译的第一个阶段，是对编译程序至关重要。词法分析就是扫描源程序字符串，按词法规则识别正确的单词，并转换成统一规格交语法分析使用。通过自己编程实现词法分析，我们能更深入地理解词法分析的过程，并提高自己的编程能力。

2 实验内容

1. 输入：字符流，正则表达式
2. 输出：标记序列
3. 单词的类由你自己定义
4. 可以包括错误处理

3 实验原理

词法分析是从左向右一个字符一个字符地读入源程序，扫描每行源程序的符号，依据词法规则，识别单词。执行词法分析的程序称为词法分析器，将给定的程序通过词法分析器，识别出一个个单词符号。

4 实验步骤概述

1. 词法的产生式（或 regular expression）转换成 NFA
2. 将这些 NFA 合并为单个 NFA
3. NFA 确定化为 DFA
4. DFA 最小化
5. 根据最小化后的 DFA 写出词法分析程序。
6. 用该语言写几个小程序，测试词法分析程序是否正确。

5 实验设计

5.1 符号约定

1. 关键字 (keyword)：program, begin, end,if, else, while, do,return,int,float,double,char,bool,void,read,write
2. 标识符 (identifier)：由字母开头，字母和数字组成
3. 常量 (constant)：整数、实数、true、false
4. 运算符 (operator)：+ − ∗ / = < > += −= ∗= /= <= >= == != && || !
5. 分隔符 (delimiter)：, ; () [] {}

表 1: 符号约定

关键字（Keyword）	常量（Constant）	运算符（Operator）
program	整数	+
begin	小数	-
end	true	*
if	false	/
else		=
while	**分割符（Delimiter）**	<
do	,	>
return	;	!
int	(	+=
float	)	-=
double	[	*=
char	]	/=
bool	{	==
void	}	<=
read		>=
write		+ +
		- -
		!=
		&&
		||

5.1.1 关键字 (keyword)

正则表达式：

$$keyword \rightarrow letter(letter)^*$$
$$letter \rightarrow [a-z]$$

5.1.2 标识符 (identifier)

正则表达式：

$$identifier \rightarrow (letter)(letter|digit)^*$$
$$letter \rightarrow [a-zA-Z]$$
$$digit \rightarrow [0-9]$$

5.1.3 常量 (constant)

这里约定为小数、整数和布尔值 true、false

正则表达式：

$$constant \rightarrow (digit)digit^*(\varepsilon|.digit^+)$$

5.1.4 运算符 (operator)

正则表达式：

$$operator \rightarrow (+|-|*|/|=|<|>|\&\&|\|\||!|+=|-=|*=|/=|<=|>=|==|!=)$$

5.1.5 分隔符 (delimiter)

正则表达式：

$$delimiter \rightarrow (\ , \ | \ ; \ | \ (\ | \ ) \ | \ [\ | \ ] \ | \ \{ \ | \ \} \ )$$

5.2 状态图分析

根据上述正则表达式，可以得到如下状态图，

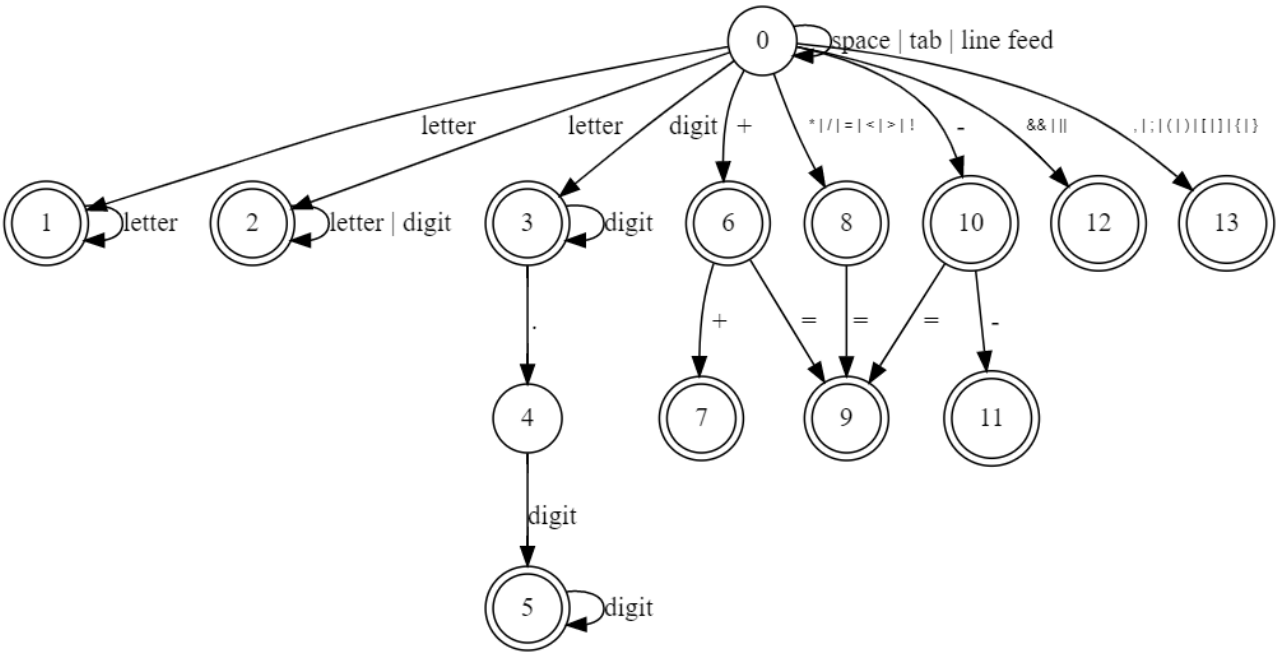


图 1: NFA

- 0 为初态
- 1 为关键字或标识符
- 2 为标识符
- 3,5 为常量
- 6,7,8,9,10,11,12 为运算符
- 13 为分隔符

5.3 算法流程图

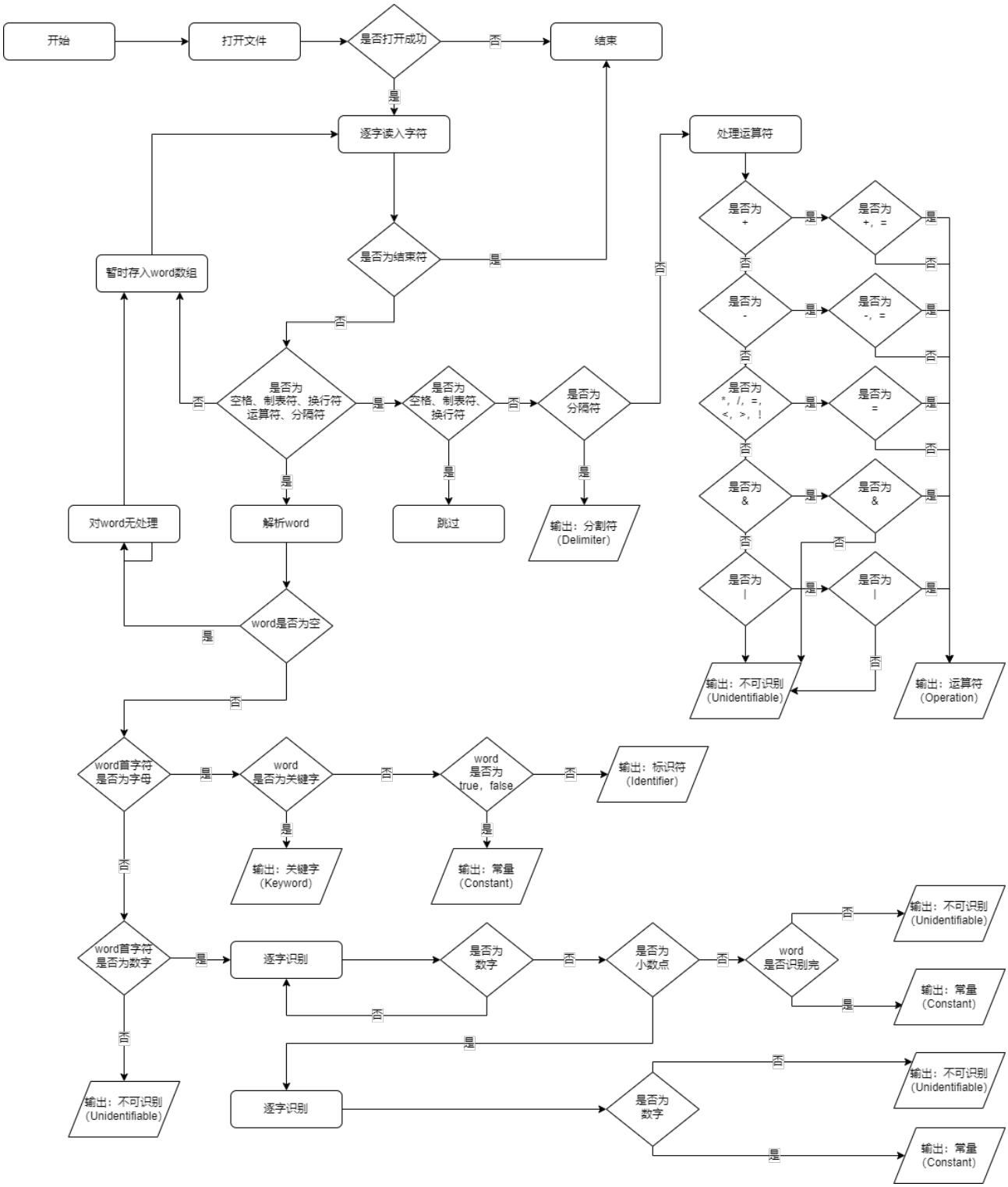


图 2: 算法流程图

6 完整代码

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <iomanip>
using namespace std;
ifstream inFile; //输入文件
int errorNum = 0; //错误数
//所有字符种类（关键字、标识符、常量、运算符、分隔符、不可识别）
const string token[6] = {"Keyword", "Identifier", "Constant", "Operation", "Delimiter", "
    Unidentifiable"};
//定义一个输出函数，输出已经分析好的词法分析结果
void output(string str, int id)
{
    cout << "<" << setw(10) << str << setw(5) << "," << setw(15) << token[id] << setw(5) << ">\n";
    if (id == 5)
        errorNum++;
}

//判断字符是否为字母
bool isLetter(char ch)
{
    if ((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z'))
        return true;
    else
        return false;
}

//判断字符是否为数字
bool isDigit(char ch)
{
    if (ch >= '0' && ch <= '9')
        return true;
    else
        return false;
}

//是否是空格，回车，制表符
bool isUesless(char ch)
{
    if (ch == ' ' || ch == '\0' || ch == '\n' || ch == '\t')
        return true;
    else
        return false;
}
```

```
//关键字
const string keywords[16] = {"program", "begin", "end", "if", "else", "while", "do", "return", "int",
    "float", "double", "char", "bool", "void", "read", "write"};
//运算符
const char operation[] = {'+', '-', '*', '/', '=', '>', '<', '!', '&', '|'};
//{"+", "-", "*", "/", "=", ">", "<", "!", "+=", "-=", "*=", "/=", "==", ">=", "<=", "!=", "&&",
    "||","++","--"};
//分隔符
const char delimiter[] = {',', ';', '(', ')', '{', '}', '[', ']'};

//判断是否是关键字
bool isKeyword(string str)
{
    for (int i = 0; i < 16; i++)
    {
        if (keywords[i] == str)
        {
            return true;
        }
    }
    return false;
}

//判断是否是运算符
bool isOperation(char ch)
{

    for (int i = 0; i < 10; i++)
    {
        if (operation[i] == ch)
            return true;
    }
    return false;
}

//判断是否是分隔符
bool isDelimiter(char ch)
{
    for (int i = 0; i < 8; i++)
    {
        if (delimiter[i] == ch)
            return true;
    }
    return false;
}

//首字符为字母的识别
void onLetter(string str)
{
```

```
    //检查str中是否有非法字符
    for (int i = 0; i < str.size(); i++)
    {
        if (!isLetter(str[i]) && !isDigit(str[i]))
        {
            output(str, 5);
            return;
        }
    }
    // 如果是 true false
    if (str == "true" || str == "false")
    {
        output(str, 2);
        return;
    }
    //判断是否是关键字
    if (isKeyword(str))
    {
        output(str, 0);
    }
    else
    {
        output(str, 1);
    }
}

//首字符为数字的识别
void onDigit(string str)
{
    int pos = 0;
    while (isDigit(str[pos]))
    {
        pos++;
    }
    if (str[pos] == '.')
    {
        pos++;
        while (isDigit(str[pos]))
        {
            pos++;
        }
        if (pos == str.size())
        {
            output(str, 2);
        }
        else
        {
            output(str, 5);
        }
```

```
    }
    else
    {
        if (pos == str.size())
        {
            output(str, 2);
        }
        else
        {
            output(str, 5);
        }
    }
}

//处理word
void solveWord(string str)
{
    // str 的第一个字符是字母,onLetter
    if (isLetter(str[0]))
    {
        onLetter(str);
    }
    // str 的第一个字符是数字,onDigit
    else if (isDigit(str[0]))
    {
        onDigit(str);
    }
    //不是字母也不是数字，不可识别
    else
    {
        output(str, 5);
    }
}

//处理运算符
char solveOpration(char ch)
{
    char str[3];
    switch (ch)
    {
    case '+':
        str[0] = ch;
        ch = inFile.get();
        if (ch == '=' || ch == '+')
        {
            str[1] = ch;
            str[2] = '\0';
            output(str, 3);
            ch = inFile.get();
```

```
            return ch;
        }
        else
        {
            str[1] = '\0';
            output(str, 3);
            return ch;
        }
        break;
    case '-':
        str[0] = ch;
        ch = inFile.get();
        if (ch == '=' || ch == '-')
        {
            str[1] = ch;
            str[2] = '\0';
            output(str, 3);
            ch = inFile.get();
            return ch;
        }
        else
        {
            str[1] = '\0';
            output(str, 3);
            return ch;
        }
        break;
    case '*':
    case '/':
    case '=':
    case '<':
    case '>':
    case '!':
        str[0] = ch;
        ch = inFile.get();
        if (ch == '=')
        {
            str[1] = ch;
            str[2] = '\0';
            output(str, 3);
            ch = inFile.get();
            return ch;
        }
        else
        {
            str[1] = '\0';
            output(str, 3);
            return ch;
        }
```

```
        break;
    case '&':
        str[0] = ch;
        ch = inFile.get();
        if (ch == '&')
        {
            str[1] = ch;
            str[2] = '\0';
            output(str, 3);
            ch = inFile.get();
            return ch;
        }
        else
        {
            str[1] = '\0';
            output(str, 5);
            return ch;
        }
        break;
    case '|':
        str[0] = ch;
        ch = inFile.get();
        if (ch == '|')
        {
            str[1] = ch;
            str[2] = '\0';
            output(str, 3);
            ch = inFile.get();
            return ch;
        }
        else
        {
            str[1] = '\0';
            output(str, 5);
            return ch;
        }
        break;
    default:
        //好像没必要
        str[0] = ch;
        str[1] = '\0';
        output(str, 5);
        ch = inFile.get();
        return ch;
        break;
    }
}

int main()
```

```
{
    string filename;
    cout << "Please enter the path and name of the source file: (default to the test.txt file in the
        same directory)" << endl;
    //点击回车键，使用默认文件
    getline(cin, filename);
    // 如果filename为空,则默认为test.txt
    if (filename.empty())
    {
        filename = "test.txt";
    }
    //输出文件名
    cout << "The file name is: " << filename << endl;
    // 打开文件
    inFile = ifstream(filename, ios::in);
    // 如果文件打开失败,则报错
    if (!inFile)
    {
        cerr << "Unable to open file! " << filename.c_str() << endl;
        exit(-1);
    }
    // 打印文件中的内容
    cout << endl;
    char all[1000];
    inFile.getline(all, 1000, EOF);
    cout << "测试用例如下：\n"
        << all << endl;

    // 词法分析
    inFile = ifstream("test.txt", ios::in);
    //设置一个字符数组，用来存储读取的字符
    char word[100];
    char ch;
    int p = 0;
    ch = inFile.get();
    cout << "\n词法分析测试结果如下：\n"
        << "<" << setw(10) << "单词本身" << setw(5) << "," << setw(15) << "所属类别" << setw(5) << ">\n
            ";
    while (ch != EOF)
    {
        //没有遇到空格，回车，制表符，运算符，分隔符，就继续读取字符，并存入word数组中处理
        if (!(isUesless(ch) || isOperation(ch) || isDelimiter(ch)))
        {
            word[p++] = ch;
            ch = inFile.get();
        }
        else
        {
            //遇到空格，回车，制表符，运算符，分隔符，就将word数组中的字符输出
```

```
            //处理一下word数组，将p置为0
            if (p != 0)
            {
                word[p] = '\0';
                solveWord(word);
                p = 0;
            }
            if (isUesless(ch))
            {
                ch = inFile.get();
            }
            if (isOperation(ch))
            {
                ch = solveOpration(ch);
            }
            if (isDelimiter(ch))
            {
                char str[2];
                str[0] = ch;
                str[1] = '\0';
                output(str, 4);
                ch = inFile.get();
            }
        }
    }
    if (p != 0)
    {
        word[p] = '\0';
        solveWord(word);
        p = 0;
    }
    cout << "\n词法分析成功! \n共\t" << errorNum << "\t个无法识别的单词。" << endl;
    inFile.close();
    return 0;
}
```

7 运行样例

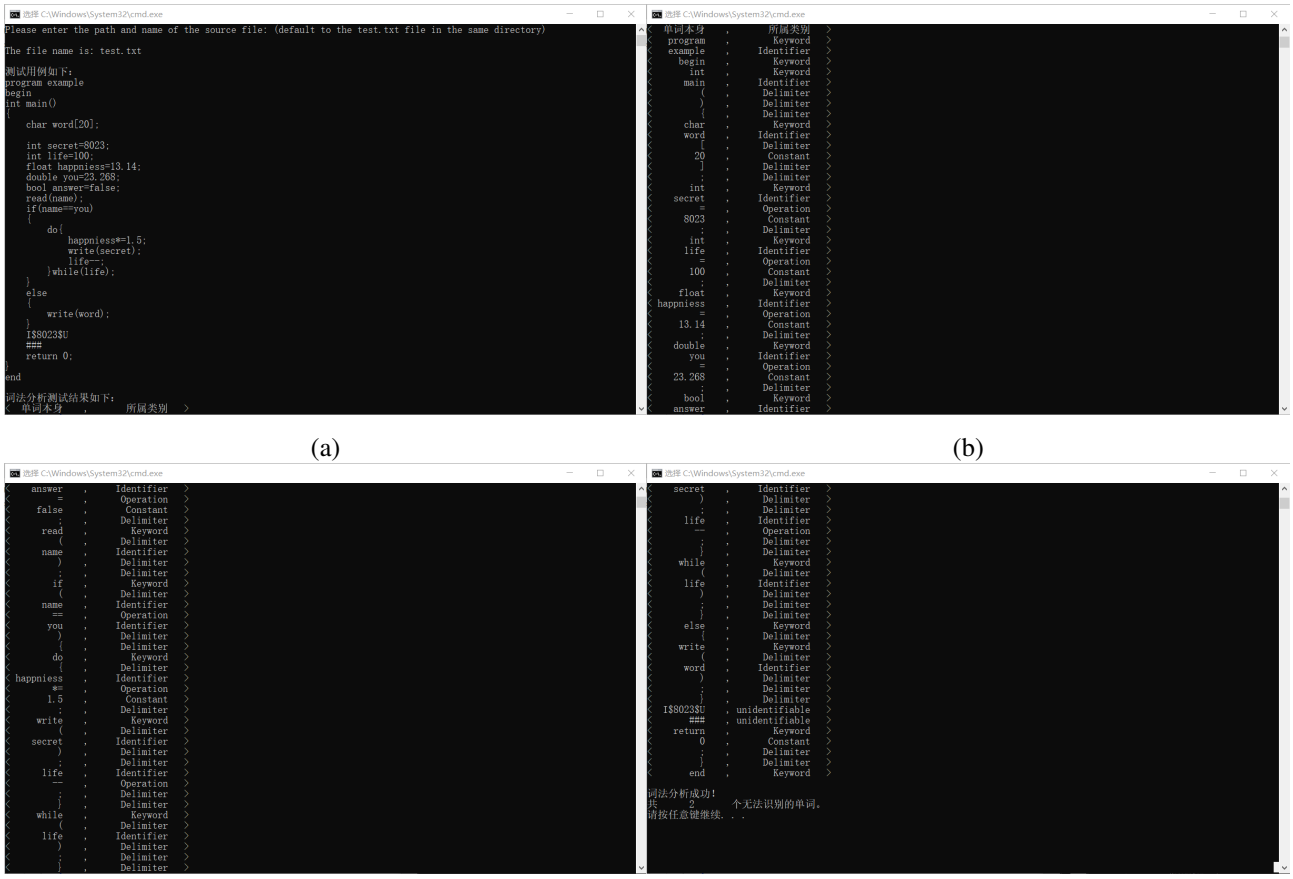
与程序文件在同一路径下的 test.txt 文件，内容如下：

7.1 输入样例

```
program example
begin
int main()
{
    char word[20];

    int secret=8023;
    int life=100;
    float happniess=13.14;
    double you=23.268;
    bool answer=false;
    read(name);
    if(name==you)
    {
        do{
            happniess*=1.5;
            write(secret);
            life--;
        }while(life);
    }
    else
    {
        write(word);
    }
    I$8023$U
    ###
    return 0;
}
end
```

7.2 输出样例



(a) (b)

(c) (d)

图 3: 输出样例

8 实验中出现的问题与相应的解决方法

8.1 错误检测问题

检测代码中出现这样一串字符时

```
I$8023$U
###
```

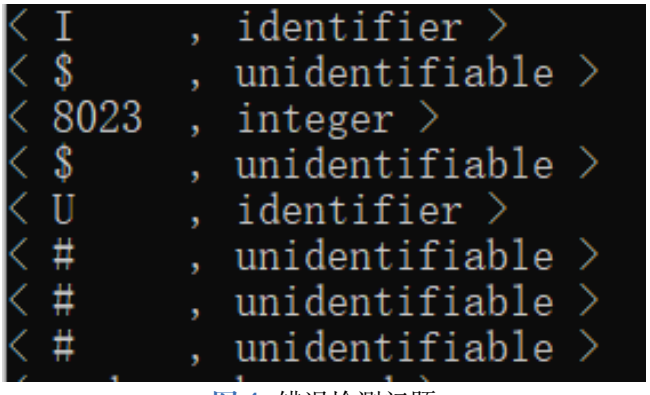


图 4: 错误检测问题

发现依照一个字符一个字符的读取、识别、跳转，不满足预期的要求，因此需要将读取的字符存入一个数组中，对整词进行判断然，这样就可以解决这个问题，输出错误单词。

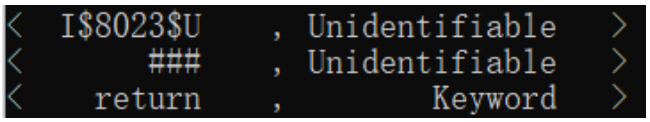


图 5: 修正错误检测

8.2 运算符识别问题

识别下列运算符

```
{"+", "-", "*", "/", "=", ">", "<", "!", "+=", "-=", "*=", "/=", "==", ">=", "<=", "!=", "&&", "||
    ","++","--"};
```

有一个字符和两个字符的运算符，有些符号之间有所联系，因此需要对读取的字符进行判断。如果是一个字符的运算符，检验下一个字符是否是满足要求的运算符，如果是，将两个字符合并成一个运算符，否则，将一个字符作为一个运算符。

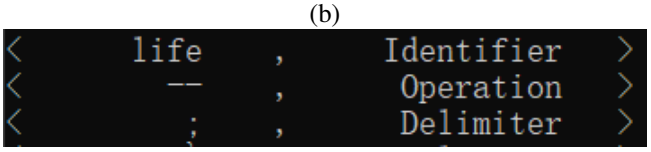
代码详见

```
//处理运算符
char solveOpration(char ch){}
```

效果如下：

(a)

(b)



(c)

图 6: 运算符识别

8.3 绘制 DFA

在有限状态自动机较为复杂时，利用以下编程自动绘制有限状态自动机的网站能够比较好的解决这个问题。

https://dreampuf.github.io/GraphvizOnline/

本次实验中的有限状态自动机绘制的源代码如下：

```
digraph finite_state_machine {
//rankdir = LR;
// size = "8,5"
 node [shape = doublecircle];
 1 2 3 5 6 7 8 9 10 11 12 13;
```

```
node [shape = circle];
// keyword
0 -> 0 [ label = "space | tab | line feed" ];
0 -> 1 [ label = "letter" ];
1 -> 1 [ label = "letter" ];
//identifie
0 -> 2 [ label = "letter" ];
2 -> 2 [ label = "letter | digit" ];
//constant
0 -> 3 [ label = "digit"];
3 -> 3 [ label = "digit"];
3 -> 4 [ label = " ."];
4 -> 5 [ label = "digit"];
5 -> 5 [ label = "digit"];
//operator
0 -> 6 [ label = " + "];
6 -> 7 [ label = " + "];
0 -> 8 [ label = " * | / | = | < | > | dž ",fontname="Arial", fontsize=8, labelangle=100];
0 -> 10 [ label = " - "];
10 -> 11 [ label = " - "];
6 -> 9 [ label= " = "]
8 -> 9 [ label= " = "]
10 -> 9 [ label= " = "]
0 -> 12 [ label = "&& | || ",fontname="Arial", fontsize=8, labelangle=100];
//delimiter
0 -> 13 [ label = ", | ; | ( | ) | [ | ] | { | } ",fontname="Arial", fontsize=8, labelangle=100];
}
```

9 实验总结

在本次实验中，设计了一个简单的词法分析器，能够识别关键字、标识符、常量、运算符、分割符等，对词法的判定的分析和绘制有限状态自动机有了更加熟练的操作。

在错误检测中也发现了一些问题，并按照本人对一般代码编写的习惯进行了一些错误判定的改变读取文件做词法分析时更注重了“整词”的思想，争取遇见一个字符取一个词一个词进行分析。

通过本次实验，我对程序编译中词法分析这一阶段有了更深的理解，自身的编程能力与解决问题的能力也得到了提升。

实验二：语法分析器实验报告

1 实验目的

程序的编译是从输入源程序到输出目标代码，其中，语法分析是编译的第二个阶段，紧跟在词法分析之后，对编译程序至关重要。语法分析就是通过语法分解，确定词法分析得到的“词”能否构成语法上正确的句子。通过自己编程实现语法分析，我们能更深入地理解语法分析的过程，并提高自己的编程能力。

2 实验内容

学生自选语言或某语言的子集，对该语言的代码进行语法分析。分析方法可以是 LL(1) 或者 LR(1)。要求：输入词法分析后得到的 token 序列，输出语法分析的过程。

1. 输入：字符流、CFG(某类句子的 CFG 组合)
2. 输出：语法分析的过程
   (a). 如果使用自顶向下的语法分析方法，则派生序列。
   (b). 如果使用自底向上语法分析方法，则简化顺序。
3. 句子类别由自己定义
4. 可以包括错误处理

3 实验原理

语法分析器是在词法分析之后，根据词法分析的结果和定义的语法规则判断输入的程序是否有语法错误，LL(1) 分析是使用显式栈而不是递归调用来完成分析。以标准方式表示这个栈非常有用，这样 LL(1) 分析程序的动作就可以快捷地显现出来。LL(1) 的含义是：第一个 L 表明自顶向下分析是从左向右扫描输入串，第 2 个 L 表明分析过程中将使用最左推导，1 表明只需向右看一个符号便可决定如何推导，即选择哪个产生式 (规则) 进行推导。

4 实验假设

本次实验实现 LL(1) 分析方法，目标语言与上次基本一致，在输入输出上做了一些简化，其余的语言定义均与上次一致。

5 实验设计

5.1 定义语法分析使用的文法语言：

在这里我们简单处理一下赋值语句

< 赋值语句 > ::= < 标识符 > = < 表达式 > ;

< 表达式 > ::= < 表达式 > + < 表达式 > | < 表达式 > * < 表达式 > | < 标识符 > | < 常量 >

原文法 $G = (V_N, V_T, P, S)$, 其中 $V_N = \{S, R, E\}$,$V_T = \{Identifier, Constant, (,), +, *, =\}$,

产生式 P：

$$S \to R = E;$$
$$R \to Identifier$$
$$E \to E + E|E * E|(E)|Identifier|Constant$$

消除二义性：

$$S \to R = E;$$
$$R \to Identifier$$
$$E \to E + T|D$$
$$T \to T * F|F$$
$$F \to (E)|Identifier|Constant$$

消除左递归：

$$S \to R = E;$$
$$R \to Identifier$$
$$E \to TD$$
$$D \to +TD|\varepsilon$$
$$T \to FH$$
$$H \to *FH|\varepsilon$$
$$F \to (E)|Identifier|Constant$$

求 FIRST 集和 FOLLOW 集：

表 2: FIRST 集和 FOLLOW 集

	First	Follw
$S \to R = E;$	Id	#
$R \to Identifier$	Id	=
$E \to TD$	(Id Const	;)
$D \to +TD$	+	;)
$D \to \varepsilon$	ε	
$T \to FH$	(Id Const	+ ;)
$H \to *FH$	*	+ ;)
$H \to \varepsilon$	ε	
$F \to (E)$	(	* + ;)
$F \to Identifier$	Id	
$F \to Constant$	Const	

其中 Id 表示 Identifier，Const 表示 Constant，ε 表示空串，# 表示结束符号。进而得到 LL(1) 分析表：

表 3: LL(1) 分析表

	Id	Const	+	*	(	)	;
S	$S \to R = E;$						
R	$R \to Identifier$						
E	$E \to TD$	$E \to TD$			$E \to TD$		
D			$D \to +TD$			$D \to \varepsilon$	$D \to \varepsilon$
T	$T \to FH$	$T \to FH$			$T \to FH$		
H			$H \to \varepsilon$	$H \to *FH$		$H \to \varepsilon$	$H \to \varepsilon$
F	$F \to Identifier$	$F \to Constant$			$F \to (E)$		

6 完整代码

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <iomanip>
#define MAXN 99
using namespace std;
ifstream inFile; //输入文件
int n;          // token序列的长度
struct Token   // token结构体
{
    string str;
    string type;
};
Token tokenStream[MAXN]; //读入数据存入其中
char stack[MAXN];     //栈
char top;             //栈顶字符
int tail, pos, step; //栈尾，输入串位置，步数
//初始化
void init()
{
    n = 0;
    string str_in, type_in;
    while (inFile >> str_in >> type_in)
    {
        tokenStream[n].str = str_in;
        tokenStream[n].type = type_in;
        n++;
    }
    tokenStream[n].str = "#";
    tokenStream[n].type = "END";
}
//输出
void show()
{
```

```
    cout << setw(5) << step << setw(10); //<< “步骤"
    for (int i = 0; i < tail; i++)  // << ": 栈内: ";
    {
        cout << stack[i];
    }
    cout << setw(10) << "\t";
    for (int i = pos; i <= n; i++)
    {
        cout << tokenStream[i].str;
    }
    cout << setw(10) << "\t";
}
//处理
void process()
{
    tail = 0;
    pos = 0, step = 0;
    stack[tail++] = '#';
    stack[tail++] = 'S';
    bool flag = true;
    while (flag)
    {
        show();
        top = stack[tail - 1];
        step++;
        switch (top)
        {
        case '#':
            if (tokenStream[pos].str == "#")
            {
                cout << "分析成功! \n";
                flag = false;
                break;
            }
        case 'S':
            if (tokenStream[pos].type == "Identifier")
            {
                cout << "推导 S->R=E; \n";
                tail--;
                stack[tail++] = ';';
                stack[tail++] = 'E';
                stack[tail++] = '=';
                stack[tail++] = 'R';
                break;
            }
        case 'R':
            if (tokenStream[pos].type == "Identifier")
            {
                cout << "推导 R->Identifier \n";
```

```
            tail--;
            stack[tail++] = 'i';
            break;
        }
    case 'E':
        if (tokenStream[pos].type == "Identifier" || tokenStream[pos].type == "Constant" ||
            tokenStream[pos].str == "(")
        {
            cout << "推导 E->TD \n";
            tail--;
            stack[tail++] = 'D';
            stack[tail++] = 'T';
            break;
        }
    case 'D':
        if (tokenStream[pos].str == "+")
        {
            cout << "推导 D->+TD \n";
            tail--;
            stack[tail++] = 'D';
            stack[tail++] = 'T';
            stack[tail++] = '+';
            break;
        }
        if (tokenStream[pos].str == ")" || tokenStream[pos].str == ";")
        {
            cout << "推导 D->epsilon \n";
            tail--;
            break;
        }
    case 'T':
        if (tokenStream[pos].type == "Identifier" || tokenStream[pos].type == "Constant" ||
            tokenStream[pos].str == "(")
        {
            cout << "推导 T->FH \n";
            tail--;
            stack[tail++] = 'H';
            stack[tail++] = 'F';
            break;
        }
    case 'H':
        if (tokenStream[pos].str == "*")
        {
            cout << "推导 H->*FH \n";
            tail--;
            stack[tail++] = 'H';
            stack[tail++] = 'F';
            stack[tail++] = '*';
            break;
```

```
        }
        if (tokenStream[pos].str == "+" || tokenStream[pos].str == ")" || tokenStream[pos].str == ";
            ")
        {
            cout << "推导 H->epsilon \n";
            tail--;
            break;
        }
    case 'F':
        if (tokenStream[pos].type == "Identifier")
        {
            cout << "推导 F->Identifier \n";
            tail--;
            stack[tail++] = 'i';
            break;
        }
        if (tokenStream[pos].type == "Constant")
        {
            cout << "推导 F->Constant \n";
            tail--;
            stack[tail++] = 'c';
            break;
        }
        if (tokenStream[pos].str == "(")
        {
            cout << "推导 F->(E) \n";
            tail--;
            stack[tail++] = ')';
            stack[tail++] = 'E';
            stack[tail++] = '(';
            break;
        }
    case 'i':
        if (tokenStream[pos].type == "Identifier")
        {
            cout << "匹配 Identifier \n";
            tail--;
            pos++;
            break;
        }
    case 'c':
        if (tokenStream[pos].type == "Constant")
        {
            cout << "匹配 Constant\n";
            tail--;
            pos++;
            break;
        }
    case '+':
```

```
        if (tokenStream[pos].str == "+")
        {
            cout << "匹配 + \n";
            tail--;
            pos++;
            break;
        }
    case '*':
        if (tokenStream[pos].str == "*")
        {
            cout << "匹配 * \n";
            tail--;
            pos++;
            break;
        }
    case '(':
        if (tokenStream[pos].str == "(")
        {
            cout << "匹配 ( \n";
            tail--;
            pos++;
            break;
        }
    case ')':
        if (tokenStream[pos].str == ")")
        {
            cout << "匹配 ) \n";
            tail--;
            pos++;
            break;
        }
    case '=':
        if (tokenStream[pos].str == "=")
        {
            cout << "匹配 = \n";
            tail--;
            pos++;
            break;
        }
    case ';':
        if (tokenStream[pos].str == ";")
        {
            cout << "匹配 ; \n";
            tail--;
            pos++;
            break;
        }
    default:
        cout << "ERROR!\n";
```

```
            flag = false;
            break;
        }
    }
}
int main()
{
    string filename;
    cout << "Please enter the path and name of the source file: (default to the test.txt file in the
        same directory)" << endl;
    //点击回车键，使用默认文件
    getline(cin, filename);
    // 如果filename为空,则默认为test.txt
    if (filename.empty())
    {
        filename = "test.txt";
    }
    //输出文件名
    cout << "The file name is: " << filename << endl;
    // 打开文件
    inFile = ifstream(filename, ios::in);
    // 如果文件打开失败,则报错
    if (!inFile)
    {
        cerr << "Unable to open file! " << filename.c_str() << endl;
        exit(-1);
    }
    //=================
    cout << "<" << 1 << ">"
         << "\t"
         << "S->R=E;" << endl;
    cout << "<" << 2 << ">"
         << "\t"
         << "R->Identifier" << endl;
    cout << "<" << 3 << ">"
         << "\t"
         << "E->TD" << endl;
    cout << "<" << 4 << ">"
         << "\t"
         << "D->+TD" << endl;
    cout << "<" << 5 << ">"
         << "\t"
         << "D->epsilon" << endl;
    cout << "<" << 6 << ">"
         << "\t"
         << "T->FH" << endl;
    cout << "<" << 7 << ">"
         << "\t"
         << "H->*FH" << endl;
```

```
    cout << "<" << 8 << ">"
         << "\t"
         << "H->epsilon" << endl;
    cout << "<" << 9 << ">"
         << "\t"
         << "F->(E)" << endl;
    cout << "<" << 10 << ">"
         << "\t"
         << "F->Identifier" << endl;
    cout << "<" << 11 << ">"
         << "\t"
         << "F->Constant" << endl;

    init();
    cout << "\n词法分析测试结果如下: \n"
         << setw(5) << "步骤" << setw(15) << "栈内"
         << setw(20) << "输入串" << setw(30) << "动作"
         << "\n";

    process();
    return 0;
}
```

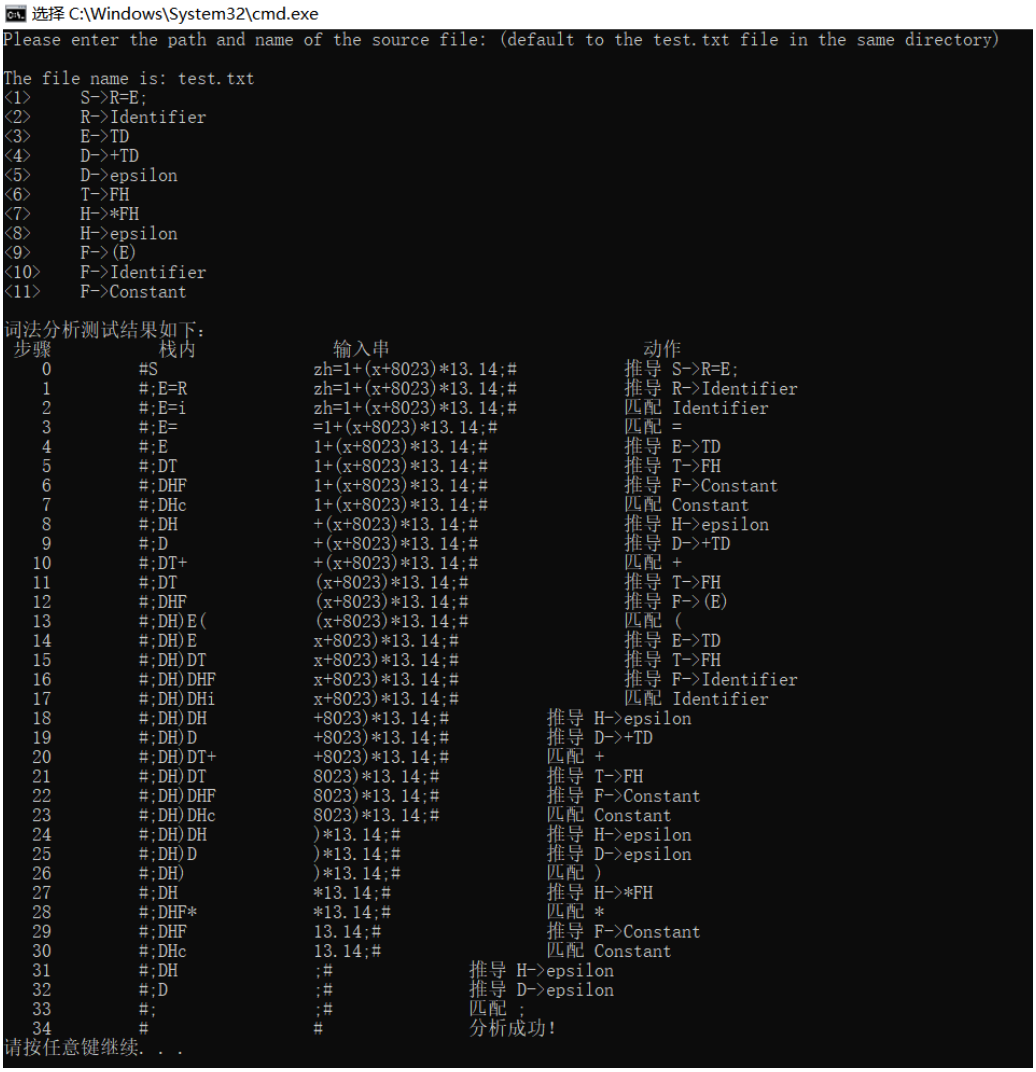
7 运行样例

7.1 输入样例

与程序文件在同一路径下的 test.txt 文件中存储了 token 序列:，表示 $zh = 1 + (x + 8023) * 13.14;$ 内容如下:

```
zh Identifier
= Operation
1 Constant
+ Operation
( Delimiter
x Identifier
+ Operation
8023 Constant
) Delimiter
* Operation
13.14 Constant
; Delimiter
```

7.2 输出样例



```
选择 C:\Windows\System32\cmd.exe
Please enter the path and name of the source file: (default to the test.txt file in the same directory)

The file name is: test.txt
<1>     S->R=E;
<2>     R->Identifier
<3>     E->TD
<4>     D->+TD
<5>     D->epsilon
<6>     T->FH
<7>     H->*FH
<8>     H->epsilon
<9>     F->(E)
<10>    F->Identifier
<11>    F->Constant

词法分析测试结果如下:
 步骤          栈内              输入串                          动作
    0          #S              zh=1+(x+8023)*13.14;#          推导 S->R=E;
    1          #;E=R           zh=1+(x+8023)*13.14;#          推导 R->Identifier
    2          #;E=i           zh=1+(x+8023)*13.14;#          匹配 Identifier
    3          #;E=            =1+(x+8023)*13.14;#            匹配 =
    4          #;E             1+(x+8023)*13.14;#             推导 E->TD
    5          #;DT            1+(x+8023)*13.14;#             推导 T->FH
    6          #;DHF           1+(x+8023)*13.14;#             推导 F->Constant
    7          #;DHc           1+(x+8023)*13.14;#             匹配 Constant
    8          #;DH            +(x+8023)*13.14;#              推导 H->epsilon
    9          #;D             +(x+8023)*13.14;#              推导 D->+TD
   10          #;DT+           +(x+8023)*13.14;#              匹配 +
   11          #;DT            (x+8023)*13.14;#               推导 T->FH
   12          #;DHF           (x+8023)*13.14;#               推导 F->(E)
   13          #;DH)E(         (x+8023)*13.14;#               匹配 (
   14          #;DH)E          x+8023)*13.14;#                推导 E->TD
   15          #;DH)DT         x+8023)*13.14;#                推导 T->FH
   16          #;DH)DHF        x+8023)*13.14;#                推导 F->Identifier
   17          #;DH)DHi        x+8023)*13.14;#                匹配 Identifier
   18          #;DH)DH         +8023)*13.14;#          推导 H->epsilon
   19          #;DH)D          +8023)*13.14;#          推导 D->+TD
   20          #;DH)DT+        +8023)*13.14;#          匹配 +
   21          #;DH)DT         8023)*13.14;#           推导 T->FH
   22          #;DH)DHF        8023)*13.14;#           推导 F->Constant
   23          #;DH)DHc        8023)*13.14;#           匹配 Constant
   24          #;DH)DH         )*13.14;#               推导 H->epsilon
   25          #;DH)D          )*13.14;#               推导 D->epsilon
   26          #;DH)           )*13.14;#               匹配 )
   27          #;DH            *13.14;#                推导 H->*FH
   28          #;DHF*          *13.14;#                匹配 *
   29          #;DHF           13.14;#                 推导 F->Constant
   30          #;DHc           13.14;#                 匹配 Constant
   31          #;DH            ;#              推导 H->epsilon
   32          #;D             ;#              推导 D->epsilon
   33          #;              ;#              匹配 ;
   34          #               #               分析成功!
请按任意键继续. . .
```

8 实验总结

本次实验使用 LL(1) 分析方法来对赋值语句所对应的 token 序列进行了分析。在编写程序之前，首先要对 LL(1) 文法进行分析，将其转化为 LL(1) 分析表，然后根据分析表来进行分析，最后输出分析过程。主要部分是根据分析表和当前栈中状态来判定下一步的动作，进行入栈或者出栈操作。

如若语句包含种类较多，如：

< 语句 > ::= < 变量定义语句 > | < 赋值语句 > | <if 语句 > | ε

会直接提升分析过程较为复杂度，也会产生一个很大的分析表，但在步骤和实现上没有本质差别。

通过本次实验，我对程序编译中语法分析这一阶段有了更深的理解，自身的编程能力与解决问题的能力也得到了提升