

深圳大学实验报告

课程名称： 编译原理

实验项目名称： 词法分析程序设计

学院： 计算机与软件学院

专业： 软件工程

指导教师： 蔡树彬

报告人： 郑彦薇 学号： 2020151022 班级： 软件工程 01 班

实验时间： 2023 年 3 月 21 日至 4 月 14 日

实验报告提交时间： 2023 年 4 月 13 日

教务处制

实验目的与要求:

目的: 通过实验, 加深对词法分析技术的理解并能进行应用。

要求:

第一部分: 基于 DFA 的关键字、运算符和分隔符识别

1. 参考你所使用的编程语言, 定义你所处理语言的关键字、分隔符和运算符
2. 采用“查表”和 DFA 两种方法, 对比两种方法的识别效率

第二部分: 基于 DFA 的标识符和无符号数识别

1. 标识符为字母打头, 后面是任意长度的字母或数字;
2. 无符号数可由整数部分、小数部分和指数部分构成; 整数、小数部分可单独出现也可依次出现, 指数部分需跟随在整数或小数后;
3. 给定一个字符串(源代码文件), 切分并输出里面的标识符和无符号数, 对无符号数, 计算并输出其数值(禁用数字串直接输出, 不要使用字符串转数值相关函数, 使用单个字符转数值的形式)

第三部分: 基于 DFA 的分词程序

输入文件: 普通源码文件, 如实验 1 的源代码; 空白、注释等按普通定义

输出文件: 除去输入文件中的空白、注释; 并切分、输出源程序中的单词。例如, 输入文件的内容是 “for(int i=0;i<5;i++) {sum +=i;} //这是有问题的循环” 输出是

```
(FOR 关键字, for)
(左括号, ()
(INT 关键字, int)
(标识符, i)
(赋值运算符, =)
(整数, 0)
(分号分隔符, ; )
(标识符, i)
(小于运算符, <)
(整数, 5)
(分号分隔符, ; )
(标识符, i)
(自增 1 运算符, ++ )
(右括号, ))
(左花括号, {)
(标识符, sum)
(加赋值运算符, +=)
(标识符, i)
(分隔符, ; )
(右花括号, })
```

异常提示: 对输入文件中的异常, 按最长匹配的原则输出可识别部分的内容及所在位置(行号+下标), 然后结束运行即可。

选做部分: 正则表达式引擎的设计

通过实现系列算法, 可以将正则表达式转换成 NFA, 再确定化、最小化为 DFA, 最后利用 DFA 进行单词的匹配。应用这种思路, 实现一个正则表达式的引擎, 能支持连接、选择和闭包 3 种基本结构。

方法、步骤:

要完成本实验，依据实验要求进行分解，需要完成的实验步骤是：

1. 如何将“当前状态，遇到字符，进行处理，下一状态”的基本 DFA 描述变成代码？

①定义枚举类型 State 表示 DFA 的状态；处理函数 ProcessFunc，对 DFA 状态进行处理；状态转移函数 StateTransFunc，能够根据当前状态和输入字符得到下一个状态。采用函数指针数组存储上述两种函数，通过上述 3 部分实现“当前状态，遇到字符，进行处理，下一状态”这一 DFA 描述框架。

②对于若干个状态以及状态转移，定义若干个处理状态函数，对应 DFA 中的每一个状态，对当前状态进行处理。同样的定义若干个状态转移函数，分别对应 DFA 中的一个状态和输入，计算具体的下一个状态。

③遍历输入的字符串，根据当前状态和输入字符调用相应的函数对下一个状态进行计算，并更新当前状态。

实现的代码框架如下：

```
//定义枚举类型State
enum State {
    State0,
    State1,
    //其他状态
};
//定义处理函数ProcessFunc
typedef void(*ProcessFunc)(char);
//定义状态转换函数StateTransFunc
typedef State(*StateTransFunc)(char);
//定义若干个处理函数
void Process0(char ch) {
    //处理State0
}
//其他状态的处理函数
//定义若干个状态转换函数
State StateTran0(char ch) {
    //计算从State0输入字符ch后的下一状态
}
//其他状态转换函数
//函数指针数组存储处理函数和状态转换函数
ProcessFunc p_Funcs[] = {
    Process0,
    //其他处理函数
};
StateTransFunc s_Funcs[] = {
    StateTran0,
    //其他状态转换函数
};
//确定当前状态state，设置循环遍历输入的字符串
//在循环内：调用对应的s_Funcs[state]和p_Funcs[state]进行下一状态的计算和处理，更新当前状态
```

2. 如何利用 DFA，快速识别数量有限的关键字、运算符和分隔符？

对于数量有限的关键字、运算符和分隔符，可以把它们看成一个独立的字符，并为每一个字符确定一个接受状态。然后为每一个字符确定一个状态以及状态转换函数，识别输入的代码字符串，遍历每一个字符，判断当前状态和输入该字符的下一个状态，如果计算得到的状态为接受状态，则输入字符为关键字、运算符或分隔符。

3. 如何利用 DFA，识别标识符？并如何与关键字进行区分？

对于标识符识别和关键字识别，分别定义一个标识符 DFA 和关键字 DFA。对于输入的代码字符串，识别其中的标识符，在定义的标识符 DFA 下，根据当前状态和下一个字符，计算

出下一个状态，遍历整个字符串，如果能够找到一个接受状态，则说明识别到标识符；关键字 DFA 的识别同理。如果使用标识符 DFA 识别字符序列成功，则输出该字符序列为标识符；如果失败，则此时再使用关键字 DFA 进行识别，识别成功则输出该字符序列为关键字；否则说明该字符序列不是关键字也不是标识符。

对于能够同时被标识符 DFA 和关键字 DFA 识别的字符序列，应该输出该字符序列为关键字而不是标识符。

4. 如何利用 DFA，识别无符号数并计算其数值？

类似上述对其他类型数据的识别，可以定义一个无符号数 DFA，然后确定开始状态和接受状态以及状态之间的转换函数。对于无符号数，转换条件应该是从上一个状态到下一个状态，字符串的开头是 0~9，到下一个状态的条件是字符串的结尾是 0~9。除了开始状态，其他状态都可以定义为接受状态，即 DFA 可以识别任意长度的无符号数。

若识别成功，对于识别得到的无符号数，统计其位数，然后利用每一位乘以对应 10 的 $n-i$ 次方即可计算其数值。

5. 如何将第一、二部分识别各种单词类别的 DFA 合并成一个大的 DFA，用于完成代码的词法分析？

在两个 DFA 的状态表中添加一个新的状态，连接这两个 DFA 的起始状态到这个新状态，实现合并，然后根据输入字符进行状态转移。在识别过程中，按照合并后的 DFA 状态转换表，进行状态转移，直到到达一个终止状态，然后根据终止状态判断当前是关键字、运算符和分隔符的终止状态，还是标识符和无符号数的终止状态。对不同类型的字符序列进行识别，输出相应的 token，从而实现代码的词法分析。

6. 最长匹配和搜索回退是什么意思，在前面的代码中是如何体现？

最长匹配和搜索回退是文本处理中常用的两种技术。

最长匹配：指在一段文本中查找某个模式时，找到所有可能的匹配，然后选择其中最长的匹配结果作为最终结果。如上述在对标识符和无符号数进行切分时，就是以其他符号作为终止，将终止前的最长匹配结果作为识别的 token。

搜索回退：指在一段文本中查找某个模式时，如果当前位置无法匹配，则回退到前面的位置重新开始匹配。这种技术常用于正则表达式的匹配、语法分析和图像识别等领域。

两者并不是互斥的概念，在某些情况下，还需要结合使用。

实验过程及内容：

一、基于 DFA 的关键字、运算符和分隔符识别

1. 基于编程语言定义所处理语言的关键字、分隔符和运算符

```
//关键字集
const vector<string>KeyWord = { "void", "char", "int", "float", "bool", "w_char", "include", "enum",
    "iostream", "scanf", "main", "printf", "struct", "union",
    "class", "typedef", "std", "long", "short", "signed", "unsigned",
    "const", "volatile", "do", "for", "while",
    "break", "continue", "return", "goto",
    "auto", "register", "static", "extern", "inline",
    "if", "else", "switch", "case", "default",
    "new", "delete", "sizeof",
    "private", "protected", "public",
    "this", "friend", "true", "false", "template", "typename",
    "using", "namespace", "throw", "try", "catch", "operator" };

//运算符集
const vector<string>Operator = { "+", "-", "*", "/", "=", "==", "!=", "<", ">", "<=", ">=" };

//分隔符集
char Separater[8] = { ' ', ',', '.', ':', ';', '{', '}', '[', ']', '(', ')' };
```

2. 采用“查表”和 DFA 两种方法，对比两种方法的识别效率

2.1. 查表方式识别关键字、分隔符和运算符

(1) 定义一个识别类，包含用户输入的字符串、用于存储字符串中关键字的 `keyword`、存储运算符和 `opeword` 以及存储分隔符的 `sepwd`。采用的数据结构如下：

```
string input;
vector<string>keyword;
vector<string>opeword;
vector<char>sepwd;
```

(2) 对于关键字的识别：因为关键字不是单个的字符，因此使用 `string` 中的 `find` 方法实现字符串中关键字的查找。遍历定义的关键字集合，取出每一个关键字，然后判断能否在输入的字符串中找到相应的子串，如果可以，则将当前的关键字存放到 `keyword` 中。具体操作方法如下：

//匹配关键字，获得关键字集

```
void scan_keyword() {
    for (int i = 0; i < KeyWord.size(); i++) {
        string temp = KeyWord[i];
        if (input.find(temp) != string::npos) { //在input中能够找到某个关键字
            keyword.push_back(temp);
        }
        else {
            continue;
        }
    }
}
```

(3) 对于运算符的识别：遍历运算符集合，取出每一个运算符，然后与输入的字符串中的字符一一对应，如果匹配成功，就将该运算符放入 `opeword` 中。第一次找到这个运算符即说明当前字符串中已经存在该运算符，可以使用 `break` 提前退出循环，结束对该运算符的识别。具体操作方法如下：

//匹配运算符，获得运算符集

```
void scan_opeword() {
    for (int i = 0; i < Operator.size(); i++) {
        string temp = Operator[i];
        if (input.find(temp) != string::npos) { //在input中能够找到某个运算符
            opeword.push_back(temp);
        }
        else {
            continue;
        }
    }
}
```

对于分隔符的识别同理。

(4) 运行程序，测试关键字、运算符和分隔符的识别：

```
please input your string:for(int i=0;i<=100;i++){sum += i;}
在这个字符串中，匹配到的关键字集为:int for
在这个字符串中，匹配到的运算符集为:+= < <=
在这个字符串中，匹配到的分隔符集为:; { } ( )
```

2.2. DFA 方式识别关键字、分隔符和运算符

(1) 使用 `unordered_set` 数据结构存储 C++ 语言下的关键字集、运算符集和分隔符集，可以使用其中的 `count` 方法判断输入的字符串中是否含有某个关键字、运算符或分隔符。


```

bool isKeyword(string str) {
    unordered_set<string> keywords = { "void", "char", "int", "float", "bool", "w_char", "include", "enum",
        "iostream", "scanf", "main", "printf", "struct", "union",
        "class", "typedef", "std", "long", "short", "signed", "unsigned",
        "const", "volatile", "do", "for", "while",
        "break", "continue", "return", "goto",
        "auto", "register", "static", "extern", "inline",
        "if", "else", "switch", "case", "default",
        "new", "delete", "sizeof",
        "private", "protected", "public",
        "this", "friend", "true", "false", "template", "typename",
        "using", "namespace", "throw", "try", "catch", "operator" };
    return keywords.count(str) > 0; //unordered_set中, 统计str在keywords中出现的次数
}

bool isOperator(string str) {
    unordered_set<string> operators = { "+", "-", "*", "/", "=", "==", "!=", "<", ">", "<=", ">=" };
    return operators.count(str) > 0;
}

bool isSeparator(char ch) {
    unordered_set<char> separators = { ',', ' ', '}', '{', '}', '[', ']', '(', ')', ';' };
    return separators.count(ch) > 0;
}

```

(2) 基于 DFA 方式的识别, 按照方法、步骤 1 中所述, 可以按照当前状态→输入字符→下一状态的思路, 分别得到输入的 C++语言中的字母子串、运算符子串或分隔符字母。如对于关键字, 读取输入字符串的每一位, 当识别到字母时, 将其添加到当前待判断的子串的尾部, 然后对于分隔出来的字母子串进行判断, 如果在关键字集合中, 该子串出现的次数大于 0, 说明查找到该关键字, 将它放到关键字列表中。

```

for (int i = 0; i < input.length(); i++) {
    char ch = input[i];
    if (isalpha(ch)) { // 字母
        current_token += ch;
        while (i + 1 < input.length() && (isalpha(input[i + 1]))) {
            i++;
            current_token += input[i];
        }
        if (isKeyword(current_token)) {
            keyword.push_back(current_token);
        }
        current_token = "";
    }
    else if (isSeparator(ch)) { // 分隔符
        sepword.push_back(ch);
    }
    else if (isOperator(string(1, ch))) { // 运算符
        current_token += ch;
        while (i + 1 < input.length() && isOperator(current_token + input[i + 1])) {
            i++;
            current_token += input[i];
        }
        opword.push_back(current_token);
        current_token = "";
    }
}

```

(3) 通过上述方法从字符串识别得到的关键字集、分隔符集和运算符集, 还需要进行去重处理。使用 vector 数据结构进行存储, 就可以结合 sort 和 erase 进行去重。

```
//对关键字、分隔符和运算符进行去重并进行输出
sort(keyword.begin(), keyword.end()); // 将vector排序
keyword.erase(unique(keyword.begin(), keyword.end()), keyword.end()); // 去重
cout << "在这个字符串中，匹配到的关键字集为：";
for (const auto& s : keyword) {
    cout << s << " ";
}
```

(4) 运行程序，测试关键字、运算符和分隔符的识别：

```
please input your string based on C++: for(int i=0;i<=100;i++){sum += i;}
在这个字符串中，匹配到的关键字集为：for int
在这个字符串中，匹配到的运算符集为：+ <= =
在这个字符串中，匹配到的分隔符集为：( ) ; { }
```

二、基于 DFA 的标识符和无符号数识别

1. 思路：基于 DFA 实现标识符和无符号数的识别，可以首先定义识别 DFA，构造出其状态转换表，然后使用状态转换表对输入的字符串进行匹配。构造的状态表可以为：

状态（代码中用 0~4 表示）	字母	数字	其他
开始状态 q0	q1	q2	q3
q1	q1	q1	q3
q2	q4	q2	q3
终止状态 q3	q4	q4	q3

对上述状态表进行解释：q1 表示标识符状态，q2 表示无符号数状态，q3 表示其他字符状态，该状态作为终止状态。对于输入的字符串，因为标识符以字母开头，后面可以是任意长度的字母和数字，于是当开始状态 q0 识别到字母时，进入标识符状态 q1，继续识别到字母和数字时，仍处于当前状态，直到遇到其他字符进入终止状态 q3。同样的，当开始状态 q0 识别到数字时，进入无符号数状态 q2，如果继续识别到数字，则保持当前状态进行无符号数的识别，如果遇到字母或其他字符，则进入终止状态。

在实现中，对于上述是字母、数字还是其他字符的判断，可以借助 `isalpha()` 方法和 `isdigit()` 方法实现。使用一个二维数组存放上述状态表，读取文件获得每一行输入。然后定义一个构建中的 `token`，如果识别到字母和数字，则不断添加到 `token` 中，直到遇到其他字符。如果是无符号数，需要使用过单个字符转数值的形式得到其数值。

2. 代码实现：

2.1. 定义状态表

```
int dfa[4][3] = {
    {1, 2, 3},
    {1, 1, 3},
    {3, 2, 3},
    {3, 3, 3}
};
```

2.2. 识别字母、数字或其他字符，进行对应的状态转换

```

int type = 0;
if (isalpha(c) || c == '_' ) {
    type = 0; // 字母
}
else if (isdigit(c)) {
    type = 1; // 数字
}
else {
    type = 2; // 其他字符
}
state = dfa[state][type];

```

2.3. 判断当前状态是否为结束状态，如果不是则说明正在进行 token 的创建（即标识符或无符号数的继续识别），如果是则输出当前识别结果，并重置 token 值和 state 值

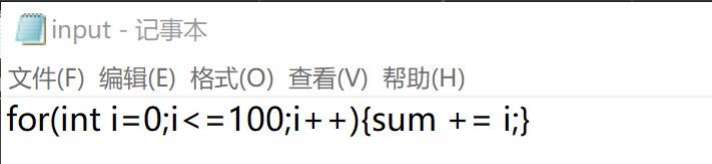
```

if (state == 1 || state == 2) {
    token += c;
}
else if (state == 3) {
    if (!token.empty()) {
        if (isdigit(token[0])) {
            // 通过遍历字符串中的每个字符，将其转换为数值，单个字符转数值形式
            int value = 0;
            for (auto c : token) {
                value = value * 10 + (c - '0');
            }
            cout << "无符号数: " << value << endl;
        }
        else {
            cout << "标识符: " << token << endl;
        }
    }
    //重置token和state
    token = "";
    state = 0;
}

```

3. 运行程序：

读取的 txt 文件内容为：



可以看到程序已对标识符和无符号数进行了切分，并计算出无符号数的数值。


```
标识符: for
标识符: int
标识符: i
无符号数: 0
标识符: i
无符号数: 100
标识符: i
标识符: sum
标识符: i
```

三、基于 DFA 的分词程序

1. 根据方法、步骤中的 5 中的合并方法，可以重新定义 DFA 的状态转换表为

```
# 定义DFA状态转移表
state_transition = {
    0: {'letter': 1, 'digit': 2, 'operator': 3, 'delimiter': 4, 'other': 5},
    1: {'letter': 1, 'digit': 1, 'delimiter': 0, 'other': 0},
    2: {'digit': 2, 'dot': 6, 'operator': 0, 'other': 0},
    3: {'operator': 3, 'other': 0},
    4: {'delimiter': 4, 'other': 0},
    5: {'other': 0},
    6: {'digit': 7},
    7: {'digit': 7, 'other': 0}
}
```

这里使用 python 实现该分词程序，重新规定关键字、分隔符和运算符如下：

```
# 定义关键字集合
keywords = {'auto', 'break', 'case', 'char', 'const', 'continue', 'default', 'do', 'double', 'else', 'enum',
            'extern', 'float', 'for', 'goto', 'if', 'int', 'long', 'register', 'return', 'short', 'signed', 'sizeof',
            'static', 'struct', 'switch', 'typedef', 'union', 'unsigned', 'void', 'volatile', 'while'}

# 定义运算符集合
operators = {'+', '-', '*', '/', '%', '=', '>', '<', '!', '&', '|', '^', '~', '++', '--', '+=', '-=', '*= ', '/=',
            '%=', '==', '!=', '>=', '<=', '&&', '||', '>>', '<<', '>>=', '<<=', '&=', '|=', '^='}

# 定义分隔符集合
delimiters = {' ', ';', '(', ')', '[', ']', '{', '}'}
```

2. 代码实现：

2.1. 定义注释和空白字符的正则表达式，去除文件中的注释和空白字符。

```
# 定义注释正则表达式
comment_pattern = re.compile(r'//.*?$|/\*.*?\*/', re.S | re.M)

# 定义空白字符正则表达式
whitespace_pattern = re.compile(r'\s+')

# 去除注释
code = comment_pattern.sub('', code)

# 去除空白字符
code = whitespace_pattern.sub('', code)
```

2.2. 接着就可以开始逐行读取文件，对其进行去注释的操作。这里实际上应该也对空白进行忽略，但在实际操作过程中，如果在切分前就对空白进行去除，关键字可能会跟标识符连接，导致最终的识别结果有误，于是在这里我没有真正对空白进行去除，而是把它作为一个

“other”类型的字符存在，当读到空白时，可以进入接收状态。

根据上述状态转换表，可以依次读取文件中每一行代码的每一个字符，然后记录字符的类型。

遍历代码字符

```
for i, c in enumerate(code):
    # 获取字符类型
    if c.isalpha():
        char_type = 'letter'
    elif c.isdigit():
        char_type = 'digit'
    elif c in operators:
        char_type = 'operator'
    elif c in delimiters:
        char_type = 'delimiter'
    else:
        char_type = 'other'
```

相对于当前状态去更新读入新字符后的状态，如果为终止状态，则进行切分。并把切分结果以及其对应的类型添加到 tokens 中。

更新缓存和状态

```
state = state_transition[state][char_type]
if state != 0:
    buffer += c
```

检查是否达到终止状态

```
if state == 0:
    # 输出Token
    if buffer in keywords:
        tokens.append(('关键字', buffer))
    elif buffer in operators:
        tokens.append(('运算符', buffer))
    elif buffer in delimiters:
        tokens.append(('分隔符', buffer))
    elif re.match(r'^\d+(\.\d+)?$', buffer):
        tokens.append(('整数', buffer))
    elif re.match(r'^[a-zA-Z_]\w*$', buffer):
        tokens.append(('标识符', buffer))
    buffer = ''
```

2.3. 对于当前程序，还需检查是否存在未处理的缓存，如果有，对未处理缓存重复上述判断并添加的操作。最后遍历 tokens 进行输出即可。

检查是否还有未处理的缓存

```
if buffer:
    if buffer in keywords:
        tokens.append(('关键字', buffer))
    elif buffer in operators:
        tokens.append(('运算符', buffer))
    elif buffer in delimiters:
        tokens.append(('分隔符', buffer))
    elif re.match(r'^\d+(\.\d+)?$', buffer):
        tokens.append(('整数', buffer))
    elif re.match(r'^[a-zA-Z_]\w*$', buffer):
        tokens.append(('标识符', buffer))
```

3. 运行结果:

运行下图所示的 c 程序

 input - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
for ( int i = 0 ; i <= 10 ; i ++ ) { sum += i ; } //this is a wrong code
```

可以看到运行结果中已经对不同类型的数据进行了切分

```
('关键字', 'for')
('分隔符', '(')
('关键字', 'int')
('标识符', 'i')
('运算符', '=')
('整数', '0')
('分隔符', ';')
('标识符', 'i')
('运算符', '<=')
('整数', '10')
('分隔符', ';')
('标识符', 'i')
('运算符', '++')
('分隔符', ')')
('分隔符', '{')
('标识符', 'sum')
('运算符', '+=')
('标识符', 'i')
('分隔符', ';')
('分隔符', '}')
```

进程已结束,退出代码0

由于代码中对关键字、分隔符和运算符的定义中，每个集合所包含的元素较多，为减少代码对识别结果判断的冗余，在我的程序中没有进一步去识别具体是哪一个关键字。

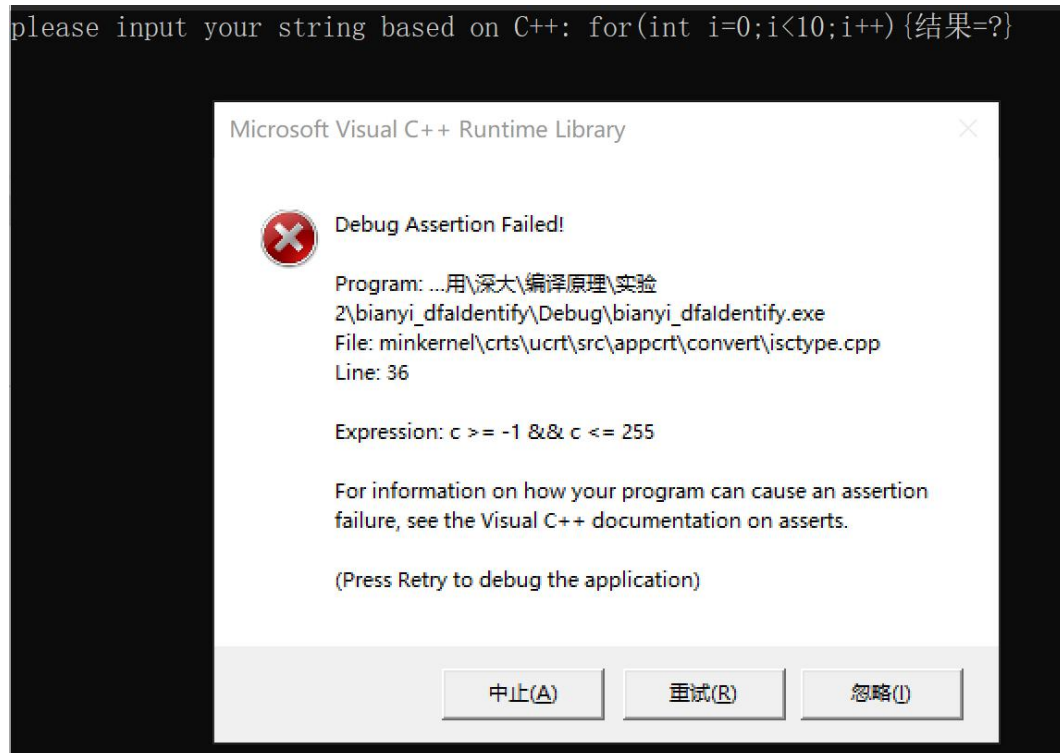
实验结论：

1. 为了验证你第一部分编写的程序是准确的，你设计了什么测试数据（测试用例至少包含 1 个正例、1 个违例、1 个临界例）进行测试，得到的结果如何。

在实验过程与内容部分，我对正例的测试结果进行了展示，接着设置违例、临界例对程序进行进一步测试。

（1）设置违例进行测试：

当输入的字符串中包含非法字符（如中文字符）时，可以看到程序无法正常运行。



可以在程序中加入是否存在中文的判断：定义一个判断方法，如果输入的字符串中包含中文，输出错误信息，然后退出程序。

//返回0：无中文，返回1：有中文

```
int IncludeChinese(string s)
{
    const char* str = s.data();
    char c;
    while (1)
    {
        c = *str++;
        if (c == 0) break; //如果到字符串尾则说明该字符串没有中文字符
        if (c & 0x80) //如果字符高位为1且下一字符高位也是1则有中文字符
            if (*str & 0x80) return 1;
    }
    return 0;
}
```

再次输入违例，可以看到程序输出了报错信息。


```
please input your string based on C++: for(int i=0;中文)
字符串中包含中文!
```

(2) 设置临界例进行测试:

当输入的字符串中为空时, 可以看到程序仍然打印了空集合。

```
please input your string based on C++:
基于DFA的运行时间: 2081ms
在这个字符串中, 匹配到的关键字集为:
在这个字符串中, 匹配到的运算符集为:
在这个字符串中, 匹配到的分隔符集为:
```

此时在程序中加入输入字符串为空或集合为空的判断, 然后输出相应的判断结果。

```
if (input.length() == 0) {
    cout << "输入的字符串为空" << endl;
    return 0;
}
```

运算符和分隔符集是否为空的判断同理。

```
if (keyword.size() == 0) {
    cout << "匹配不到任何关键字" << endl;
}
```

再次输入临界例, 可以看到运行结果为:

```
please input your string based on C++:
输入的字符串为空
```

```
please input your string based on C++: int
基于DFA的运行时间: 1379ms
在这个字符串中, 匹配到的关键字集为: int
匹配不到任何运算符
匹配不到任何分隔符
```

```
please input your string based on C++: i=0
基于DFA的运行时间: 5134ms
匹配不到任何关键字
```

```
在这个字符串中, 匹配到的运算符集为: =
匹配不到任何分隔符
```

```
please input your string based on C++: ;;
基于DFA的运行时间: 4355ms
匹配不到任何关键字
匹配不到任何运算符
```

```
在这个字符串中, 匹配到的分隔符集为: ;
```

2. 为了验证你第二部分编写的程序是准确的, 你设计了什么测试数据(测试用例至少包含 1 个正例、1 个违例、1 个临界例)进行测试, 得到的结果如何。

在实验过程与内容部分, 我对正例的测试结果进行了展示, 接着设置违例、临界例对程序进行进一步测试。

(1) 设置违例进行测试:

同理于 1 中违例的设置，可以在程序中加入输入代码中是否存在“不支持的字符”的判断，如果存在不支持的字符，打印相关的错误信息。这里可以通过在状态表中添加一个不支持状态实现。当遇到不支持的字符时，DFA 进入该状态。在对状态进行判断时，如果遇到该状态，则输出相应信息。

```
int dfa[5][3] = {
    {1, 2, 4},
    {1, 1, 3},
    {3, 2, 3},
    {3, 3, 3},
    {4, 4, 4}
};

else if (state == 4) {
    cout << "不支持的字符：" << c << endl;
    token = "";
    state = 0;
}
```

因为程序仅仅只是对标识符和无符号整数进行识别，于是分隔符等对于当前程序来说，就是不支持的字符。运行程序，可以看到程序对相应信息进行了打印

```
标识符: for
标识符: int
标识符: i
无符号数: 0
标识符: i
不支持的字符: =
无符号数: 10
标识符: i
不支持的字符: +
不支持的字符: )
不支持的字符: {
不支持的字符:
标识符: sum
不支持的字符: +
不支持的字符: =
不支持的字符:
标识符: i
不支持的字符: }
```

(2) 设置临界例进行测试:

同理于 1 中临界例的设置，可以设置输入的代码中不存在标识符或无符号数的临界例，相对的，在程序中加入当前没有识别到标识符或无符号数的判断，对判断信息进行输出。

```
if (token.empty()) {
    cout << "当前行中不存在标识符或无符号数" << endl;
}
```

在输入的代码文件中加入一行注释符

 input - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
for(int i=0;i<=10;i++){ sum += i;}
//
```

运行程序，可以看到程序对相应信息进行了打印。

```
标识符: for
标识符: int
标识符: i
无符号数: 0
标识符: i
无符号数: 10
标识符: i
标识符: sum
标识符: i
当前行中不存在标识符或无符号数
当前行中不存在标识符或无符号数
```

3. 为了验证你第三部分编写的程序是准确的，你设计了什么测试数据（测试用例至少包含 1 个正例、1 个违例、1 个临界例）进行测试，得到的结果如何。

在实验过程与内容部分，我对正例的测试结果进行了展示，接着设置违例、临界例对程序进行进一步测试。

（1）设置违例进行测试：

同理于 1、2 的违例，我们可以在程序中加入对不支持的字符的判断。在该 python 程序中，我们已经在状态表中定义了一个 other 状态，要对错误信息进行输出，只要在对状态进行判断的时候，加上当状态为 other 时，把当前识别结果记录为“不合法”，然后同样加入待输出的 tokens 中。

检查状态是否合法

```
if char_type not in state_transition[state]:
```

```
    # 不合法则回退一个字符
```

```
    tokens.append(('不合法', buffer))
```

```
    buffer = ''
```

```
    state = 0
```

```
    continue
```

```
# 输出Token
```

```
if buffer in keywords:
```

```
    tokens.append(('关键字', buffer))
```

```
elif buffer in operators:
```

```
    tokens.append(('运算符', buffer))
```

```
elif buffer in delimiters:
```

```
    tokens.append(('分隔符', buffer))
```

```
elif re.match(r'^\d+(\.\d+)?$', buffer):
```

```
    tokens.append(('整数', buffer))
```

```
elif re.match(r'^[a-zA-Z_]\w*$', buffer):
```

```
    tokens.append(('标识符', buffer))
```

```
else:
```

```
    tokens.append(('不合法', buffer))
```

在读取文件时，还可以直接在读取过程中加入对是否有中文进行判断。

```
with open('input.c','r',encoding="utf-8") as f:
    code = f.read()
pattern = re.compile(r'[\u4e00-\u9fa5]')
result = pattern.findall(code)
if result:
    print('该文件包含中文')
```

现在在文件中加入中文，运行程序，可以看到程序对相应的信息进行了输出。

```
1.py x input.c x
for ( int i = 0 ; i <= 10 ; i ++ ) { sum += i ; 中文 } //this is a wrong code
```

该文件包含中文

进程已结束,退出代码0

(2) 设置临界例进行测试:

设置临界例: 在 input 文件中, 有两个运算符连续存在

```
1.py x input.c x
for ( int i = 0 ; i <= 10 ; i ++ ) { sum +== i ; } //this is a wrong code
```

运行程序, 可以看到程序对该情况判断为“不合法”并将错误信息进行了输出(只截取该部分输出结果)。

```
('标识符', 'sum')
('不合法', '+==')
('标识符', 'i')
('分隔符', ';')
('分隔符', '}')
```

进程已结束,退出代码0

心得体会:

通过本次实验, 对如何识别出某编程语言下字符串的关键字、分隔符、运算符等字符序列进行识别。并学会如何基于 DFA, 确定识别过程中可能出现的状态以及状态之间的转换条件, 能够设置相应的状态转换表, 以实现基于 DFA 的词法分析程序。

一个基于 DFA 的词法分析器是编译器中的一个重要阶段, 它能够将源代码转换成一个个 token, 为语法分析和语义分析提供基础数据。可以通过手写实现词法分析器, 也可以使用生成工具 lex、flex 等, 还可以使用正则表达式。

指导教师批阅意见：

成绩评定：

指导教师签字：蔡树彬
2023 年 4 月 18 日

备注：