# 汇编实验词法分析器实验报告

09017423 杨彬

## 一、实验目的

本实验我结合编译原理课程所学习词法分析的知识，基于 C++，利用有限状态机 DFA，构造一个词法分析器。通过实际的编程，更加深入地理解词法分析涉及到的技术，同时为之后的实验，包括语法分析等打下基础。

## 二、内容简介

实验内容：

Input: Stream of characters

Output: Sequence of tokens

实验步骤:

1、 首先自己定义正则表达式，手动将正则表达式转化为 NFA，再完成 NFA 转化为 DFA，DFA 最小化得到最终的 DFA。

2、 我用 C++实现了一个可高度定制的状态机模型(状态机类)。

3、 基于 1 中得到的 DFA 图和 2 中的状态类，我将 DFA 的状态，和状态变换关系输入到状态机模型中，在 main 函数中完成状态机模型的初始化

4、 我基于自己设计的语法，将代码写入文本文件中，初始化后的状态机模型，以该文本文件为输入，读入字符串，进行词法分析，将词法分析得到的 token 序列输出到文本文件中。

5、 通过分别进行正确用例和错误用例的测试，验证实验是否正

确

## 三、 思路方法

由于本实验的 DFA 的构造过程是人工构造的（也就是我通过自己定义的 Res 构造 DFA）并不由代码生成的。因此本实验的实际的编程难点在于如何用代码实现 DFA。在本实验中，我采用构造 DFA 类和构造 state 类的方式，state 类是状态类，state 类包含指向下一个 state 的边，而 DFA 类则保存所有的 state 类，并且知道初始的 state。构造 DFA 类的过程，就是设置 DFA 类 state 成员变量，以及设置各个 state 之间边关系的过程。在进行词法分析的时候，每输入一个字符，当前的 state 会根据输入的字符自动返回指向下一个 state 的指针，这样就完成了状态到状态之间的变化。

DFA 中除了保存所有的 state 之外，还保存了表示终结状态的 state 的 id，当扫描到最后一个字符之后，我只需要根据 DFA 中保存的终结状态 id，判断当前的 state 是不是终态就能够得到 token、判断分析是否成功。

而对于一些系统的保留字，比如说 if、else、while，则无需使用状态机进行扫描，只需要将所有的关键字保存在一张哈希表中，哈希表的 key 就是这些保留字。在进行词法分析的时候，我们先查哈希表，如果能从哈希表中找到，就说明是保留字，直接将输出该保留字代表的 token。否则用 DFA 进行下一步的分析。

综上，整个词法分析器：

1 用哈希表判断保留字（关键字）

2 用上述说明的 DFA 类进行，id(变量)，数字、字符串、字符等的识别。而无需使用一堆的 if else switch 语句.

## 四、假设定义

### 1、词法定义：

在本实验中我仿照常用的编程语言，设计了词法，词法包括，

(1)保留字：if else then while do ( ) + - * / ; = and or not > < >+ <= == != true false { } (用空格隔开)

```
vector<string>({ "if","else","while","do","(",")","+","-","*","/", ";" ,"=","and","or","not",">","<",">=","<","==","!=" ,"true","false",
            "then","{" ,"}" });
```

(2)非保留字:

说明:letter: 代表 26 个字母(不分大小写), num:代表 0-9 数字,chars1:除了 " 之外的所有字符，chars2:除了 ' 之外的所有字符

id(变量) : RE: _|letter(_|letter|num)* (*代表闭包)

num(数字常量): digit(digit)*.digit(digit)*

string(字符串): "(chars1)* "

char(字符): 'chars2'

在我设计的词法中：

1 id(变量)只能包含字母、下划线、数字，且只能由字母或者下划线开头。

2 num 可以为整数或者小数（不支持 4e10 这种形式）

3 字符串为双引号之间可以有任意数目除了″的字符(包括空格，但不支持转义)

4 字符为单引号之间除了’的字符（包括空格，但不支持转义）

(3)token 定义:

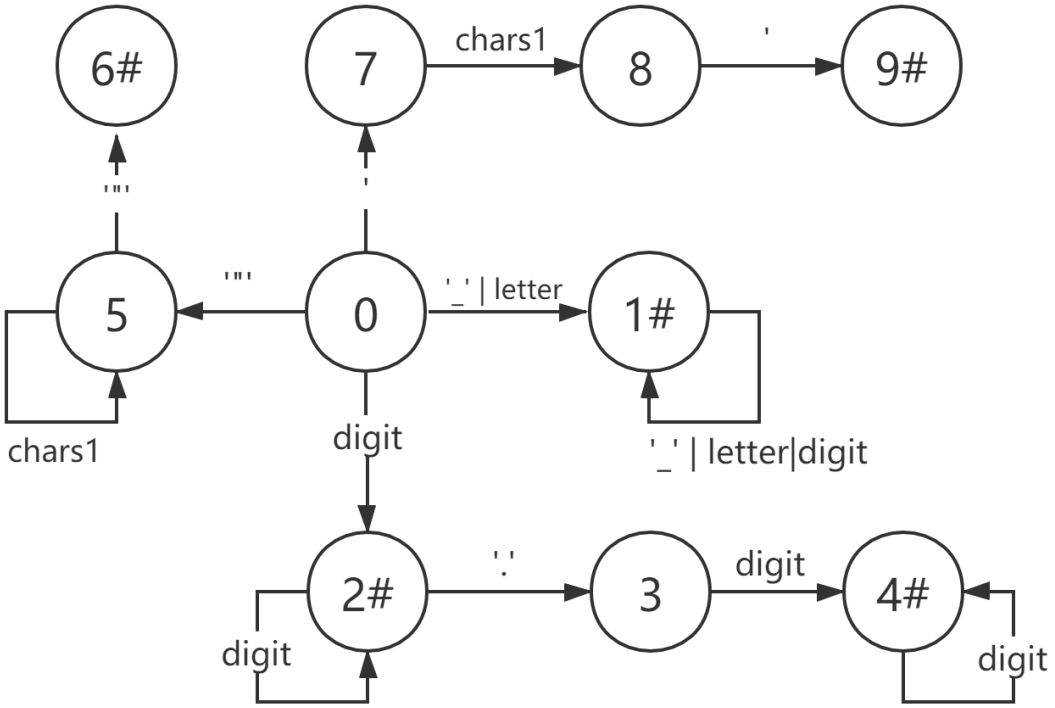
在这个实验中，我使用的 token 为(token_id,token_name,value),其中 token_name 是所有的保留字，和 id num string char,token_id 是对 token_name 的编号,token_id 和 token_name 一一对应。Value 是 token 的值,对于保留字来说 token 没有值，其值为默认的“_”,而对于 id, num,string,char 来说其值分别使其变量名，常量值，字符串值，字符值。为了简化问题，token_value 采用字符串（string）进行存储，如果后续需要，再根据其 token_name 将 token_value 转化为所需要的相应值，比如 int,double 等

**2 其它假设**

由于本实验的输入是一个文本文件，输出是一个 token 序列。而 DFA 的扫描函数的输入是一个 string,输出是一个 token。因此我首先需要把文本文件的内容提取为字符串流。为了实现从文本文件到字符串流的转换，我定义在字符串之间以空格、制表符、换行符为分隔符。(’ ’、’\n’、’\t’)，eg：如果文本文为 q = 10 ,那么字符串序列为{”q”,”=”,”10”},因为我定义了分割符所以常用的写法 q=10 会被认为是“q=10”,导致出现词法分析错误，这点需要注意。

## 五、状态图

根据 四 中对词法的定义，我构造出了下面的 DFA：




（ps ：其中有#的代表终态，0 状态是初态）

## 六、数据结构

如前所述，本实验我最重要的编程任务就是实现一个高度可定制的有限状态机模型。核心的数据结构如下：（我用 C++实现，出于篇幅问题，这里只给出头文件，和重要函数的描述，实现细节请查阅源代码（附件或者 github,我将代码上传到了 github，文末附 github 地址）。

### 1 token 类

```
class token
{
public:
    token(int t_id,string t_name, string v);
    ~token();
```

```
    friend ostream& operator<<(ostream& output, const token& t);
    friend ofstream& operator<<(ofstream& foutput, const token& t);
    void generateTokenStream  (ofstream& fout);
    int getTokenID();
private:
    int token_id;
    string token_name;
    string value;
};
```

Token 类是用来保存上述提到的 token 数据的数据结构，除了重载输入输出符号，以及将 token 内容输出到文本的函数之外，token 类的属性有 token_id ， token_name, value

## 2 State 类

```
    // 这里的State是DFA的State
class State
{
public:
    State(int id);
    ~State();
    // getter
    inline int getStateID() {
        return stateID;
    }
    bool addPtrs(const string c, State* next);        //增加某条边
    bool removePtrs(const string c);                  // 移除某条边
    void setEdgeJudgeFunction(string(*f)(string str));
    State* transfer(const string c);                 //根据输入的字符返回下一个状态的指
针
protected:
    unordered_map<string, State*> next_ptrs ;
    string (*edgeJudgeFunction) (string str);
    int stateID;
};
```

State 类是构成 DFA 类的重要的数据结构，其基本原理类似于链表。在 State 类中有一个哈希表（unordered_map 是 STL 定义的哈希表数据结构），这个哈希表的 key 是字符串，代表边的名称，值是一个指向 State 的指针。这个哈希表保存从一个 State 到另一个 State

的转换关系，stateID 则是一个编号。

而 transfer 函数，做的是就根据输入字符串 c，通过 edgeJudgeFunction 函数映射为一个 key，在边哈希表里查找 key，得到指向下一个状态的指针，完成状态的变化。edgeJudgeFuntion 是一个函数指针，其目的就是将 tranfer 的输入 c 根据用户的需求重新映射为一个 key。设置这个属性是为了使得这个 DFA 类做到高度可定制，同时还有一些其它原因（将在后面的部分阐述）

## 3 DFA 类

```
class DFA
{
    struct tokenItem{      //该结构体是 token的 id 和 name对应的一个结构体
        int token_id;
        string token_name;
        tokenItem(int t_id, string t_name):token_id(t_id),token_name(t_name) {
        }
    };
public:
    DFA();
    ~DFA();
    bool addTokenItem(int key, tokenItem i);
    bool addTokenItem(int key, int token_id, string token_name);
    bool removeTokenItem(int key, tokenItem i);
    bool addKeepWord(string keyword,int token_id);
    void addState(State s);                          //添加状态
    State& getState(int i);
    void initStates(const int stateNum,  vector<int>** stateMatrix, const string
edgeList[]);
    token isKeepWord(string keyword);                        //判断是不是保留字
    token scan(string str,  string(*inputConvert)(char c) = [](char c)->string {return
string(1, c); });// 扫描字符串生成token
private:
    unordered_map<int, tokenItem> tokenDict;       // 终结符
    unordered_map<string, token> keepWords;        // 保留字
    vector<State> states;
};
```

这个类是本程序的主类，

类成员：

tokenDict: 事实上在状态图每一个终结状态对应着一种 token，这个属性是一个将终结态的状态 id 映射到对应 token 的哈希表（tokenDict,是一个结构体,和 token 的区别在于其预设了 token_id 和 token_name，但是没有 token_value，可以通过 tokenItem 生成 token)。

keepWords: 如前所述，对于保留字，我们通过直接查找哈希表来实现，这个属性是一个将字符串（保留字字符串）映射到相应保留字对应的 token 的哈希表。通过查找这个哈希表实现识别保留字。

states : 保存 DFA 所有的状态（每个状态是一个 State 对象)，其中 states[0]为初始状态。

类方法:

除了增加 states 等操作方法之外，这个类核心的方法是:

scan： 这个函数输入是一个字符串，输出是一个 token，这个函数的作用就是将字符串识别为 token，如果字符串无法识别为任意一种 token，那么将会打印错误信息，同时返回一个错误的 token,默认错误值是（-1,“error”,string）string 代表字符串本身。注意到 scan 函数还有一个默认参数，这个参数是一个函数指针，其作用就是将每次扫描时的字符 c 转化为字符串 str，这个 str 是 state 的 tranfer 函数的输入，默认情况下这个转化函数就是将字符直接转化为字符串。设置这个参数的主要目的就是为了实现一个高度可定制化的 DFA，通过这个参数，和 state 类中的 edgeJudgeFunction，我设

计了一个高度可定制的有限状态机类。

## 七、核心算法

本实验的核心是将 string 识别为 token，因此核心的函数是 DFA 类中的 scan 函数，

**scan 函数的流程如下：**

//输入 string, 输入 token, 参数：转化函数 string(*inputConvert)(char c)

(1) 查 hash 表判断 string 是不是关键字

(2) IF string 是关键字

(3) RETURN string 对应的 token ，算法结束

(4) goto (5)

(5) state = states[0] ，取出初始状态

(6) c = string 的第一个字符

(7) while state 不是空状态 do

(8) if c 是 string 字符串最后一个字符 then

(9) if 当前 state 是终态 then

(10) RETURN 该终态对应的 token

(11) else 词法分析 error RETURN error token // 词法分析失败

(12) str = convertInput(c);

(13) nextState = state.tranfer(str);

(14) If nextState 是空状态 then

(15) 词法分析 error, RETURN error token

(16)　Else c = string 下一个字符

注：state.tranfer(str)函数在之前的部分有过叙述，该函数比较简单，详情查看源代码。

**主函数算法流程:**

(1) 根据我们从词法推导的 DFA，初始化所有的 state 类和 DFA 类，初始化后 DFA 的实例 myDFA

(2) 根据需要设置上述提到的 convertInput 函数，和 edgeJudgeFunction 函数

(3) char c

(4) string str

(5) isIgnore = true　　//是否忽略分割符（在双引号之间的就应该忽略分隔符获得完整的字符串

(6) while 能成功获得文件流的当前字符 c do :

(7)　if c 为 空格|制表符|换行（' '|'\n'|'\t'）and isIgnore ==false then:

(8)　　If str 长度为 0 :

(9)　　Goto (6)　　//防止连续的分隔符

(10)　　Token t = myDFA.scan(c, convertInput)

(11)　　将 t 输出到输出文件同时 str 清空

(12)　else:

(13)　　If c 为双引号 then:

(14)　　将 isIgnore 取反　　//取反代表进入字符

串和离开字符串

(15) str+= c //不是分割符说明 c 是 str 的一部分

## 八、测试用例

1 正确用例：

Input:

```
i = 0 ;
k = 5 ;
p = 'c' ;
q = "hello world" ;
while ( i < 10 ) do {
   if ( ( i > 5 ) and ( k > j ) ) then {
      k = k - 1 ;
   }
   i = i + 1 ;
}
```

output:

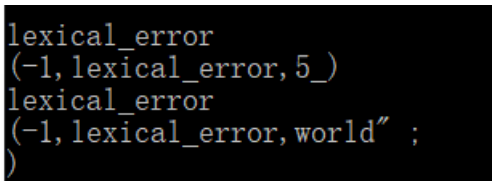
```
(0,id,i)
(16,=,_)
(1,num,0)
(15,;,_)
(0,id,k)
(16,=,_)
(1,num,5)
(15,;,_)
(0,id,p)
(16,=,_)
(4,char,'c')
(15,;,_)
(0,id,q)
(16,=,_)
(3,string,"hello world")
(15,;,_)
(7,while,_)
(9,(,_)
(0,id,i)
(21,<,_)
(1,num,10)
(10,),_)
(8,do,_)
(29,{,_)
(5,if,_)
(9,(,_)
(9,(,_)
(0,id,i)
(20,>,_)
(1,num,5)
(10,),_)
(17,and,_)
(9,(,_)
(0,id,k)
(20,>,_)
(0,id,j)
(10,),_)
(10,),_)
(28,then,_)
(29,{,_)
(0,id,k)
(16,=,_)
(0,id,k)
(12,-,_)
(1,num,1)
(15,;,_)
(30,},_)
(0,id,i)
(16,=,_)
(0,id,i)
(11,+,_)
(1,num,1)
(15,;,_)
(30,},_)
```

2错误用例：

Input:

```
5_i = 0 ;
k = 5 ;
p = 'c' ;
q = "hello " world" ;
```

Output:




```
(-1,lexical_error,5_)
(16,=,_)
(1,num,0)
(15,;,_)
(0,id,k)
(16,=,_)
(1,num,5)
(15,;,_)
(0,id,p)
(16,=,_)
(4,char,'c')
(15,;,_)
(0,id,q)
(16,=,_)
(3,string,"hello ")
(-1,lexical_error,world" ;
)
```

## 九、出现的问题和解决方案

在实验过程中我遇到的问题有

1 一开始在 DFA.scan 函数中，我是通过判断当前输入的字符 c 是不是等于我设定的终结符，如果遇到了终结符我认为字符串读完了，这种方式。但这种方式显然有问题，因为对于字符串来说，字符串内可以是任何非“的字符，采用上述方法就会导致，在识别字符传的时候，scan 函数会将字符传内的符号认为是终结符（如果 c==终结符的话），导致 scan 函数提前结束，从而出错。后来我更改为，判断 c 的下标是否>=string.size()就很好解决了这个问题，问题产生的原因主要是我一开始考虑问题不周，一开始我只是想构建一个状态机并没有考虑具体的使用场景从而导致出错。

2 在上面的部分中我提到了 设置 convertInput，和

edgeJudgeFunction 是为了使得 DFA 能够高度定制化，这只是原因之一，更直接的原因是是因为，最开始我没有设置这两个函数指针，会出现假设在 tranfer 的时候我需要：当读入字符是字母的时候，跳转到同一个状态，如果 convertInput 函数，我必须设置 26+26（大小写）条边，这显然是不行的。有了 convertInput 函数之后，我能够自己写一个映射函数：如果输入是字母，我将 c 映射为”letter”这个字符串，在保存边的时候将 letter 作为 key 储存下来。

但是如果之后 convertInput 函数还是不能完全解决问题，以本实验的状态图为例，假设只有 convertInput，我需要将所有的 字母映射为 letter， 同时将所有的非 ”字符映射为 chars1,这时候问题来了因为 letter 和 chars1 是含有公共前缀的，换句话说 a 能够同时映射到 letter 和 chars1，如果在状态 5 中我存储的边是 chars1 的话，那么读到 a ，convertInput 函数可能会将 a 映射为 letter 而不是 chars1，会导致状态转换出错。如果只用 convertInput 可能会让 convertInput 函数异常复杂，充满很多 if else。

因此又设置了 edgeJudgeFunction，它将 states.tranfer(str) 的输入 str 做二次映射（默认情况下是不进行转换），这样即使存在上述的公共前缀问题，我们仍然可以通过指定第二个映射，如果 states.tranfer 的输入是 letter,那么我在 edgeJudgeFunction 这个函数中将 letter 映射为 chars1，这样就能以较小的代价解决问题。同时这两个函数指针都是有默认值的，我们只需要针对我们的需

要进行定制，有了这两个函数指针，整个 DFA 系统变得高度可定制化了。

## 十、实验感悟

通过本实验，我更加深入地理解了词法分析的原理，自己构造 RE，再将 RE 转化为 NFA，转化为 DFA，最后实现一个 DFA。这个过程我复习了课程内容，同时使用 DFA 进行词法分析的过程原理有了更深刻的理解。虽然本实验的变成人物从逻辑上不难，但我还是结合我在其他课程上学习到的内容（设计模式），尝试设计了一个可定制的有限状态机模型，提升了我的编程能力，和对软件开发的理解。受益匪浅。

Github 源代码（包括相关文件，这是本人的 github 仓库）：

https://github.com/chonepieceyb/SEU-CS-LEARNING/tree/master/%E7%BC%96%E8%AF%91%E5%8E%9F%E7%90%86/%E5%AE%9E%E9%AA%8C/LexicalAnalyzer