词法分析器实验报告

学号：19335151　　姓名：吕文禧

实验要求

> 实验描述：手动设计实现，或者使用Lex实现词法分析器。
>
> - 设计XX(以C为例)语言的词法分析器
>   - 词法规则
>   - 了解所选择编程语言单词符号及其种别值
> - 功能
>   - 输入一个C语言源程序文件demo.c
>   - 输出一个文件tokens.txt，该文件包括每一个单词及其种类枚举值，每行一个单词
> - 提交5个文件
>   - 实验报告（所支持的单词范围，自动机设计，设计思路）
>   - C语言词法分析源程序：source.c（源程序包）
>   - C语言词法分析程序的可执行文件：clang.out/clang.exe
>   - C语言源程序文件：demo.c（实验输入）
>   - 词法分析及结果文件： tokens.txt（实验输出）
> - 同时上传源码至Github

实验思路

构造C语言对应内部表

- 首先构造C语言中关键词表，建立对应token类别码的映射。
  -

{"break","4"}	{"case","5"}	{"char","6"}	{"const","7"}	{"continue","8"}
,{"default","9"},	{"do","10"}	{"double","11"}	{"else","12"}	{"enum","13"}
{"extern","14"}	{"float","15"},	{"for","16"}	{"goto","17"}	{"if","18"}
{"int","19"}	{"long","20"}	{"register","21"}	{"return","22"}	{"short","23"}
{"signed","24"}	{"sizeof","25"}	{"static","26"}	{"struct","27"}	{"switch","28"}
{"typedef","29"},	{"union","30"}	{"unsigned","31"}	{"void","32"}	{"volatile","33"}
{"while","34"}				

- 构造C语言的界符表，此处考虑更多的运算符以及界符，将`++` `--` `+=`等纳入词表中，具体词表如下：

- <table>
<thead>
<tr><th>{">=","35"}</th><th>{"<=","36"}</th><th>{"==","37"}</th><th>{"!=","38"}</th><th>{"=","39"}</th></tr>
</thead>
<tbody>
<tr><td>{">","40"}</td><td>{"<","41"}</td><td>{"%","42"}</td><td>{"+","43"}</td><td>{"+=","44"}</td></tr>
<tr><td>{"++","45"}</td><td>{"-","46"}</td><td>{"-=","47"}</td><td>{"--","48"}</td><td>{"*","49"}</td></tr>
<tr><td>{"*=","50"}</td><td>{"/","51"}</td><td>{"/=","52"}</td><td>{"(","53"}</td><td>{")","54"}</td></tr>
<tr><td>{"{","55"}</td><td>{"}","56"}</td><td>{",","57"}</td><td>{";","58"}</td><td>{"[","59"}</td></tr>
<tr><td>{"]","60"}</td><td>{"|“,”61"}</td><td>{"&","62"}</td><td>{"^","63"}</td><td>{"!","64"}</td></tr>
<tr><td>{"<<","65"}</td><td>{">>","66"}</td><td>{"->","67"}</td><td>{".","68"}</td><td>{"#","69"}</td></tr>
<tr><td>{"||","70"}</td><td>{"&&","71"}</td><td></td><td></td><td></td></tr>
</tbody>
</table>

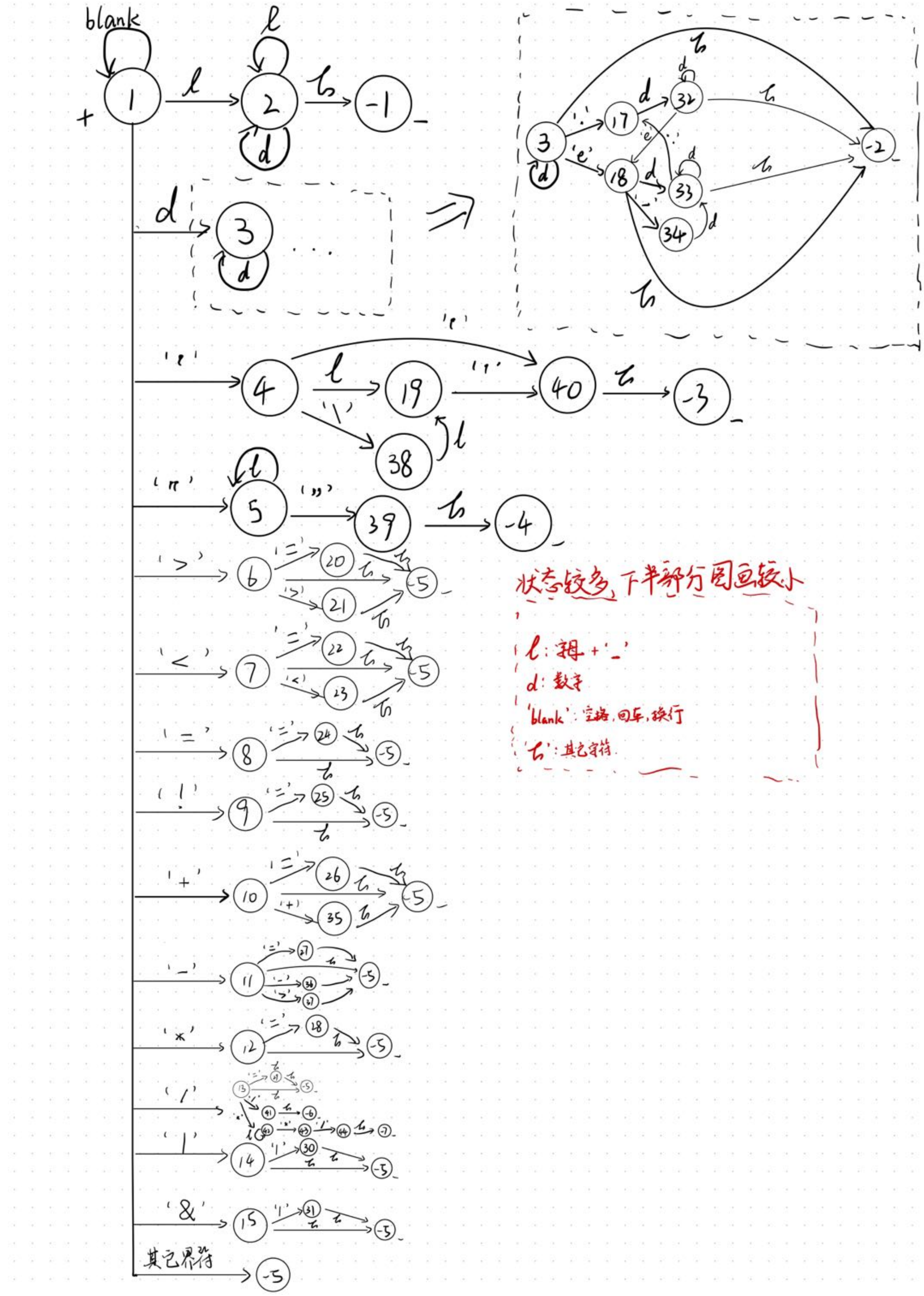
> 在预先建立好上述内部表后，词法分析器就能将关键词与标识符区分开来，同时可以自动识别界符并赋予token，便于后续语法分析的操作使用。

构造词法分析器DFA

为了可以识别不同界符、区分所有标识符、常数变量、字符、字符串，设计DFA如下，在起始状态遇到不同字符时转变为不同状态。为了最终可以区分识别，我将终态设计为6个不同的值，0为异常，即遇到非法词如10. 、10.a$、‘da’等不符合C语言词法的时返回0状态（在图中未画出，异常）；-1为标识符或关键字；-2为常数；-3为字符；-4为字符串；-5为界符；-6为行末注释；-7为行内注释。

DFA总共有51种状态，可以识别绝大多数C语言中会出现的符号，全部合法的标识符，除auto以外的全部关键字，用科学记数法的常数或是普通常数以及字符与字符串。同时还可以做到简单识别一些不合C语言词法的代码，具有一定的可用性。

还需要考虑代码中存在注释的情况，对于/**/注释可能跨行，也可能在行内，也可能跨行，而//注释较为简单，其只能时行末且不能跨行。但识别到他们后都不需要对注释生成token_code. 得到终态后对应选择忽略即可。

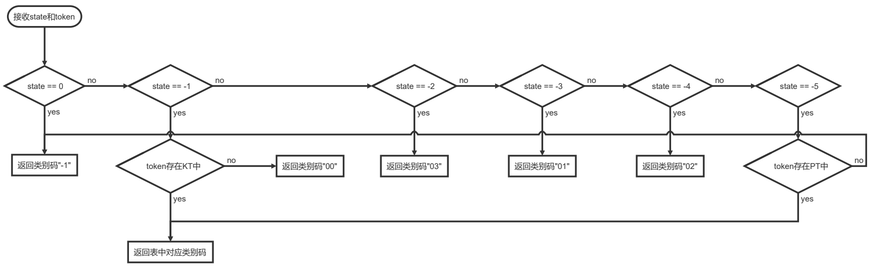


构造token类别码生成器

- 将字符送入DFA分析后，可以得到每个词对应的最终状态：0为异常，即遇到非法词如10.、10.a$、‘da’等不符合C语言词法的时返回0状态；-1为标识符或关键字；-2为常数；-3为字符；-4为字符串；-5为界符。

  根据DFA最后返回的非正状态，可以实现分辨token对应的类别码，从而实现token类别码生成器：

  - token逻辑流程如下：
  - ”-1“为异常，”00“为标识符，”01“为字符，”02“为字符串，”03“为常数，其他对应内部表中所映射的类别码。

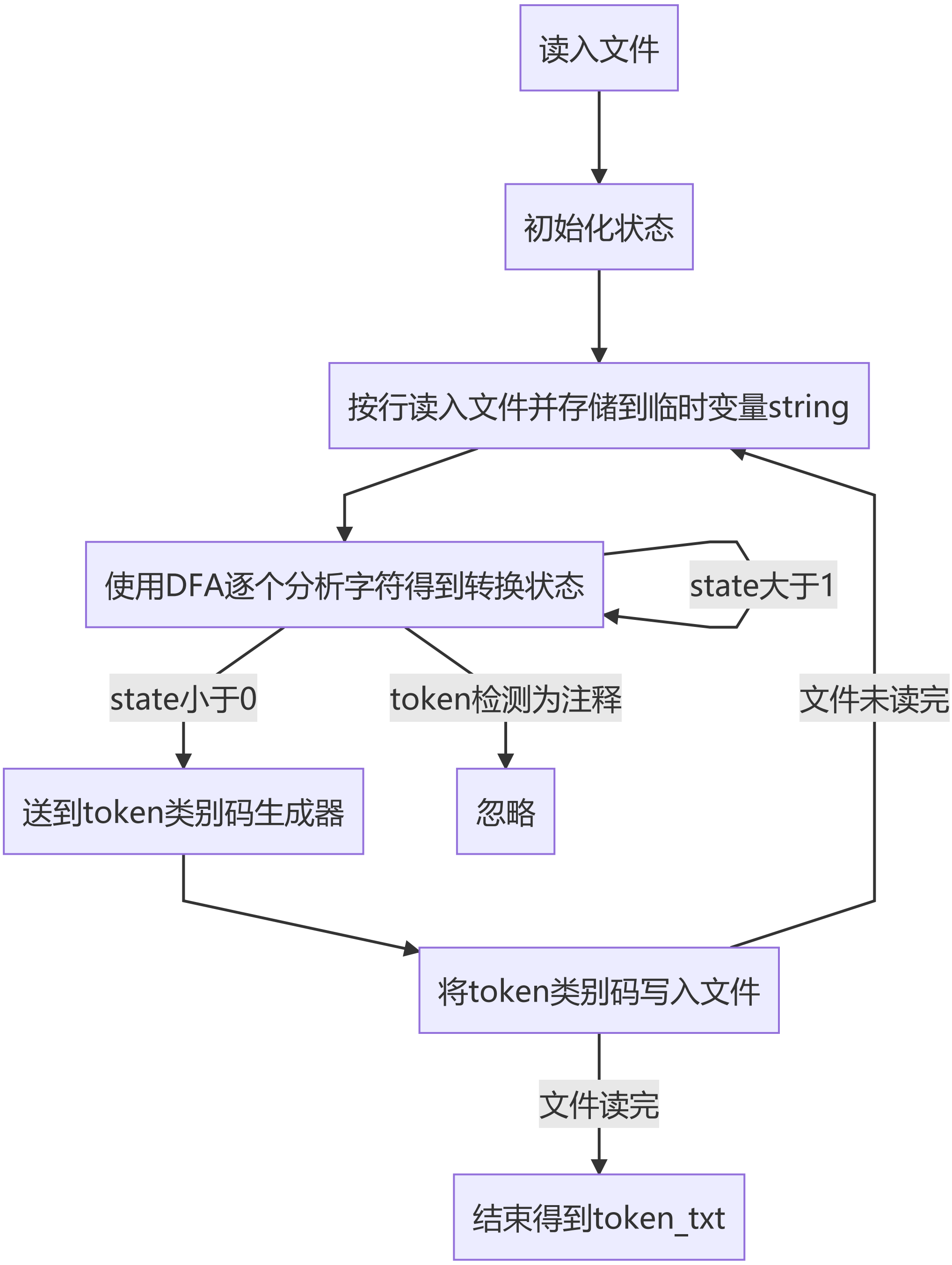


实现词法分析器扫描逻辑

- 词法分析器的扫描逻辑伪代码如下:
  - ```
    Input: string of filename
    Output: file of the token sequence
    Scan Algorithm:
        Open the file
        Initial variable  //(like state:= 1 ...)
        while file is not EOF
        begin
            read the file by line into a temp string
            begin
                c := getchar from the temp string
                state := state_change(state, c)
                if state > 1
                begin
                    token := token + c
                end
                if state == -6 or -7
                begin
                    reinitial the state
                    continue
                end
                if state <= 0
                begin
                    token_code = state2code(state, token)
                    append the token_code to the target file
    ```
    ```

```
                  back to the last char
                  reinitial the state
              end
          end
      end
```

- 流程图如下：



具体实现

实现内部表KT/PT

为了实现能够高效查表并且通过表转换得到对应的token code，内部表使用C++的unordered_map数据结构，该数据结构通过Hash实现高效的查找与映射功能，key-value对的结构天然适用于映射，且unordered_map的key可以为字符串，因此关键词的查找可以通过unordered_map内置的count()函数完成。具体代码实现如下：

```
#include <unordered_map>
using namespace std;

typedef unordered_map<string, string> StringMap;
//关键字表KT
StringMap KT({{"break","04"},{"case","05"},{"char","06"},{"const","07"},
{"continue","08"},{"default","09"},
             {"do","10"},{"double","11"},{"else","12"},{"enum","13"},
{"extern","14"},{"float","15"},
             {"for","16"},{"goto","17"},{"if","18"},{"int","19"},{"long","20"},
{"register","21"},
             {"return","22"},{"short","23"},{"signed","24"},{"sizeof","25"},
{"static","26"},{"struct","27"},
             {"switch","28"},{"typedef","29"},{"union","30"},{"unsigned","31"},
{"void","32"},
             {"volatile","33"},{"while","34"}});
//符号表PT
StringMap PT({{">=","35"},{"<=","36"},{"==","37"},{"!=","38"},{"=","39"},
{">","40"},{"<","41"},{"%","42"},
             {"+","43"},{"+=","44"},{"++","45"},{"-","46"},{"-=","47"},{"--
","48"},{"*","49"},{"*=","50"},
             {"/","51"},{"/=","52"},{"(","53"},{")","54"},{"{","55"},{"}","56"},
{",","57"},{";","58"},
             {"[","59"},{"]","60"},{"|","61"},{"&","62"},{"^","63"},{"!","64"},
{"<<","65"},{">>","66"},
             {"->","67"},{".","68"},{"#","69"},{"||","70"},{"&&","71"}});
```

这两个表属于静态存储的内部表，不会随着扫描器扫描源文件的过程而变化，所以在扫描器实例化前就会取到内存，随程序运行生成。

实现扫描器类

为了方便以后编译器的实现，我将词法分析器作为编译器的一个模块封装成类，并实现词法分析器对应功能。

类的结构

Scanner类的私有成员变量为C++词法对应DFA，标识符表`identify_T`，字符表`char_T`，字符串表`string_T`，常数表`constant_T`。

公有成员变量为一些功能函数，对应着词法分析器的功能，于下文仔细分析。

```
//便于状态变化
typedef unordered_map<string, int> CharMap;
class Scanner
{
    private:
        vector<CharMap> DFA;
        StringMap identify_T;
```

```
        StringMap char_T;
        StringMap string_T;
        StringMap constant_T;

    public:
        void build_DFA();
        Scanner(){
            build_DFA();
        }
        void scan(string file_name);
        int state_change(int state, char c);
        string state2code(int state, string token);
};
```

功能函数

- `void build_DFA();`

  `build_DFA` 函数是要将在实验思路中构思的DFA以表格形式存储在Scanner类中，以便作为词法分析过程中状态转换的依据。首先，使用 `vector` 数据结构作为表的容器，vector的下标对应状态序号，而vector的内部元素我们可以像内部表一样使用 `unordered_map` 的数据结构建立引起状态变换因子与状态的映射，但这里实现过程的细节我们可以有多种选择，我考虑过直接使用字符作为状态变换因子，但考虑Scanner实例化的过程若直接使用字符作为状态变换因子会得到一张极大的DFA变换表格，对内存并不友好，并且搜索效率与直接switch-case判断相仿，所以我使用压缩状态因子的方式，使用字符串指代一类状态变换因子简化DFA表格，减小DFA表格大小，通过外部的判断对应状态转换因子。具体实现如下：

```
void Scanner::build_DFA(){
    CharMap state1({{"blank", 1}, {"letter", 2}, {"digit", 3}, {"single
quote", 4}, {"double quote", 5}, {"greater", 6}, {"less", 7},
                   {"equal", 8}, {"not", 9}, {"plus", 10}, {"minus", 11},
{"multiply", 12}, {"divide", 13}, {"or", 14}, {"and", 15},
                   {"symbol", 16}});
    CharMap state2({{"letter", 2}, {"digit", 2}, {"other", 0}, {"end",
-1}});
    CharMap state3({{"digit", 3}, {"dot", 17}, {"e", 18},{"other", 0},
{"end", -2}});
    CharMap state4({{"char", 19},{"trans", 38}, {"single quote", 40}});
    CharMap state5({{"char", 5}, {"double quote", 39}});
    CharMap state6({{"equal", 20}, {"greater", 21}, {"end", -5}, {"other",
0}});
    CharMap state7({{"equal", 22}, {"less", 23}, {"end", -5}, {"other",
0}});
    CharMap state8({{"equal", 24}, {"end", -5}, {"other", 0}});
    CharMap state9({{"equal", 25}, {"end", -5}, {"other", 0}});
    CharMap state10({{"equal", 26}, {"plus", 35},{"end", -5}, {"other",
0}});
    CharMap state11({{"equal", 27}, {"minus", 36}, {"greater", 37}, {"end",
-5}, {"other", 0}});
    CharMap state12({{"equal", 28}, {"end", -5}, {"other", 0}});
    CharMap state13({{"equal", 29}, {"divide", 41}, {"star", 42}, {"end",
-5}, {"other", 0}});
    CharMap state14({{"or", 30}, {"end", -5}, {"other", 0}});
    CharMap state15({{"and", 31}, {"end", -5}, {"other", 0}});
    CharMap state16({{"end", -5}, {"other", 0}});
    CharMap state17({{"digit", 32}, {"other", 0}});
```

```
    CharMap state18({{"digit", 33}, {"negtive", 34}, {"end", -2}, {"other",
0}});
    CharMap state19({{"single quote", 40}, {"other", 0}});
    CharMap state20({{"end", -5}});
    CharMap state21({{"end", -5}});
    CharMap state22({{"end", -5}});
    CharMap state23({{"end", -5}});
    CharMap state24({{"end", -5}});
    CharMap state25({{"end", -5}});
    CharMap state26({{"end", -5}});
    CharMap state27({{"end", -5}});
    CharMap state28({{"end", -5}});
    CharMap state29({{"end", -5}});
    CharMap state30({{"end", -5}});
    CharMap state31({{"end", -5}});
    CharMap state32({{"digit", 32}, {"e", 18}, {"end", -2}});
    CharMap state33({{"digit", 33}, {"dot", 17},{"end", -2}});
    CharMap state34({{"digit", 33},{"other", 0}});
    CharMap state35({{"end", -5}});
    CharMap state36({{"end", -5}});
    CharMap state37({{"end", -5}});
    CharMap state38({{"char", 19}});
    CharMap state39({{"end", -4}});
    CharMap state40({{"end", -3}});
    CharMap state41({{"end", -6}});
    CharMap state42({{"char", 42}, {"star", 43}});
    CharMap state43({{"divide", 44},{"other", 0}});
    CharMap state44({{"end", -7}});

    DFA.resize(45);
    DFA[1] = state1;
    DFA[2] = state2;
    ...
    DFA[43] = state43;
    DFA[44] = state44;
}
```

因为DFA与内部表一样是既定规则，也是不会随着扫描器扫描源文件的过程而变化，因此在实例化时便完成DFA表格的构建。因此Scanner类的构造函数如下：

```
Scanner(){
    build_DFA();
}
```

- `void scan(string file_name);`

  scan功能函数目的是将源文件读入后，自动生成得到token code 序列文件，代码逻辑与实验思路中伪代码一致。

  具体实现如下：

```
void Scanner::scan(string filename){
    //源文件读入缓冲区
    ifstream infile;
    infile.open(filename, ios::in);
    //初始化状态变量
```

```
    int state = 1;
    string line;
    string token_code;
    string token = "";
    //按行读取源文件内容，自动处理换行
    while(getline(infile, line)){
        int i = 0;
        //逐字符分析词法
        while(i < line.size()){
            char c = line[i];
            //通过DFA完成状态转换
            state = state_change(state, c);
            //非终态，继续读入字符
            if(state > 1){
                token += c;
            }
            // //类型注释
            if(state == -6){
                state = 1;
                token = "";
                break;
            }
            //终态生成token_code并将字符指针回退一个字符后重新分析
            if(state <= 0){
                /*注释*/
                if(state == -7){
                    state = 1;
                    token = "";
                    seq = "";
                    continue;
                }
                //完成state到token_code的转换
                token_code += "<" + token + ", " + state2code(state, token)
+ ">" + "\n";
                state = 1;
                token = "";
                i--;
            }
            i++;
        }
    }
    infile.close();
    //生成目标文件
    ofstream outfile("token.txt", ios::out);
    outfile << token_code;
    outfile.close();
}
```

接下来解释`scan`函数用到的状态转换函数`state_change`与token_code生成函数`state2code`

- `int state_change(int state, char c);`

  该函数主要是识别字符c对应类别，交给DFA得到下一转换，完成状态转换，基本是switch-case组成的判断，判断该状态下DFA规定的状态变换以及字符类别。

```
int Scanner::state_change(int state, char c){
```

```
    int next_state = 0;
    switch (state)
    {
        case 1:
            //空白字符跳过
            if(c == ' ' || c == '\n'){
                next_state = DFA[state]["blank"];
            }
            //所有字母以及'_'字符(合法的变量起始字符)
            else if(c == '_' || (c >= 65 && c <= 90) || (c >= 97 && c <=
122)){
                next_state = DFA[state]["letter"];
            }
            //遇到数字
            else if(c >= 48 && c <= 57){
                next_state = DFA[state]["digit"];
            }
            //遇到单引号
            else if(c == '\''){
                next_state = DFA[state]["single quote"];
            }
            //遇到双引号
            else if(c == '"'){
                next_state = DFA[state]["double quote"];
            }
            //下列转换比较雷同，可以参照实现的DFA对应观察状态的转换，符合实验思路中设计
的DFA
            else if(c == '>'){
                next_state = DFA[state]["greater"];
            }
            else if(c == '<'){
                next_state = DFA[state]["less"];
            }
            else if(c == '='){
                next_state = DFA[state]["equal"];
            }
            else if(c == '!'){
                next_state = DFA[state]["not"];
            }
            else if(c == '+'){
                next_state = DFA[state]["plus"];
            }
            else if(c == '-'){
                next_state = DFA[state]["minus"];
            }
            else if(c == '*'){
                next_state = DFA[state]["multiply"];
            }
            else if(c == '\\'){
                next_state = DFA[state]["divide"];
            }
            else if(c == '|'){
                next_state = DFA[state]["or"];
            }
            else if(c == '&'){
                next_state = DFA[state]["and"];
            }
            else{
```

```
                next_state = DFA[state]["symbol"];
            }
            break;
        case 2:
            if(c == '_' || (c >= 65 && c <= 90) || (c >= 97 && c <= 122)){
                next_state = DFA[state]["letter"];
            }
            else if(c >= 48 && c <= 57){
                next_state = DFA[state]["digit"];
            }
            else{
                next_state = DFA[state]["end"];
            }
            break;
        ...
        case 42:
            if(c == '*'){
                next_state = DFA[state]["star"];
            }
            else{
                next_state = DFA[state]["char"];
            }
            break;
        case 43:
            if(c == '/'){
                next_state = DFA[state]["divide"];
            }
            else{
                next_state = DFA[state]["other"];
            }
            break;
        case 44:
            next_state = DFA[state]["end"];
            break;

        default:
            next_state = DFA[state]["other"];
            break;
    }
    return next_state;
}
```

为了做到报告篇幅尽量简洁，这里就不放出所有状态的转换代码，只展示部分代码供参考，完整代码可以查看源代码，完成了实验思路中设计的DFA全部规则。

- `string state2code(int state, string token);`

`state2code` 函数是将终态状态翻译转换得到对应的token_code，安装token类别码生成器逻辑实现。

具体代码如下：

```
string Scanner::state2code(int state, string token){
    string token_code = "";
    switch (state)
    {
        //state == -1,可以为标识符或关键字
```

```
        case -1:
            //若token存在于KT中，返回关键字对应类别码
            if(KT.count(token)){
                token_code = KT[token];
            }
            //否则返回标识符类别码并更新标识符表
            else{
                if(identify_T.count(token) == 0)
                    identify_T[token] = "00";
                token_code = identify_T[token];
            }
            break;
        //state == -2,为常数
        case -2:
            //返回常数类别码并更新常数表
            if(constant_T.count(token) == 0)
                constant_T[token] = "03";
            token_code = constant_T[token];
            break;
        //state == -3,为字符
        case -3:
            //返回字符类别码并更新字符表
            if(char_T.count(token) == 0)
                char_T[token] = "01";
            token_code = char_T[token];
            break;
        //state == -4,为字符串
        case -4:
            //返回字符串类别码并更新字符串表
            if(string_T.count(token) == 0)
                string_T[token] = "02";
            token_code = string_T[token];
            break;
        //state == -5, 为符号
        case -5:
            //查PT表返回对应类别码
            if(PT.count(token)){
                token_code = PT[token];
            }
            //若不存在于PT表中，返回非法类别码
            else{
                token_code = "-1";
            }
            break;
        //默认状态0返回非法类别码
        default:
            token_code = "-1";
            break;
    }
    return token_code;
}
```

上述基本Scanner类的全部实现，通过scan函数可以完成实验要求的功能。

主程序实现

实现了Scanner类后，主程序实现就比较简单，引入Scanner模块，调用相关功能函数即可。

具体实现如下：

```
#include "scanner.h"
#include <string>
using namespace std;

int main(int argc, char** argv){
    //实例化Scanner
    Scanner scanner;
    //若有输入参数（文件名）
    if(argv[1] != NULL){
        string filename = argv[1];
        scanner.scan(filename);
    }
    //否则在程序运行时输入文件名
    else{
        string filename;
        cin >> filename;
        scanner.scan(filename);
    }
}
```

主程序主要作为程序入口，接收文件名输入，得到token.txt文件输出。

实验结果

- 词法分析器支持单词范围：
    - KT中31个关键字
    - PT中37个符号
    - 字母or_开头的合法标识符
    - 常数，小数，科学记数法（负数属于语法分析阶段完成）
    - 字符如'a'或'\n'等
    - 字符串“abc”等
    - 支持//行末注释，以及/**/行内/跨行注释。

- 测例一

  demo.c如下：

```
/*aaaa
aaaa*/
int main(void){
    int a = 1, /*abc*/ d = 2e-1.23, c; //abc
    float b = 21.35;
    if(a <= d){
        c = a;
        a = d;
        d = c;
```

```
    }
    char ch[10] = "ok";
    char x, y = 'a';
    c = a + d;
    for(int i = 0 ; i < n ; ++i)
        d--;
    return 0;
}
```

结果如下：(main不是C语言的关键字)，符合KT，PT以及DFA的设计。

```
<int, 19><main, 00><(, 53><void, 32><), 54><{, 55><int, 19><a, 00><=, 39><1, 03><,, 57><d, 00><=, 39><2e-1.23, 03><,, 57><c, 00><;, 58><float, 15><b, 00><=, 39><21.35, 03><;, 58><if, 18><(, 53><a, 00><<=, 36><d, 00><), 54><{, 55><c, 00><=, 39><a, 00><;, 58><a, 00><=, 39><d, 00><;, 58><d, 00><=, 39><c, 00><;, 58><}, 56><char, 06><ch, 00><[, 59><10, 03><], 60><=, 39><"ok", 02><;, 58><char, 06><x, 00><,, 57><y, 00><=, 39><'a', 01><;, 58><c, 00><=, 39><a, 00><+, 43><d, 00><;, 58><for, 16><(, 53><int, 19><i, 00><=, 39><0, 03><;, 58><i, 00><<, 41><n, 00><;, 58><++, 45><i, 00><), 54><d, 00><--, 48><;, 58><return, 22><0, 03><;, 58>
```

- 测例二

来自leetcode题库的随机一题([2100. 适合打劫银行的日子](#)):

```
int* goodDaysToRobBank(int* security, int securitySize, int time, int* returnSize) {
    int * left = (int *)malloc(sizeof(int) * securitySize);
    int * right = (int *)malloc(sizeof(int) * securitySize);
    memset(left, 0, sizeof(int) * securitySize);
    memset(right, 0, sizeof(int) * securitySize);
    for (int i = 1; i < securitySize; i++) {
        if (security[i] <= security[i - 1]) {
            left[i] = left[i - 1] + 1;
        }
        if (security[securitySize - i - 1] <= security[securitySize - i]) {
            right[securitySize - i - 1] = right[securitySize - i] + 1;
        }
    }

    int * ans = (int *)malloc(sizeof(int) * securitySize);
    int pos = 0;
    for (int i = time; i < securitySize - time; i++) {
        if (left[i] >= time && right[i] >= time) {
            ans[pos++] = i;
        }
    }
    free(left);
    free(right);
    *returnSize = pos;
    return ans;
}
```

结果如下：结果也符合预期，表现该词法分析器的可用性。

```
<int, 19><*, 49><goodDaysToRobBank, 00><(, 53><int, 19><*, 49><security, 00>
<,, 57><int, 19><securitySize, 00><,, 57><int, 19><time, 00><,, 57><int, 19>
<*, 49><returnSize, 00><), 54><{, 55><int, 19><*, 49><left, 00><=, 39><(,
53><int, 19><*, 49><), 54><malloc, 00><(, 53><sizeof, 25><(, 53><int, 19><),
54><*, 49><securitySize, 00><), 54><;, 58><int, 19><*, 49><right, 00><=, 39>
<(, 53><int, 19><*, 49><), 54><malloc, 00><(, 53><sizeof, 25><(, 53><int,
19><), 54><*, 49><securitySize, 00><), 54><;, 58><memset, 00><(, 53><left,
00><,, 57><0, 03><,, 57><sizeof, 25><(, 53><int, 19><), 54><*, 49>
<securitySize, 00><), 54><;, 58><memset, 00><(, 53><right, 00><,, 57><0, 03>
<,, 57><sizeof, 25><(, 53><int, 19><), 54><*, 49><securitySize, 00><), 54>
<;, 58><for, 16><(, 53><int, 19><i, 00><=, 39><1, 03><;, 58><i, 00><<, 41>
<securitySize, 00><;, 58><i, 00><++, 45><), 54><{, 55><if, 18><(, 53>
<security, 00><[, 59><i, 00><], 60><<=, 36><security, 00><[, 59><i, 00><-,
46><1, 03><], 60><), 54><{, 55><left, 00><[, 59><i, 00><], 60><=, 39><left,
00><[, 59><i, 00><-, 46><1, 03><], 60><+, 43><1, 03><;, 58><}, 56><if, 18>
<(, 53><security, 00><[, 59><securitySize, 00><-, 46><i, 00><-, 46><1, 03>
<], 60><<=, 36><security, 00><[, 59><securitySize, 00><-, 46><i, 00><], 60>
<), 54><{, 55><right, 00><[, 59><securitySize, 00><-, 46><i, 00><-, 46><1,
03><], 60><=, 39><right, 00><[, 59><securitySize, 00><-, 46><i, 00><], 60>
<+, 43><1, 03><;, 58><}, 56><}, 56><int, 19><*, 49><ans, 00><=, 39><(, 53>
<int, 19><*, 49><), 54><malloc, 00><(, 53><sizeof, 25><(, 53><int, 19><),
54><*, 49><securitySize, 00><), 54><;, 58><int, 19><pos, 00><=, 39><0, 03>
<;, 58><for, 16><(, 53><int, 19><i, 00><=, 39><time, 00><;, 58><i, 00><<,
41><securitySize, 00><-, 46><time, 00><;, 58><i, 00><++, 45><), 54><{, 55>
<if, 18><(, 53><left, 00><[, 59><i, 00><], 60><>=, 35><time, 00><&&, 71>
<right, 00><[, 59><i, 00><], 60><>=, 35><time, 00><), 54><{, 55><ans, 00><[,
59><pos, 00><++, 45><], 60><=, 39><i, 00><;, 58><}, 56><}, 56><free, 00><(,
53><left, 00><), 54><;, 58><free, 00><(, 53><right, 00><), 54><;, 58><*, 49>
<returnSize, 00><=, 39><pos, 00><;, 58><return, 22><ans, 00><;, 58>
```

实验心得

本次实验主要是实现一个词法分析器，在自动机基础的前置知识以及基本的词法分析知识的指导下，先设计出C语言词法分析的DFA后，再按词法分析器的功能设计对应框架，在有规划的设计下实现词法分析器代码就稍微简单一点。通过本次实验，对词法分析器的理解以及对状态之间的转换梳理也更加熟练，能够设计出一个具有一定可用性的词法分析器还是比较有成就感的。

实验github：https://github.com/Vanssssry/Compiler_Lab