中山大學
SUN YAT-SEN UNIVERSITY

年级&班级	计算机教务1班	专业(方向)	计算机科学与技术（超算）
学号	19335043	姓名	方梓健

实验Github地址：https://github.com/FangFangHawk/C_p_Experiment_1

实验内容

1. **基础题目**

   完成一个词法分析器

实验描述

手动设计实现，或者使用Lex实现词法分析器

实验实现报告

实验任务分析：

要实现一个C语言的词法分析，首先我们要确定C语言相关的单词符号及其种别值。

- 我们将C语言的相关词语符号，主要分为以下五类，作为基本的状态：界符、运算符、位操作运算符，数字、标识符，关键字，以及特殊字符（注释）。
- 基于上述的分类，我们再根据每一个字符读取的状态，对源程序中的C语言字符进行读取，并构建DFA自动机。
- 在获得自动机之后，我们就可以根据自动机，建立基本的代码模块，具体的实现思路如下
  - 字符读取模块
  - 字符读取后的状态转换
  - 每个字符段读取终止后，根据字符段此时的状态，获取对应的编码，输入到解码模块，获得对应的种类枚举值，并输入到词法分析保存的容器中。
  - 最终完成文件输出
- 附加：在代码中，词法还需要一定的错误检查能力，当检查到代码源程序出错的时候，能够返回相关错误，并输出错误所在行数。

实验过程与核心代码

首先，我们需要完成读取C语言的相关分类，我们对其的具体分类以及相关状态标识设定如下：

分类：

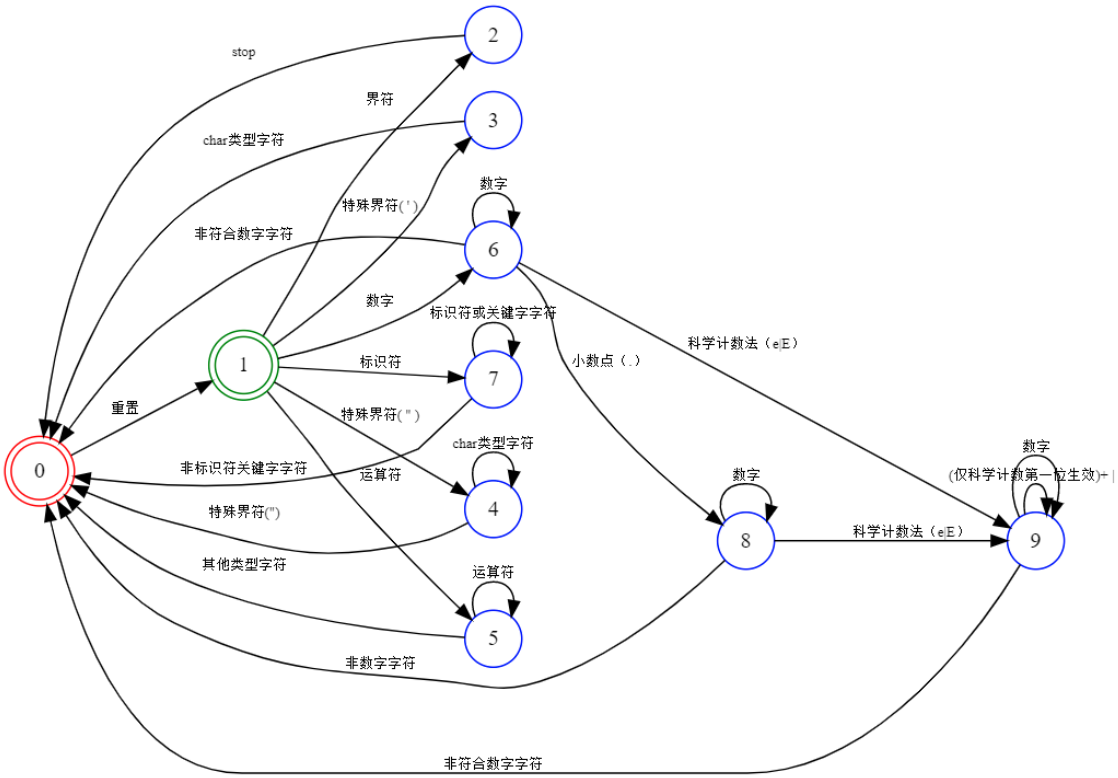
- 界符：（{}、[]、()、;、 " " 等）
- 运算符：（+，-，*，/，+=, ++，--，*=，-=，/=, ==, &, &&, |, | |等），主要包含的运算符类别有（算术运算符、关系运算符、逻辑运算符、位操作运算符、赋值运算符）

- 数字：我们主要能够完成的数字辨别包含了二进制数字，八进制数字，十进制数字，十六进制数字，小数，以及正确的科学计数法
- 标识符：C语言规定，标识符只能由字母（A~Z, a~z）、数字（0~9）和下划线（_）组成，并且第一个字符必须是字母或下划线，不能是数字
- 关键字：c语言本身定义的常用关键字集合。
- 注释：C语言支持单行注释和多行注释：
  - 单行注释以`//`开头，直到本行末尾（不能换行）；
  - 多行注释以`/*`开头，以`*/`结尾，注释内容可以有一行或多行。

状态划分：

- 初始字符读取状态为state == 1
- 由于我们需要将单引号以及双引号中的字符单独取出，放置在对应的cT字符与sT字符的容器中，所以归属于界符的状态主要有三种：
  - state == 2： 此时界符为普通界符
  - state == 3：代码此时读取到的界符为单引号，需进行进一步处理
  - state == 4： 代码此时读取到的界符为双引号，需进行进一步处理
- 运算符：state == 5 此时运算继续向下读取
- 数字：由于数字的读取中，加入了一些比较复杂的进制识别，因此数字也需要分为多个状态进行处理
  - state == 6： 当当前字符识别一直为整数（**包括相关进制代表符号 “0b” 或 “0X”等**）的时候，此时state为6
  - state == 8： 当读取字符的过程中，出现小数点，此时将转换到小数状态 state == 8.便于后续解码处理小数。
  - state == 9：当读取字符中，出现科学计数符 ‘e|E‘时候，此时状态转换为9，便于后续解码处理科学计数
- 标识符，关键字：由于这两者的构建条件较为相似，我们置于一起进行读取，在后续解码过程中，再将二者进行进一步划分。
- state == 0：当当前状态需要结束的时候，此时所有状态归为 0 ，完成当前单词的读取，并进入解码模块。具体的结束情况有以下几种（举例）
  - 当前状态转换到另外一种大类状态中：如界符后续读取到数字字符，则当前界符读取结束。等
  - 字符读取到空格等无关字符，则当前字符读取结束

根据上述状态划分，我们构建自动机**DFA**如下：


0
1
2
3
4
5
6
7
8
9
stop
界符
char类型字符
非符合数字字符
特殊界符(')
数字
数字
标识符或关键字字符
标识符
特殊界符(")
char类型字符
运算符
运算符
特殊界符(")
其他类型字符
非数字字符
非标识符关键字字符
重置
小数点（.）
科学计数法（eE）
数字
科学计数法（eE）
数字
(仅科学计数第一位生效)+|
非符合数字字符

相关种类枚举值表

iT标识符	00	cT字符	01	sT字符串	02
cT常数	03				
KT关键字					
auto	04	short	05	int	06
long	07	float	08	double	09
char	10	struct	11	union	12
enum	13	typedef	14	const	15
unsigned	16	signed	17	extern	18
register	19	static	20	volatile	21
void	22	if	23	else	24
switch	25	case	26	for	27
do	28	while	29	goto	30
continue	31	break	32	default	33
sizeof	34	return	35	mian	36
PT界符					
+	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	[	72
]	73	{	74	}	75
'	76	"	77	;	78
,	79	!	80	//	81
/*	82	*/	83	?	84

核心代码编写思路

在上述获得自动机与枚举值表之后，我们就可以根据上述两份表，构建一个我们自己的手动词法分析器

具体的核心代码如下：

核心函数与参数介绍：

```
//参数
vector< pair<string , int> > res;    //保存最后的读取结果
vector<char> cT;                     //保存字符
vector<string> sT;                   //保存字符串
map<string , int> PT;                //PT符号
map<string , int> KT;                //KT关键字
```

```
void init_all_map();                          //初始化符合与关键字的映射表

int state_change(int state , char ch);        //状态转化

int get_state_ch(char ch);                    //获得当前字符的状态

int State2Code(int state_before);             //状态转编码

int parse(int code , string token);           //解析编码

void show();                                  //展示获得的输出内容

bool check_legal_num(string token);           //检查是否为合法的数字

bool all_zero(string token);                  //检查输入全为0的时候可能出现的错误

void Error_print(int state , int line);       //输出错误信息
```

代码运行逻辑：

我们可以从主函数的调用中，来获得代码运行逻辑，并结合该逻辑，依次来介绍词法分析器中的相关函数

主函数

```
int main(){
    init_all_map();                       //初始化符合与关键字的映射表
    ...
    //打开源文件并做好前置处理
    ...
    while (getline(inf , sline)){
        //每次读取文件一行，便于进行处理操作
        for(int i = 0 ; i < sline.size() ; i++){
            ...     //相关检测，比如注释部分
            ch = sline[i];
            state_before = state;
            state = state_change(state , ch);

            if(state > 0){
                token += ch;
            }
            else if(state == 0){
```

```
                state = 1;
                int code =  State2Code(state_before);
                int check_parse = parse(code , token);
                //cout<<check_parse<<" "<<endl;
                if(check_parse < 0){
                    Error_print(check_parse , count_line);
                    return 0;
                }
                state = state_change(state , ch);
                token.clear();
                if(state){
                    token += ch;
                }
                else{
                    state = 1;
                }
            }
            if(state < 0){
                Error_print(state , count_line);
                return 0;
            }
        }
        count_line++;
    }

}
```

- **初始化映射表： init_all_map（）**

  首先是map映射的，这一部分比较简单，实施上只是建立两个 map （unordered_map），完成上述种类枚举值表的映射。这一部分比较简单，在这里不再进行赘述。

- **状态转化函数：state_change(int state , char ch);**

  在这一部分中，我们需要按照前面展示的状态转化DFA，来完成对于函数的构建，由于状态类别比较多，我们针对性的介绍3种状态的转化过程。

  为了方便进行函数的跳转，我们使用的switch关键字。

  参数介绍

  1. state 当前所处状态
  2. ch 当前字符
  3. 返回值（int），返回当前所应返回的状态

  状态转化介绍

  - 1->other：当前状态为1，默认为此时该字符前面的所有字符已经完成状态转换与编码识别，此时直接根据当前的ch字符，调用 **get_state_ch(char ch);** 获得当前字符的状态即可。
  - 2-0：我们观察状态转化表，当当前状态为2的时候，说明此时为普通界符，由于我们的界符表中的所有字符均为单个字符，因此当获得该字符时，直接返回0即可。
  - 6状态转化：
    1. 6状态的转化有多种可能，读取到字符为e的时候，此时为数字状态读取到科学计数法的，此时状态转为9，进行科学计数。
    2. 读取ch为数字，此时返回6继续读取
    3. 读取非符合状况字符，此时返回0，结束数字读取。

相关代码展示

```
int state_change(int state , char ch){
    ...
    //部分状态转化需要借助全局函数进行转换；
    switch (state)
    {
        case 1:
            return get_state_ch(ch);
            break;
        case 2:
            return 0;
            break;
        ... //其他状态
        case 6:
            if(ch == 'e' || ch == 'E'){
                return 9;
            }   //跳转科学计数法
            now_state = get_state_ch(ch);
            if(now_state <= 5){
                if(now_state < 0){
                    return now_state;    //返回错误值
                }
                return 0;
            }
            if(now_state == 8){
                return 8;
            }//跳转小数

            return 6;
            break;
            ...
    }
    return 0;
}
```

- int get_state_ch(char ch); 获取当前字符对应的state值

  这部分实现比较简单，我们只需要将字符进行简单的分类即可，具体分类可以直接参考代码注释了解：

```
int get_state_ch(char ch){
    if((int)ch > 125 || (int)ch < 32){
        return -1;
    }//不能识别字符
    if(ch == ' ' || ch == '\n'){\
        return 0;
    }//此时结束读取
    string now;
    now += ch;
    int now_code = PT[now];//检查是否为PT字符
    if(now_code){
        if(now_code > 69){
            if(now_code == 76)  //特殊处理字符
                return 3;
            else if(now_code == 77)//特殊处理字符
                return 4;
```

```
            else if(now_code == 100)
                return 8; //处理小数
            return 2;
        }
        return 5;
    }
    if( ch >= '0' && ch <= '9'){
        return 6;
    }//处理数字
    return 7;//处理标识符或者关键字
}
```

- int State2Code(int state_before); 状态转编码

  这部分的内容也比较简单，根据当前的状态，转为对应的code编码即可。在此不进行赘述，主要讲述解码部分。

- int parse(int code , string token);

  根据code值，完成解码。在这一部分中，我完成了对于注释部分的处理，我借助全局函数，若此时识别token保存到的字符，为注释符号**/***，则根据判别，完成对于相关注释的处理

  而//字符的判断时候，则放在函数识别界符种完成处理，只需忽略行后续字符即可。

```
    if(flag_all_anno){
        if(PT[token] != 83)
            return 0;    //当注释符号为/*时忽略行后续字符直到读取到*/
        flag_all_anno = 0;
        return 0;
    }
```

  其他编码处理：

  比较简单的处理，比如处理界符，只需要直接调用映射表，核心代码如下：

```
    ...
    switch (code)
    {
        case 1:
            ans = make_pair(token , PT[token]);
            res.push_back(ans);
            break;
     ...
```

  而复杂一点的处理，比如数字，由于我们构建了多个进制处理，因此这一部分比较复杂，首先我们建立了几个函数，用来检查数字的正确性：

```
bool check_legal_num(string token);//检查是否为合法的数字
int dif_base(char first , char second);  // 根据前两位计数，获得此时的数字进制
bool all_zero(string token); //检查输入全为0的时候可能出现的错误
```

  核心代码逻辑如下：

```
        ...
        case 5:
            if(check_legal_num(token)){
                return -4;
            }
            ans = make_pair(token , 3);
            res.push_back(ans);
            break;
        ...
```

此时case到code编码为5，按照数字解析，调用函数 check_leagal_num 检查是否为合法数字：

当为合法的时候，则正常存入数字，若不合法，则返回此时的错误编码，结束读取。

核心检测函数如下：

- 先检查数字字符串的size： 当数字长度小于2时，由于进入时已经确保第一位数为1，则一定正确
- 检查数字字符串的前两位，返回将当前数字字符所用进制
- 根据不同的进制，确定数字所能取值的范围，并完成此时数字检查。

```
bool check_legal_num(string token){
    if(token.size() < 2){
        return false;
    }

    int dif = dif_base(token[0] , token[1]);

    if((dif != 10 && dif != 8) && token.size() < 3 ||(all_zero(token))){
        return true;
    }
    switch (dif)
    {
    case 0:
        return true;
    case 10:
        for(int i = 1 ; i < token.size() ; i++)
            if(!isdigit(token[i]))
                return true;
        break;
    case 8:

        for(int i = 1 ; i < token.size() ; i++){
            if( token[i] > '8' || token[i] < '0')
                return true;
        }
        break;
    case 16:
        for(int i = 2; i < token.size() ; i++){
            if(!(isdigit(token[i])) &&( (token[i] < 'a' || token[i] > 'f')
|| (token[i] < 'A' || token[i] > 'F') ))
                return true;
        }
        break;
    default:
        break;
    }
    return false;
```

```
}
```

核心函数的介绍如上，当上述函数按照正确的逻辑组合起来的时候，此时我们就完成了一个基本的词法分析器

基本的实验效果详见实验结果内容。

实验结果

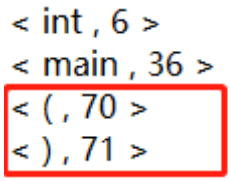
基本功能展示：

我们按照所分的几个分类来进行展示：

界符

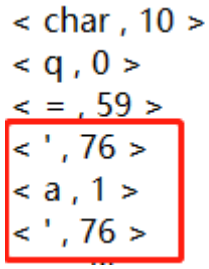
1. 普通界符

```
int main()
```



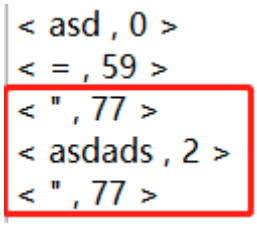
2. 单引号界符

```
char q = 'a';
```



3. 双引号界符

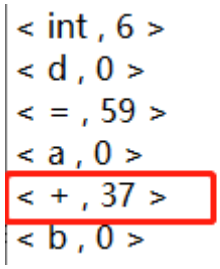
```
string asd = "asdads";
```



运算符

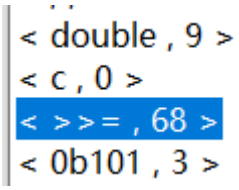
1. 算术运算符

```
int d = a + b;
```



2. 位运算符

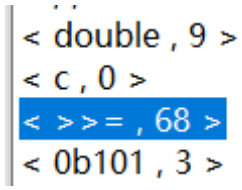
```
double c >>= 0b101;
```



数字

1. 二进制：

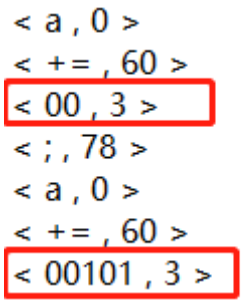
```
double c >>= 0b101;
```



2. 八进制：

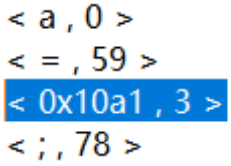
   特殊：00 ， 000将报错

```
a += 00;
a += 00101
```



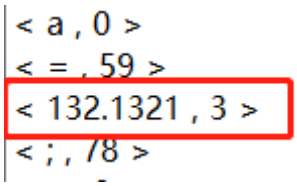
3. 十六进制：

```
a = 0x10a1;
```



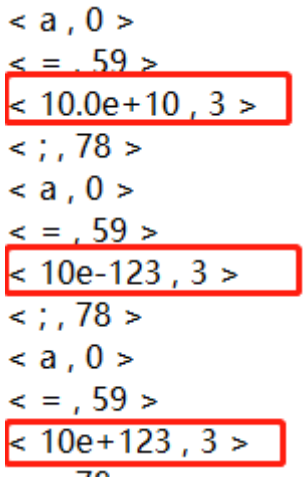
4. 小数：

```
int a  = 132.1321;
```



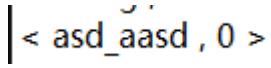
5. 科学计数法：

```
a = 10.0e+10;
a = 10e-123;
a = 10e+123;
```



标识符：

上述中的a就是标识符，其他复杂一些的标识符如下，包含了下划线和数字：



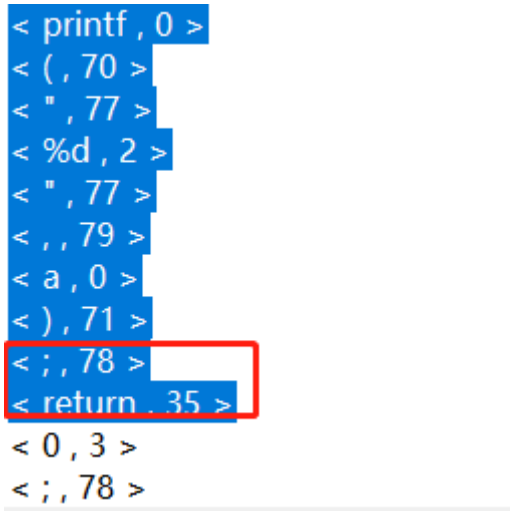
关键字：

上述的int与char等各类型即为关键字展示，在此不在过多展示。

注释

单行注释：

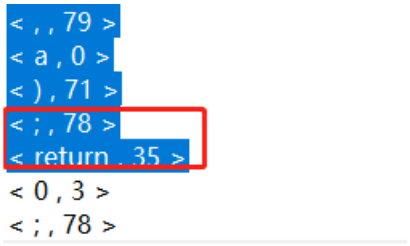
```
printf("%d" , a); //This is a test note
```



他将跳过后续内容，直接到达下一行。

多行注释：

```
printf("%d" , a); //This is a test note
/*This is a test note
asdad //nothing thing
asdsaddsa
*/
return 0;
```



他将跳过后续内容，直接到达下一行。

错误检测

主要检测字符：由于检测到错误是会直接退出，这里主代码均注释，检查时可以将代码注释去掉。

```
int main()
{
//错误检测      //error1
//char b = 'asd';//error2
//a --=- b; //error3
//a = 0c0;  //error4
//a = 12.;  //error6
//int a = 1e;   //error7
}
```

Error1：使用不可识别字符

```
错误检测
```

```
Yes(1) /t NO(0)
1
Error : Your input contains illegal characters in Line : 4 .
```

Error2: char 字符表示却包含了多于一个字符

```
char b = 'asd';
```

```
Error : Contains multiple characters in a variable of type "char" in Line : 5 .

Process returned 0 (0x0)   execution time : 18.522 s
Press any key to continue.
```

Error3：使用不能识别符号“---=-”

```
a --=- b;   //error3
```

```
Error : Incorrect use of related symbols in Line : 6 .

Process returned 0 (0x0)   execution time : 5.256 s
Press any key to continue.
```

Error4： 不合法数字

```
a = 0c0;    //error4
```

```
8
Error : The representation of numbers is wrong in Line : 6 .
```

Error6：不合法小数

```
a = 12..;   //error6
```

```
1
Error : Incorrect decimal representation is used in the statement in Line : 7 .

Process returned 0 (0x0)   execution time : 3.578 s
Press any key to continue.
```

Error 7 ： 不合法科学计数

```
int a = 1e; //error7
```

Error5：为不可识别code编码，用于防止意外情况发生。

实验总结与感想

本次词法分析器，花费了我挺多心思，结合了课上学到的关于词法的知识，与状态机的构建，最终完成了该词法分析器，为了让该词法分析器更加完善，多考虑与处理了不同情况，比如关于数字的读取保存，就考虑了不同的进制，小数以及科学计数，让这个词法分析器的功能更加完善，更便于人们的使用。

总的来说，个人觉得本次作业的收获还是非常多的，让我对于C语言的编译处理过程有了更加深入的了解，也巩固了课堂学习到的知识，收获很多，也对后续实验课程更加充满了期待。