

▼ 语法分析器

▪ 一、实验内容

▼ 二、程序输入输出展示

▪ 2.1 输入展示

▪ 2.2 输出展示

▪ 2.3 报错提示

▼ 三、自定义的REs和DFA

▪ （一）自定义的词法单元及其REs

▪ （二）合并化简后的DFA

▼ 四、代码实现

▪ （一）重要变量

▪ （二）各种词法单元判别函数

▪ （三）循环迭代过程

▪ 五、遇到的问题

▪ 六、实验心得

语法分析器

一、实验内容


自实现词法分析器：

1. 定义一组正则表达式（REs）：定义一组能够匹配不同类型的词法单元的正则表达式。
2. 将REs转换为非确定有限自动机（NFAs）
3. 合并NFAs为单一的NFA：将所有的NFAs合并为一个单一的NFA，以便能够同时处理多个词法单元。
4. 将NFA转换为具有最小状态数的确定有限自动机（DFA）
5. 编程实现基于DFA的词法分析器：根据DFA的状态转换表和动作定义，编写代码实现词法分析器。词法分析器应能够从输入程序中读取字符并根据DFA的状态转换进行匹配，并生成相应的词法单元标记。
6. 测试词法分析器：编写测试用例，包括各种类型的输入程序，以确保词法分析器能够正确地识别和生成适当的词法单元。
7. 错误处理：在词法分析器中实现适当的错误处理机制，能够检测和报告词法错误，如不匹配的字符、非法的标识符等。

二、程序输入输出展示

2.1 输入展示

以下输入以.txt文件的格式读入程序

 program.txt × +

文件 编辑 查看

```
#define GOOD 2000
// This is the c test file
int y=t/5;
int main(){
int x ;
bool change=FALSE;
x = x + 5 ;
}
/*
    Multi line Comment
*/

// Single line Comment
for(int i = 0, i <= 4; i++){
    x += i;
}
    x++;
string str="jzx"
for(const auto& op : opers)
    if (op == str)
        return true;

54dsa
_sda = "gdsdg";
```

最后两行为错误词法，作测试用

2.2 输出展示

词法分析器分析上述输入程序结果如下：

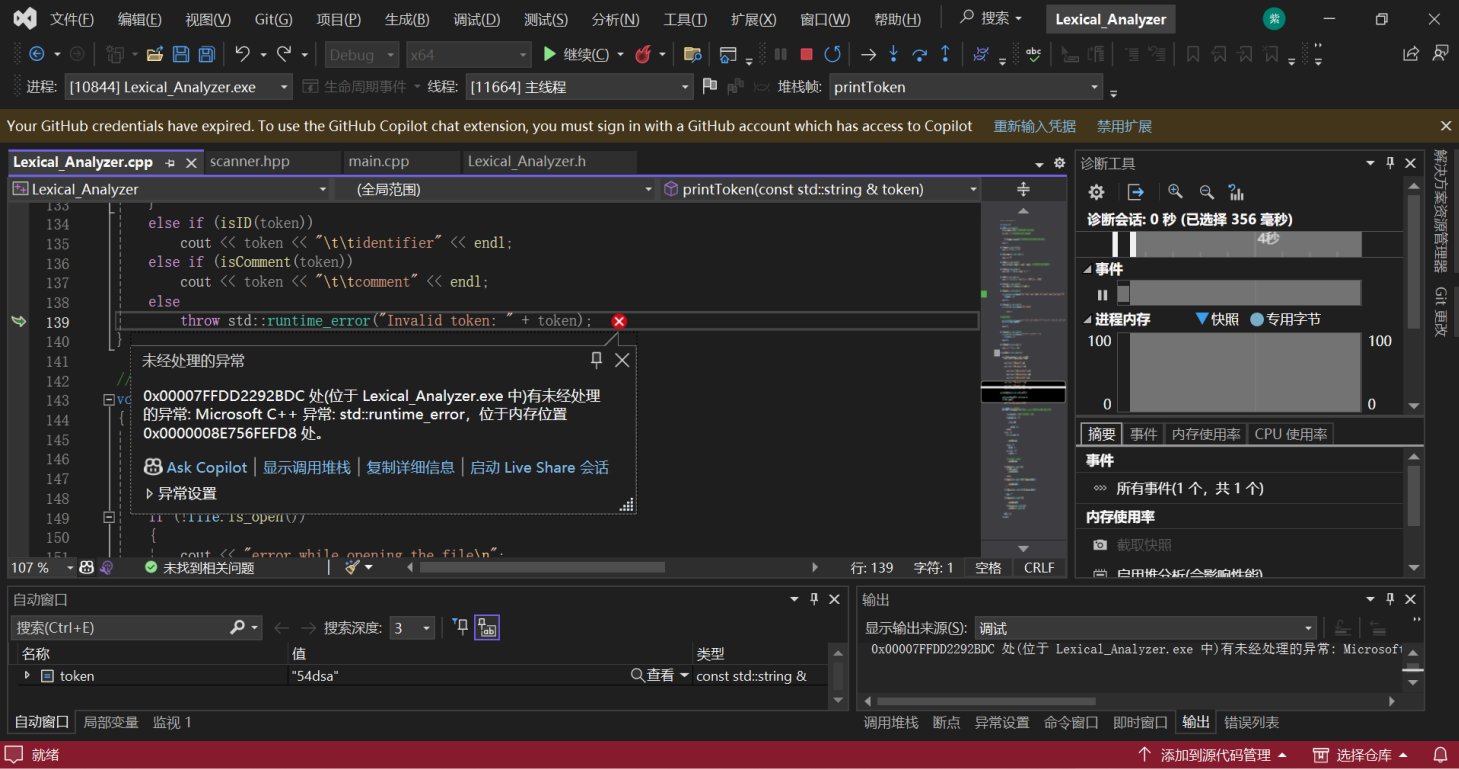
```
#define pretreatment
GOOD identifier
2000 literal:digit
// comment
int keyword
y identifier
= operator
/ operator
t identifier
; separator
int keyword
main identifier
( separator
) separator
{ separator
int keyword
x identifier
; separator
bool identifier
change identifier
= operator
FALSE literal:bool
; separator
x identifier
= operator
x identifier
+ operator
5 literal:digit
; separator
} separator
/* comment
// comment
for statement
( separator
int keyword
i identifier
= operator
0 literal:digit
, separator
i identifier
```

```
= operator
0 literal:digit
, separator
i identifier
<= operator
4 literal:digit
; separator
i identifier
++ operator
) separator
{ separator
x identifier
+= operator
i identifier
; separator
} separator
x identifier
++ operator
; separator
string identifier
str identifier
= operator
"jzx" literal:string
for statement
( separator
const keyword
auto keyword
& operator
op identifier
; separator
opers identifier
) separator
if keyword
( separator
```

```
op      identifier
==     operator
str    identifier
)      separator
return keyword
true   literal:bool
;      separator
```

2.3 报错提示

输入文件的最后两行是错误词法，我在程序中加入了捕捉非法输入并throw的机制，如下：



报错：invalid token

三、自定义的REs和DFA

(一) 自定义的词法单元及其REs

类型	词法单元	RE
literal:digit 数字	digit	digit(digit)*
comment 注释	//	/(character)*(\n)
	/* */	(/*)(character)*(/)
Pretreatment 预处理语句	#	#(character)*(\n)
Bool 布尔值	true	true
	false	false
	TRUE	TRUE
	FALSE	FALSE
String 字符串	" "	"(character)*"

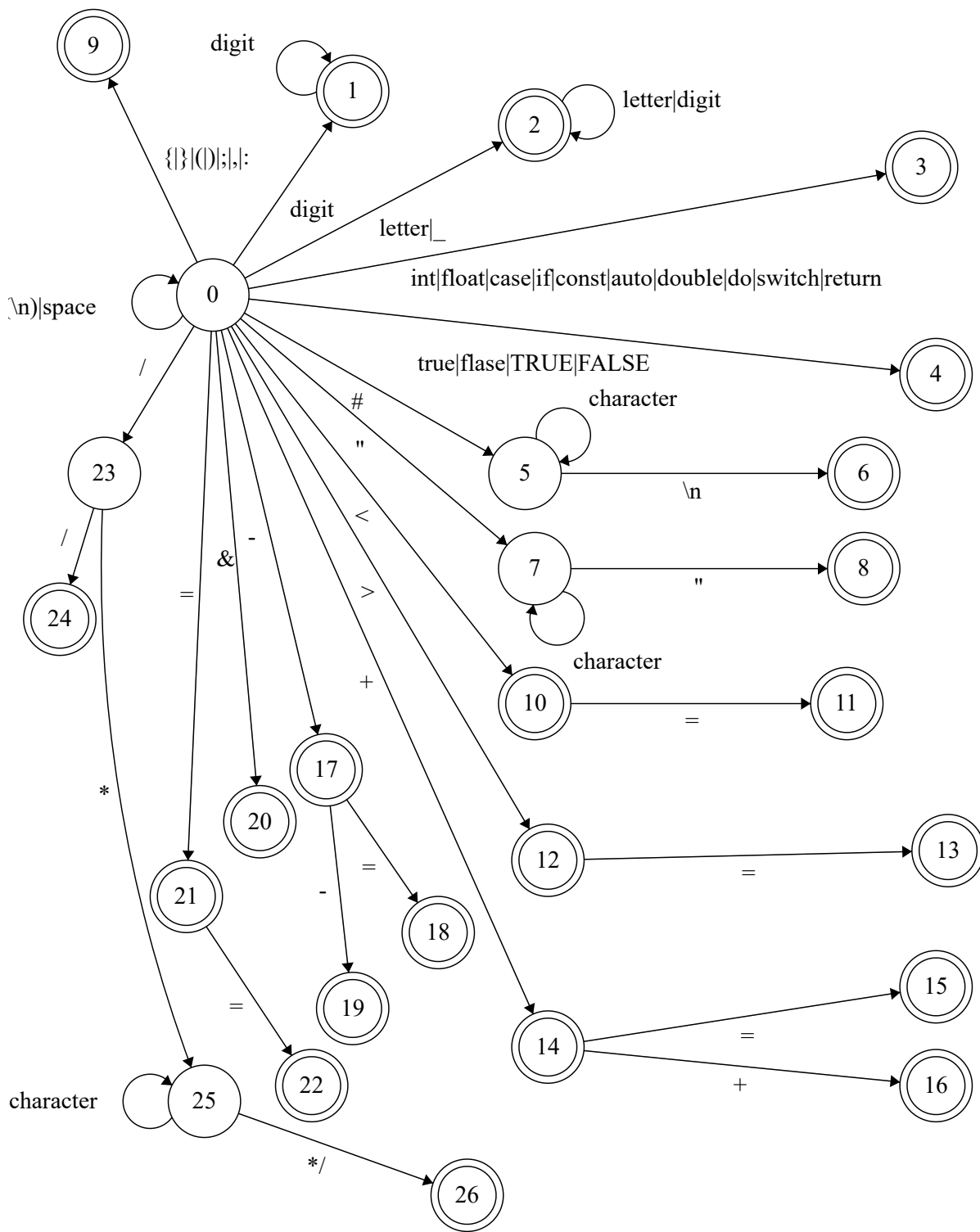
keyword 关键字	int	int
	float	float
	auto	auto
	double	double
	do	do
	switch	switch
	return	return
	case	case
	if	if
	const	const
Statement 语句	for	for
	while	while
Operator 操作符	>	>
	<	<
	>=	>=
	<=	<=
	*	*
	&	&
	+	+
	-	-
	/	/
	=	=
	-=	-=
	*=	*=
	/=	/=
	+=	+=
	--	--
	++	++
	==	==
Separator 分隔符	{	{
	}	}
	,	,
	((

))
	;	;
	:	:
identifier 标识符	identifier	(letter _)(letter digit)*

RE中的character表示任意字符

(二) 合并化简后的DFA

下述DFA是由上面RE表格中RE转化为NFA，将各NFA合并并转化为DFA后的结果：



各accept state标识如下:

- | 0为初态
- | 1为digit 数字
- | 2为identifier 标识符
- | 3为keyword 关键字

4为bool 布尔值

6为pretreatment 预处理语句

8为string 字符串

9为separator 分隔符

10-22均为operator 操作符，但这里对操作符进行了细分，具体可见词法类型表

24、25均为comment 注释，这里24是单行注释，25是多行注释

四、代码实现

（一）重要变量

在词法分析函数中，主要有两个变量

```
char ch;//当前读入字符
std::string buffer;//缓冲区
```

其中，**缓冲区buffer是一个重要变量**，用于暂时存储正在构建的词法单元。当读取到一个字符时，将其添加到缓冲区中。如果后续的字符也属于同一个词法单元的一部分，它们将被追加到缓冲区中，以构建完整的词法单元。

- 当新读入的字符为非法字符（空格或换行）时，表明现在缓冲区中是一个完整的词法单元，打印输出并清空缓冲区
- 针对操作符operator和标识符多个状态间的转换，不会在缓冲区中的字符构成一个完整的操作符词法单元时就打印，而是当新读入的字符不是操作符类型时才打印并清空缓冲区。比如 `x+=5` 语句，当缓冲区中是字符"+"是不会打印"+"，而是缓冲区中读入了"+="，新读入的5不是操作符了，才会触发缓冲区清空并打印词法单元。
- 如果当前字符是分隔符，那么需要将缓冲区中的词法单元打印出来，并清空缓冲区。然后将当前分隔符字符作为一个独立的词法单元打印出来。这样可以确保分隔符前后的词法单元被正确打印。

（二）各种词法单元判别函数

根据各种词法单元的判别函数，在迭代过程中可直接调用

如下：


```

// 判断是否为标识符
bool isID(const std::string& str)
{
    if (std::isdigit(str[0])) // 如果首字符是数字，则不是标识符
        return false;
    int counter = 0;
    if (str[0] == '_') // 如果首字符是下划线，则计数器加1
        counter++;

    for (; counter < str.size(); counter++)
        if (!isalnum(str[counter])) // 如果字符既不是字母也不是数字，则不是标识符
            return false; // isalnum检测字符串是否由字母和数字组成

    return true;
}

// 判断是否为注释
bool isComment(const std::string& str)
{
    return str == "/*" || str == "//";
}

// 判断是否为预处理语句
bool isPretreatment(const std::string& str)
{
    return str == "#";
}

// 判断是否为数字
bool isDigit(const std::string& str)
{
    return std::all_of(str.begin(), str.end(), ::isdigit); // 判断字符串中的字符是否都是数字
}

// 判断是否为字符串
bool isString(const std::string& str)
{
    return str[0] == '"' && str[str.size() - 1] == '"';
}

// 判断是否为布尔值
bool isBool(const std::string& str)
{
    return str == "true" || str == "false" || str == "TRUE" || str == "FALSE";
}

// 判断是否为字面量（数字、字符串或布尔值）
bool isLiteral(const std::string& str)
{
    return isDigit(str) || isString(str) || isBool(str);
}

// 判断是否为关键字
bool isKeyword(const std::string& str)
{
    const vector<std::string> keywords{ "int", "float", "auto", "double", "do", "switch", "return", "case", "const", "if" };
    for (const auto& keyword : keywords)
        if (keyword == str)
            return true;
}

```

```

        return false;
    }

// 判断是否为语句
bool isStatement(const std::string& str)
{
    const vector<std::string> statements{ "for", "while" };
    for (const auto& statement : statements)
        if (statement == str)
            return true;

    return false;
}

// 判断是否为操作符
bool isOperator(const std::string& str)
{
    const vector<std::string> operators{ "<", ">", "<=", ">=", "*", "&", "+", "-", "/", "=", "-=", "*=", "+=", "/=", "++", "--", "==" };
    for (const auto& op : operators)
        if (op == str)
            return true;

    return false;
}

// 判断是否为分隔符
bool isSeparator(const std::string& str)
{
    const vector<std::string> Separators{ "{", "}", ",", "(", ")", ";", ":" };
    for (const auto& separate : Separators)
        if (separate == str)
            return true;

    return false;
}

```

(三) 循环迭代过程

以下是根据DFA写出的循环迭代过程：

使用while循环逐字符读取文件内容，同时禁用跳过空格字符的功能(std::noskipws)，以便保留空格字符。对于读入的每一个字符

1. **首先检查是否处于多行注释或单行注释状态。**如果遇到"/"字符，表示可能出现注释。若是，则根据注释类型处理字符，并继续下一个字符的读取，**单行注释终止于换行，多行注释终止于"*/"**。
2. 如果当前字符不是空格字符且不是注释，则检查它是否非法字符（空格或换行）。若是非法字符且缓冲区buffer不为空，则打印缓冲区中的词法单元，并清空缓冲区。
3. 如果当前字符是操作符且缓冲区不是操作符，**表明缓冲区中已经是一个完整的token**，则打印缓冲区中的词法单元及其类型（如果不为空），并清空缓冲区。
4. 如果当前字符不是操作符且缓冲区是操作符，**表明缓冲区中已经是一个完整的操作符token**，则打印缓冲区中的词法单元及其类型，并清空缓冲区。
5. 如果当前字符是分隔符，则打印缓冲区中的词法单元及其类型（如果不为空），清空缓冲区，然后打印当前字符作为一个独立的词法单元。
6. 将当前字符添加到缓冲区中。

五、遇到的问题

1. 起初不知道如何把握状态转换的时机，譬如如何将"!="不单独读为"!="和"=", 后来引入了缓冲区的概念，使用了预先读写的思想，到了 accept state之后再输出词法单元及其类型
2. 关键词和标识符会混淆，只需要先判断关键词，不是关键词再去判断是不是标识符即可。
3. 程序有些繁琐，需要耐心debug

六、实验心得

词法分析是编译原理中的重要环节，它将源代码分解为一个个词法单元，为后续的语法分析和语义分析奠定了基础。这个实验让我更深入地理解了词法分析的原理和实现方法。

通过自己从RE构造NFA，再合并NFA、化简为DFA，更清晰的学习了DFA的构造过程，对理论学习很有帮助。

词法分析的实现是一个很繁琐的过程，在编写词法分析函数时，需要充分考虑各种可能的情况，比如注释、非法字符、操作符和分隔符等。对每一种情况都需要有相应的处理逻辑，以确保词法分析的准确性和完整性。在编写程序过程中需要耐心debug。

在实现词法分析函数时，代码的可读性和可维护性是非常重要的。使用清晰的变量命名、适当的注释和模块化的代码结构可以提高代码的可理解性和可扩展性。