词法分析程序的设计与实现

杨晨
学号 2021212171
北京邮电大学计算机学院

日期：2023 年 10 月 1 日

目录

1 概述

1.1 实验内容

1. 选定源语言，c 语言
2. 可以识别出用源语言编写的源程序中的每个单词符号，并以记号的形式输出每个单词符号。
3. 可以识别并跳过源程序中的注释。
4. 可以统计源程序中的语句行数、各类单词的个数、以及字符总数，并输出统计结果。
5. 检查源程序中存在的词法错误，并报告错误所在的位置。
6. 对源程序中出现的错误进行适当的恢复，使词法分析可以继续进行，对源程序进行一次扫描，即可检查并报告源程序中存在的所有词法错误。

1.2 开发环境

- Windows10
- Visual Studio Code
- Win flex-bison

2 程序的功能模块划分

2.1 常见的标记模式和正则表达式

```
O   [0-7]
D   [0-9]
NZ  [1-9]
L   [a-zA-Z_]
A   [a-zA-Z_0-9]
H   [a-fA-F0-9]
HP  (0[xX])
E   ([Ee][+-]?{D}+)
P   ([Pp][+-]?{D}+)
FS  (f|F|l|L)
IS  (((u|U)(l|L|ll|LL)?)|((l|L|ll|LL)(u|U)?))
```

```
CP   (u|U|L)
SP   (u8|u|U|L)
ES   (\\(['"\?\\abfnrtv]|[0-7]{1,3}|x[a-fA-F0-9]+))
NE   [^"\\\n]
WS   [ \t\v\f]
```

这段代码用于定义一些常见的标记模式和正则表达式。下面是对每个定义的模式的详细解释:

- O: 匹配八进制数字的字符集，范围从 0 到 7。
- D: 匹配十进制数字的字符集，范围从 0 到 9。
- NZ: 匹配非零的十进制数字的字符集，范围从 1 到 9。
- L: 匹配字母、下划线的字符集，包括小写字母、大写字母和下划线。
- A: 匹配字母、数字、下划线的字符集，包括小写字母、大写字母、数字和下划线。
- H: 匹配十六进制数字的字符集，包括小写字母、大写字母和数字。
- HP: 匹配十六进制数的前缀，即以”0x” 或”0X” 开头。
- E: 匹配浮点数中的指数部分，包括可选的正负号，后面跟着十进制数字。
- P: 匹配浮点数中的十六进制指数部分，包括可选的正负号，后面跟着十六进制数字。
- FS: 匹配浮点数类型后缀，可以是”f”、”F”、”l” 或”L”。
- IS: 匹配整数类型后缀，可以是各种组合，如”u”、”U”、”l”、”L”、”ll”、”LL” 等。
- CP: 匹配字符类型后缀，可以是”u”、”U” 或”L”。
- SP: 匹配字符串类型前缀，可以是”u8”、”u”、”U” 或”L”。
- ES: 匹配转义序列，包括转义字符（如”\n”、”\t” 等）、八进制字符（如”\377”）和十六进制字符（如”\xFF”）。
- NE: 匹配非转义字符和换行符的字符集。
- WS: 匹配空白字符，包括空格、制表符、垂直制表符和换页符。

2.2 声明部分

```
%{
#include <stdio.h>

/* C token types */
#define KEYWORD 1
#define IDENTIFIER 2
#define I_CONSTANT 3 // integer constant
#define F_CONSTANT 4 // floating constant
#define C_CONSTANT 5 // character constant
#define STRING_LITERAL 6
#define PUNCTUATOR 7
```

```
/* statistics */
int charCount = 0;
int lineCount = 1;
int columnCount = 1;
int keywordCount = 0;
int identifierCount = 0;
int integerConst = 0;
int floatingConst = 0;
int charConst = 0;
int stringCount = 0;
int punctuatorCount = 0;
int errorCount = 0;

/* handle errors */
void yyerror(const char *);

/* handle multi-line comments */
static void isComment(void);
%}
```

这段代码是使用 Lex 工具编写的 C 语言词法分析器，用于识别和分类 C 语言源代码中的不同词法单元，如关键字、标识符、常量、字符串、标点符号等。让我们对其进行详细解释：

`#include <stdio.h>`这是 C 语言中的预处理指令，用于包含标准输入输出库的头文件`#include <stdio.h>`，以便在词法分析器中使用`printf`和其他相关函数。

定义 C 语言中的不同令牌类型：这一系列`#define`语句定义了常量，用于标识 C 语言中的不同 token 类型，如关键字、标识符、常量、字符串和标点符号。

统计信息变量：这些变量用于统计分析结果，如字符数、行数、列数、关键字数、标识符数、常量数、字符串数、标点符号数和错误数。

`void yyerror(const char *)`这是一个函数原型，用于输出错误和详细信息。

`static void isComment(void)`这是一个静态函数，用于处理注释。

上述代码片段中的内容是 Lex 工具的规则部分之前的全局声明和定义部分。在 Lex 工具中，通过在规则部分使用正则表达式来匹配输入的文本，并在匹配到特定模式时执行相应的动作。

2.3 翻译规则部分

2.3.1 继续解析

```
"\n"    { lineCount++; columnCount=0; return -1; } /* wrap */
"/*"    { isComment(); return -1; }   /* multi-line comment begins */
```

```
"//".* { return -1; }  /* single-line comment */
"#".+  { return -1; }  /* preprocessor directive */
{WS}   {return -1;}    /* white space */
```

在 Lex 中，规则可以返回一个整数值，用于指示词法分析器采取的下一个操作。通常情况下，返回-1 表示继续解析，而返回其他值可能表示要进行一些特定的操作或者终止解析过程。上述代码的作用如下：

- 当遇到换行符（"\n"）时，会执行相应的动作。动作是递增 lineCount 变量（表示行数）并将 columnCount 变量（表示列数）重置为 0。
- 当遇到换行符（"\n"）时，会执行相应的动作。动作是递增 lineCount 变量（表示行数）并将 columnCount 变量（表示列数）重置为 0。
- 当遇到以"//"开头的注释时要执行的操作。它匹配从"//"开始的任意字符序列（.*表示匹配任意字符零次或多次）
- 当遇到以"#"开头的预处理指令时要执行的操作。它匹配以"#"开头的任意字符序列（.+表示匹配任意字符至少一次）
- 当遇到空字符时，跳过。

2.3.2 关键字

```
"auto"                  { return KEYWORD; } /* keywords */
"break"                 { return KEYWORD; }
"case"                  { return KEYWORD; }
...
"_Static_assert"        { return KEYWORD; }
"_Thread_local"         { return KEYWORD; }
"__func__"              { return KEYWORD; }
```

这段 Lex 代码用于匹配并返回 C 语言中的关键字。每行定义了一个关键字，例如”auto”、”break”、”case” 等等。当输入的源代码中出现这些关键字时，相应的规则会匹配，并返回一个标识符 (声明部分定义的 KEYWORD)

每个规则都以关键字作为匹配模式，并在匹配成功时返回 KEYWORD 标识符。这个标识符可以在 Lex 文件的其他规则中使用，用于进一步处理关键字。

2.3.3 标识符

```
{L}{A}*                 { return IDENTIFIER; }  /* identifiers */
```

这段 Lex 代码用于匹配并返回标识符（identifiers），标识符是由字母、下划线和数字组成的变量名或标识名称。

这行代码定义了一个规则，它以字母或下划线（L）作为开头，后面可以跟随零个或多个字母、下划线或数字（A）。这个规则匹配了标识符的模式，例如 myVariable、_count、x2 等等。

当输入的源代码中出现符合这个规则的标识符时，Lex 将匹配到该规则，并执行相应的动作，即返回一个标识符（声明部分定义的 IDENTIFIER）。这个标识符可以在 Lex 文件的其他规则中使用，用于进一步处理标识符。

2.3.4 整数常量

```
{HP}{H}+{IS}?                { return I_CONSTANT; }  /* constants */
{NZ}{D}*{IS}?                { return I_CONSTANT; }
"0"{O}*{IS}?                 { return I_CONSTANT; }
```

这段 Lex 代码用于匹配并返回 C 语言中的整数常量（integer constants）。它包含了三个规则：

- 匹配十六进制整数常量的模式。它以 HP（0x 或 0X）作为起始，后面跟随一个或多个十六进制数字（H），并可选择性地包含一个后缀（IS）来指示整数的类型。
- 匹配十六进制整数常量的模式。它以 HP（0x 或 0X）作为起始，后面跟随一个或多个十六进制数字（H），并可选择性地包含一个后缀（IS）来指示整数的类型。
- 匹配八进制整数常量的模式。它以字符 0 作为起始，后面可以跟随零个或多个八进制数字（O），并可选择性地包含一个后缀（IS）来指示整数的类型。

2.3.5 字符常量

```
{CP}?"'"({NE}|{ES})+"'"      { return C_CONSTANT; }
```

这段 Lex 代码用于匹配并返回 C 语言中的字符常量（character constants）。

这个规则匹配字符常量的模式。它由几个部分组成：

- CP?：这是一个可选部分，表示字符常量可以选择性地以字符类型前缀（u、U、L）开头。CP 匹配字符类型前缀（u、U、L）的模式。
- "'"：这是字符常量的起始引号。
- (NE|ES)+：这是字符常量的内容部分。它由一系列的非转义字符（NE）或转义序列（ES）组成。NE 匹配非转义字符的模式，而 ES 匹配转义序列的模式。(NE|ES)+ 表示可以有一个或多个非转义字符或转义序列。
- "'"：这是字符常量的结束引号。

当输入的源代码中出现符合这个规则的字符常量时，Lex 会匹配到该规则，并执行相应的动作，即返回一个标识符 C_CONSTANT，表示匹配到了一个字符常量。

这个规则允许字符常量的内容包含非转义字符或转义序列，并在匹配成功时返回 C_CONSTANT 标识符，以便在 Lex 文件的其他规则中对字符常量进行特殊处理或进一步分析。

2.3.6 浮点常量

```
{D}+{E}{FS}?                    { return F_CONSTANT; }
{D}*"."{D}+{E}?{FS}?            { return F_CONSTANT; }
```

```
{D}+"."{E}?{FS}?                    { return F_CONSTANT; }
{HP}{H}+{P}{FS}?                    { return F_CONSTANT; }
{HP}{H}*"."{H}+{P}{FS}?             { return F_CONSTANT; }
{HP}{H}+"."{P}{FS}?                 { return F_CONSTANT; }
```

这段 Lex 代码用于匹配并返回 C 语言中的浮点数常量（floating-point constants）。它包含了六个规则，如下：

- 以一个或多个数字（D）开头，接着是一个指数部分，由字母 E 表示，后面跟着一个可选的符号和一个或多个数字。最后，可能会包含一个可选的浮点数后缀（FS）。
- 以零个或多个数字（D）开头，接着是一个小数点，然后是一个或多个数字。可选地，可以包含一个指数部分，由字母 E 表示，后面跟着一个可选的符号和一个或多个数字。最后，可能会包含一个可选的浮点数后缀（FS）
- 以一个或多个数字（D）开头，接着是一个小数点。可选地，可以包含一个指数部分，由字母 E 表示，后面跟着一个可选的符号和一个或多个数字。最后，可能会包含一个可选的浮点数后缀（FS）。
- 以十六进制前缀（HP）开头，后面跟着一个或多个十六进制数字（H），然后是一个指数部分，由字母 P 表示。最后，可能会包含一个可选的浮点数后缀（FS）。
- 以十六进制前缀（HP）开头，后面跟着零个或多个十六进制数字（H），然后是一个小数点，接着是一个或多个十六进制数字。最后，需要包含一个指数部分，由字母 P 表示。最后，可能会包含一个可选的浮点数后缀（FS）。
- 以十六进制前缀（HP）开头，后面跟着一个或多个十六进制数字（H），然后是一个小数点，接着是一个指数部分，由字母 P 表示。最后，可能会包含一个可选的浮点数后缀（FS）。

这些规则按照特定的模式匹配浮点数常量，并在匹配成功时返回 F_CONSTANT 标识符，以便在 Lex 文件的其他规则中对浮点数常量进行特殊处理或进一步分析。

2.3.7 字符串字面量

```
({SP}?\"({NE}|{ES})*\"{WS}*)+   { return STRING_LITERAL; }
/* strings literal */
```

这个 Lex 代码片段是用于在词法分析器中识别字符串字面量（string literals）的模式。

这个模式可以被分解如下：

- SP?：匹配可选的空格前缀（unicode、u、U、L）。
- \”：匹配开始的双引号字符。
- (NE|ES)*：匹配零个或多个非转义字符（NE）或转义字符（ES）。
- \”：匹配结束的双引号字符。
- WS*：匹配零个或多个空格字符。
- 整个模式用括号括起来，并且带有加号（+），表示可以重复匹配多个字符串字面量。

当词法分析器遇到匹配这个模式的输入时，它会返回 STRING_LITERAL 标记，表示它识别到了一个字符串字面量。

2.3.8 标点符号

```
"..."                   { return PUNCTUATOR; }  /* punctuators */
">>="                   { return PUNCTUATOR; }
"<<="                   { return PUNCTUATOR; }
...
">"                     { return PUNCTUATOR; }
"^"                     { return PUNCTUATOR; }
"|"                     { return PUNCTUATOR; }
"?"                     { return PUNCTUATOR; }
```

这段 Lex 代码片段用于匹配和识别各种标点符号（punctuators）。

每行的模式都是一个特定的标点符号，例如 »=, +=, &&, (等等。当输入与任何一个模式匹配时，词法分析器将返回 PUNCTUATOR 标记，表示它识别到了一个标点符号。大体可分为如下几类：

- ...，它通常用于表示可变参数。
- 各种复合赋值运算符（如 »=, +=, -=, *=, /=, %= 等）、位移运算符（如 », «）、递增递减运算符（如 ++, –）、箭头运算符（->）、逻辑运算符（如 &&, ||）、比较运算符（如 <=, >=, ==, !=）和分号（;）。
- 各种单个字符的标点符号，例如大括号（, ）、逗号（,）、冒号（:）、等号（=）、圆括号（(,)）、方括号（[,]）、点号（.）、位与（&）、逻辑非（!）、位非（~）、减号（-）、加号（+）、乘号（*）、除号（/）、取模（%）、小于号（<）、大于号（>）、异或号（^）、位或（|）和问号（?）。

总之，这段代码定义了一系列模式，用于匹配和识别各种标点符号我注意到这些模式中的返回值是 PUNCTUATOR，这表示当匹配成功时，词法分析器将返回一个 PUNCTUATOR 标记，表示它识别到了一个标点符号。

2.3.9 错误

```
/* errors */
{D}+({A}|\.)+    { yyerror("illegal name"); return -1; }
{SP}?\"({NE}|{ES})*{WS}*  { yyerror("unclosed string"); return -1; }
.                { yyerror("unexpected character"); return -1; }
```

这部分定义了三个规则：

- D+(A|)+：匹配非法的名称。它匹配一个或多个数字（D+），后跟一个字母或点号（A|.)+）。当匹配到这种模式时，词法分析器将输出一个错误消息 “illegal name”。
- SP?(NE|ES)*WS*：匹配未关闭的字符串常量。它匹配一个可选的前缀（SP?），后跟一个双引号，然后匹配零个或多个非转义字符（NE）或转义字符（ES），再匹配零个或多个空格字符，由于缺少结束双引号”，词法分析器将输出一个错误信息”unclosed string”

- 匹配除换行符外的任意字符。当词法分析器遇到任何无法匹配其他规则的字符时，它会匹配这个规则。当匹配到这种模式时，词法分析器将返回一个错误消息”unexpected character”，表示遇到了意外的字符。

3 使用说明

3.1 LEX 编译程序

利用 win_flex 编译写好的.l 文件，生成对应的.c 代码

```
> D:\win_flex_bison-latest\win_flex.exe --wincompat --outfile=D:\
   win_flex_bison-latest\code.yy.c D:\win_flex_bison-latest\code.l
```

3.2 C 语言编译程序

其中

```
yyin = fopen("D:\\program\\work\\code\\test\\test1.c", "r");
```

这里可以指定待分析的源代码路径

4 测试

4.1 测试集 1

该测试集用于测试程序在没有词法错误时，是否能正常运行。

4.1.1 输入的源代码

```
// Line comment *///*/
#include <stdio.h>

int main()
{
    /*
     * Block comment *
     */
    char *msg = "Hello ";
    char ch = 'w';
    float f = 0.145e+3;
    double d = 3.e3;
    float f2 = 0.145e-3;
```

```
    double d2 = 3.;
    int integer = 864;
    long long int longint = 1234567890123456789;
    printf("%s %f\n", msg, d);
    printf("%c\t%d\n", ch, integer);
    printf("%lld\v", longint);
    return 0;
}
```

4.1.2 输出结果

```
4:1: [Keyword]: int
4:5: [Identifier]: main
4:9: [Punctuator]: (
4:10: [Punctuator]: )
5:1: [Punctuator]: {
9:5: [Keyword]: char
9:10: [Punctuator]: *
9:11: [Identifier]: msg
9:15: [Punctuator]: =
9:17: [String]: "Hello "
9:25: [Punctuator]: ;
10:5: [Keyword]: char
10:10: [Identifier]: ch
10:13: [Punctuator]: =
10:15: [Character]: 'w'
10:18: [Punctuator]: ;
11:5: [Keyword]: float
11:11: [Identifier]: f
11:13: [Punctuator]: =
11:15: [Floating]: 0.145e+3
11:23: [Punctuator]: ;
12:5: [Keyword]: double
12:12: [Identifier]: d
12:14: [Punctuator]: =
12:16: [Floating]: 3.e3
12:20: [Punctuator]: ;
13:5: [Keyword]: float
```

```
13:11: [Identifier]: f2
13:14: [Punctuator]: =
13:16: [Floating]: 0.145e-3
13:24: [Punctuator]: ;
14:5: [Keyword]: double
14:12: [Identifier]: d2
14:15: [Punctuator]: =
14:17: [Floating]: 3.
14:19: [Punctuator]: ;
15:5: [Keyword]: int
15:9: [Identifier]: integer
15:17: [Punctuator]: =
15:19: [Integer]: 864
15:22: [Punctuator]: ;
16:5: [Keyword]: long
16:10: [Keyword]: long
16:15: [Keyword]: int
16:19: [Identifier]: longint
16:27: [Punctuator]: =
16:29: [Integer]: 1234567890123456789
16:48: [Punctuator]: ;
17:5: [Identifier]: printf
17:11: [Punctuator]: (
17:12: [String]: "%s %f\n"
17:21: [Punctuator]: ,
17:23: [Identifier]: msg
17:26: [Punctuator]: ,
17:28: [Identifier]: d
17:29: [Punctuator]: )
17:30: [Punctuator]: ;
18:5: [Identifier]: printf
18:11: [Punctuator]: (
18:12: [String]: "%c\t%d\n"
18:22: [Punctuator]: ,
18:24: [Identifier]: ch
18:26: [Punctuator]: ,
18:28: [Identifier]: integer
18:35: [Punctuator]: )
```

```
18:36: [Punctuator]: ;
19:5: [Identifier]: printf
19:11: [Punctuator]: (
19:12: [String]: "%lld\v"
19:20: [Punctuator]: ,
19:22: [Identifier]: longint
19:29: [Punctuator]: )
19:30: [Punctuator]: ;
20:5: [Keyword]: return
20:12: [Integer]: 0
20:13: [Punctuator]: ;
21:1: [Punctuator]: }

Total characters:       415
Total lines:    21

Keyword:        12
Identifier:     17
Integers:       3
Floatings:      4
Characters:     1
Strings:        4
Punctuators:    36
Errors:         0
```

4.1.3 输出结果分析

代码总共包含了 415 个字符，分布在 21 行中。词法分析结果提供了每个词法单元的行号、列号和类型。

- 代码中包含了 12 个关键字。
- 代码中包含了 17 个标识符。
- 代码中包含了 7 个数值常量，包括浮点数 4 个和整数常量 3 个。
- 代码中包含了 1 个字符常量。
- 代码中包含了 4 个字符串字面值。
- 代码中包含了 36 个标点符号。

同时，词法分析结果未发现任何错误。

4.2 测试集 2

4.2.1 输入的源代码

```
int main()
{
    double 2ch = 1.a;
    double a = 1ee2;
    char *num = "unclose\++";
    char *str = "unclose\";
    s = ''
    w = \$
    int a = '@;
    . = 1.2.3;
}
/* unclose_Comment
```

4.2.2 输出结果

```
1:1: [Keyword]: int
1:5: [Identifier]: main
1:9: [Punctuator]: (
1:10: [Punctuator]: )
2:1: [Punctuator]: {
3:5: [Keyword]: double
3:12:[Error: illegal name]: 2ch
3:16: [Punctuator]: =
3:18:[Error: illegal name]: 1.a
3:21: [Punctuator]: ;
4:5: [Keyword]: double
4:12: [Identifier]: a
4:14: [Punctuator]: =
4:16:[Error: illegal name]: 1ee2
4:20: [Punctuator]: ;
5:5: [Keyword]: char
5:10: [Punctuator]: *
5:11: [Identifier]: num
5:15: [Punctuator]: =
5:17:[Error: unclosed string]: "unclose
```

```
5:25:[Error: unexpected character]: \
5:26: [Punctuator]: ++
5:28:[Error: unclosed string]: ";
6:5: [Keyword]: char
6:10: [Punctuator]: *
6:11: [Identifier]: str
6:15: [Punctuator]: =
6:17:[Error: unclosed string]: "unclose\";
7:5: [Identifier]: s
7:7: [Punctuator]: =
7:9:[Error: unexpected character]: '
7:10:[Error: unexpected character]: '
8:5: [Identifier]: w
8:7: [Punctuator]: =
8:9:[Error: unexpected character]: \
8:10:[Error: unexpected character]: $
9:5: [Keyword]: int
9:9: [Identifier]: a
9:11: [Punctuator]: =
9:13:[Error: unexpected character]: '
9:14:[Error: unexpected character]: @
9:15: [Punctuator]: ;
10:5: [Punctuator]: .
10:7: [Punctuator]: =
10:9:[Error: illegal name]: 1.2.3
10:14: [Punctuator]: ;
11:1: [Punctuator]: }
13:1:[Error: unterminated comment]:

Total characters:       188
Total lines:    13

Keyword:        6
Identifier:     7
Integers:       0
Floatings:      0
Characters:     0
Strings:        0
```

```
Punctuators:    20
Errors:         15
```

4.2.3 输出结果分析

输出结果指示了在给定的代码中存在多个错误:

- 第 3 行第 12 列的[Error: illegal name]:2ch表示在该位置上发现了一个错误，指出标识符"2ch"不是一个合法的名称。
- 第 3 行第 18 列的[Error: illegal name]:1.a表示在该位置上发现了一个错误，指出标识符"1.a"不是一个合法的名称。
- 第 4 行第 16 列的[Error: illegal name]:1e表示在该位置上发现了一个错误，指出标识符"1e"不是一个合法的名称。
- 第 5 行第 17 列的[Error: unclosed string]:"unclose表示在该位置上发现了一个错误，指出字符串字面量"unclose"未闭合。
- 第 5 行第 25 列的[Error: unexpected character]:\表示在该位置上发现了一个错误，指出意外字符\。
- 第 6 行第 17 列的[Error: unclosed string]:"unclose\"表示在该位置上发现了一个错误，指出字符串字面量"unclose"未闭合。
- 第 8 行第 9 列和第 10 列的[Error: unexpected character]:\$表示在该位置上发现了一个错误，指出意外的字符\$。
- 第 9 行第 13 列和 14 列的[Error: unexpected character]:'@ 表示在该位置上发现了一个错误，指出意外的字符 ‘@。
- 第 10 行第 9 列的[Error: illegal name]:1.2.3表示在该位置上发现了一个错误，指出标识符"1.2.3"不是一个合法的名称。
- 第 13 行第 1 列的[Error: unterminated comment]表示在该位置上发现了一个错误，指出这里的注释被破坏

5 实验总结

本次实验中我利用 LEX 工具编写了一个词法分析程序，使我对词法分析的流程更加清楚，对相关知识点的掌握更加牢固。

为了实现 C 语言的词法分析，我参考了 C11 的 ISO 标准，仔细阅读了标准中对各种词法元素的定义，包括关键字、标识符、常量、字符串字面量、运算符等。然后根据标准中给出的语法和课上所学的正则表达式，我设计了词法分析所需的匹配表达式。

在设计词法分析程序时，我预先设计了一系列常用的正则表达式，后续的词法分析，可以利用这些工具

在具体实现时，我遇到了一些困难。在第一次编写的程序中，出现了许多疏漏的情况，无法正确解析输入的 C 代码。为了解决这些问题，我多次细致地阅读语法定义，设计不同的样例进

行测试，反复调试程序。在这过程中，我找到并修复了程序中的许多 bug。

通过这个实验，我对词法分析的流程和实现有了更深入的理解，也提高了自己的编程能力，尤其是利用自动机识别字符串的能力。在阅读语法标准方面，我的英文文献阅读能力也得到了提高。总而言之，这个实验使我收获颇丰，对以后课程的学习非常有帮助。