lab1 实验报告

学号 201907040102 姓名 李平凡

实验要求

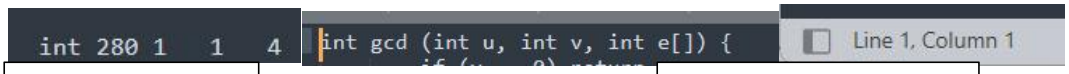
本次实验需要各位同学根据 cminux-f 的词法补全 lexical_analyer.l 文件，完成词法分析器，能够输出识别出的 token，type ,line(刚出现的行数)，pos_start(该行开始位置)，pos_end(结束的位置,不包含)。

实验难点

正则表达式书写，理解 flex 的使用过程，yylex（）的运行也就是它的输入和输出，正则表达式识别顺序，如何检测和测试代码实验正确性。

实验设计

1，首先需要清楚本实验需要识别的是哪些类型，根据 lexical_analyzer.h 文件，发现需要识别 5 类记号，但是某一类中包括的记号又有很多，比如 IDENTIFIER，我们还需要明白我们得识别出这些 token（符号）对应的 type 比如 int 类型的数字 286。pos_start 和 pos_end 可以理解为将代码放入 txt 文本中光标指向的位置.



TA_TOKE 结果　　光标起始位置

2，此外我们还需要知道一件事，根据自己写的 test.l 测试文件发现，yylex（）是一个词法分析例程，具体来讲就是会根据你的正则表达式来进行匹配，比如如果说你开始之前定义了一个关于 blank 的正则表达式，也就是“ ”，{}中加入了 return，那么你只使用一次 yylex（）在你的终端输入一个空格，他就会识别，而后面定义的正则表达式就算能匹配上相应的字符也不会有任何操作（因为 yylex 已经返回），而且如果你没有关于空格的正则表达式，当你的 yylex 只调用一次，你后面定义的正则表达式的假如能识别出来，但不会指向正确的位置，因为少算一个空格的位置（多个空格，认为是一个\t）。而且只会识别一个能匹配的字符。

```
" " {return BLANK;}
[a-zA-Z]+ { chars += strlen(yytext);words++;}

. {}
 /*对其他所有字符，不做处理，继续执行*/

%%

int main(int argc, char **argv){
    //yylex()是flex提供的词法分析例程，默认读取stdin
    line=1;
    int token;

    pos_end=1;
    pos_start=1;
    token = yylex();
    if(token==BLANK)
    {
        printf("this is blank");
    }
    printf("%d\n",token);
    printf("pos_start %d,pos_end %d\n",pos_start,pos_end);
    printf("look, I find %d words of %d chars\n", words, chars);
```

```
^[[A^[[Alpf@LAPTOP-P8EMT48L:/mnt/
 if
this is blank292
pos_start 1,pos_end 1
look, I find 0 words of 0 chars
```

所以我们就需要一个循环一直不停的去识别（这是提供代码中写死的，同时 index 就是识别出的每一个 token 的下标），同时也说明了为什么我们需要识别空格和 error，如果我们没有识别空格的能力，pos_start 和 pos_end 也不会指向正确的位置，同时 error 也是必要的，也是为了保证 pos_start 和 pos_end 指向正确的位置，同时识别出无法识别的（实验没有要求的）token。同时还有一点，yylex 是有返回值的，如果是一个无法识别的值（正则表达式未定义的值）它的返回值是 0，如果是定义的正则表达式，返回的值可以通过自己在{}中填写 return，否则返回值仍然是 0。

```
" " {}
[a-zA-Z]+ { chars += strlen(yytext);words++;}

. {}
 /*对其他所有字符，不做处理，继续执行*/

%%

int main(int argc, char **argv){
    //yylex()是flex提供的词法分析例程，默认读取stdin
    line=1;
    int token;

    pos_end=1;
    pos_start=1;
    token = yylex();
    if(token==BLANK)
    {
        printf("this is blank");
    }
    printf("%d\n",token);
    printf("pos_start %d,pos_end %d\n",pos_start,pos_end);
    printf("look, I find %d words of %d chars\n", words, chars);
    return 0;
}
```

```
 ifjo

0
pos_start 1,pos_end 1
look, I find 1 words of 4 chars
```

3，自己定义的正则表达式需要注意顺序问题，比如关键字和标识符，如果关键字是在标识符识别之后，关键字是不会被识别出来的。

```
[a-zA-Z]+ { chars += strlen(yytext);words++;}
if {pos_start=pos_end;pos_end += 2;return IF;}
. {}
 /*对其他所有字符，不做处理，继续执行*/

%%

int main(int argc, char **argv){
    //yylex()是flex提供的词法分析例程，默认读取stdin
    line=1;
    int token;

    pos_end=1;
    pos_start=1;
    yylex();
    if(token==BLANK)
    {
        printf("this is blank");
    }
    printf("%d\n",token);
    printf("pos_start %d,pos_end %d\n",pos_start,pos_end);
    printf("look, I find %d words of %d chars\n", words, chars);
    return 0;
}
```

这里会产生一个警告，其实是说明 if 关键字是不会被识别出来的

```
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall/src/lexer$ flex test.l
test.l:25: warning, rule cannot be matched
```

根据测试，标识符 ID 和 NUM，以及 ERROR 一定放在最后，ERROR 是.{}的内容。本实验关键在于把所有需要注意的 token 用正则表达式给表示出来。先看主循环

```
while(token = yylex()){
    switch(token){
        case COMMENT:{
         //STUDENT TO DO
            pos_start=pos_end;
            pos_end=pos_start+2;
            int i=2;
            while(yytext[i]!='*' || yytext[i+1]!='/')
            {
                if(yytext[i]=='\n')
                {
                    lines=lines+1;
                    pos_end=1;
                }
                else
                    pos_end=pos_end+1;
                i=i+1;
            }
            pos_end=pos_end+2;
            break;
           }

        case BLANK:{
            //STUDENT TO DO
            pos_start=pos_end;
            pos_end=pos_start+1;
            break;}
        case EOL:{
            //STUDENT TO DO
            lines+=1;
            pos_end=1;
            break;}
```

当识别到一个符合正则表达式定义的 token，那么先判断是否为注释，如果是注释，那么首先将光标拉平，也就是 pos_start 和 pos_end 位置一致，然后 pos_end 向后移动 2 个位置因为/*，然后通过一个 while 循环保证注释的内容全部遍历完全（判断条件就是看是否识别了*/因为实验要求注释里面不能嵌套注释），如果注释的内容中有换行，那么 line 加一，到了

新的一行 pos_end 为 1，循环结束后 pos_end 需要加 2 因为要留出*/的位置。
如果是 blank，pos_start 与 pos_end 拉平，pos_end+1
如果是 EOL 换行，line 加 1，pos_end 等于 1（因为到了新的一行）
EOL 和 blank 是不会输出到分析结果中，它们的存在是为了定位需要得到分析结果的 token，比如 if\n，我们需要识别 if 的准确位置，起始位置 1，终止位置 3，但不需要识别\n 的位置，\n 是为了下一行的准确定位。

```
\n {return EOL;}
\t {return BLANK;}
" " {return BLANK;}
\r {return BLANK;}
\s {return BLANK;}
else {pos_start=pos_end;pos_end=pos_start+4;return ELSE;}
if {pos_start=pos_end;pos_end=pos_start+2;return IF;}
int {pos_start=pos_end;pos_end=pos_start+3;return INT;}
return {pos_start=pos_end;pos_end=pos_start+6;return RETURN;}
void {pos_start=pos_end;pos_end=pos_start+4;return VOID;}
while {pos_start=pos_end;pos_end=pos_start+5;return WHILE;}
float {pos_start=pos_end;pos_end=pos_start+5;return FLOAT;}
"[]" {pos_start=pos_end;pos_end=pos_start+2;return ARRAY;}
\+ {pos_start=pos_end;pos_end=pos_start+1;return ADD;}
\- {pos_start=pos_end;pos_end=pos_start+1;return SUB;}
\* {pos_start=pos_end;pos_end=pos_start+1;return MUL;}
\/ {pos_start=pos_end;pos_end=pos_start+1;return DIV;}
\< {pos_start=pos_end;pos_end=pos_start+1;return LT;}
"<=" {pos_start=pos_end;pos_end=pos_start+2;return LTE;}
\> {pos_start=pos_end;pos_end=pos_start+1;return GT;}
">=" {pos_start=pos_end;pos_end=pos_start+2;return GTE;}
"==" {pos_start=pos_end;pos_end=pos_start+2;return EQ;}
"!=" {pos_start=pos_end;pos_end=pos_start+2;return NEQ;}
\= {pos_start=pos_end;pos_end=pos_start+1;return ASSIN;}
\; {pos_start=pos_end;pos_end=pos_start+1;return SEMICOLON;}
\, {pos_start=pos_end;pos_end=pos_start+1;return COMMA;}
\( {pos_start=pos_end;pos_end=pos_start+1;return LPARENTHESE;}
\) {pos_start=pos_end;pos_end=pos_start+1;return RPARENTHESE;}
\[ {pos_start=pos_end;pos_end=pos_start+1;return LBRACKET;}
\] {pos_start=pos_end;pos_end=pos_start+1;return RBRACKET;}
\{ {pos_start=pos_end;pos_end=pos_start+1;return LBRACE;}
\} {pos_start=pos_end;pos_end=pos_start+1;return RBRACE;}
[a-zA-Z]+ {pos_start=pos_end;pos_end=pos_start+strlen(yytext);return IDENTIFIER;}
[0-9]+ {pos_start=pos_end;pos_end=pos_start+strlen(yytext);return INTEGER;}
[0-9]*\.[0-9]+ {pos_start=pos_end;pos_end=pos_start+strlen(yytext);return FLOATPOINT;}
[0-9]+\. {pos_start=pos_end;pos_end=pos_start+strlen(yytext);return FLOATPOINT;}
\/\*\/*([^\/]*[^\/\*]\/[\/]*)*[^\/]*\*\/ {return COMMENT;}
. {pos_start=pos_end;pos_end=pos_start+strlen(yytext);return ERROR;}
```

识别的正则表达式如图所示，根据实验指导以及特殊关键字和符号可以得到正则表达式如上，同时，每一个被识别的正则表达式都需要的操作就是将光标移到正确的位置，比如 else if，如果 else 和 if 都是识别的 token，那么当识别出 else 后 pos_end 的光标指向了 e 的后面，pos_start=pos_end 的作用就是为了下一个识别的 if 的起始位置，因为 if 的起始位置就是 else 的终结位置加一个空格，然后 pos_end=识别出的 if 的长度+pos_start。同时返回关键字 if 的类型（一个大于 0 的整数）。这也就是为什么主循环 token=yylex()一直运行的结果，当然如果这个函数可以修改，可以在每个正则表达式{}中直接将类型起始位置终止位置等参数写入到结构体中，该函数可以只需要调用一次 yylex()。
后来发现在识别出正则表达式后进行的操作部分是相同的 pos_start = pos_end 以及 pos_end = 识别的 token 的长度+pos_start 所以最后更改正则表达式和函数如下

```
 //STUDENT TO DO
\n {return EOL;}
\t {return BLANK;}
" " {return BLANK;}
\r {return BLANK;}
\s {return BLANK;}
else {return ELSE;}
if {return IF;}
int {return INT;}
return {return RETURN;}
void {return VOID;}
while {return WHILE;}
float {return FLOAT;}
"[]" {return ARRAY;}
\+ {return ADD;}
\- {return SUB;}
\* {return MUL;}
\/ {return DIV;}
\< {return LT;}
"<=" {return LTE;}
\> {return GT;}
">=" {return GTE;}
"==" {return EQ;}
"!=" {return NEQ;}
\= {return ASSIN;}
\; {return SEMICOLON;}
\, {return COMMA;}
\( {return LPARENTHESE;}
\) {return RPARENTHESE;}
\[ {return LBRACKET;}
\] {return RBRACKET;}
\{ {return LBRACE;}
\} {return RBRACE;}
[a-zA-Z]+ {return IDENTIFIER;}
[0-9]+ {return INTEGER;}
[0-9]*\.[0-9]+ {return FLOATPOINT;}
[0-9]+\. {return FLOATPOINT;}
\/\*\/*([^\/]*[^\/\*]\/[\/]*)*[^\/]*\*\/ {return COMMENT;}
. {pos_start=pos_end;pos_end=pos_start+strlen(yytext);return ERROR;}
```

```
default :
    if (token == ERROR){
        sprintf(token_stream[index].text, "[ERR]: unable to analysize %s at %d line, from %d t
    } else {
        pos_start=pos_end;
        pos_end=pos_start+strlen(yytext);
        strcpy(token_stream[index].text, yytext);
    }
    token_stream[index].token = token;
    token_stream[index].lines = lines;
    token_stream[index].pos_start = pos_start;
    token_stream[index].pos_end = pos_end;
    index++;
    if (index >= MAX_NUM_TOKEN_NODE){
        printf("%s has too many tokens (> %d)", input_file, MAX_NUM_TOKEN_NODE);
        exit(1);
    }
```

实验结果验证

```
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall$ diff ./tests/lab1/token ./tests/lab1/TA_token
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall$
```

验证实验结果是正确的。

请提供部分自己的测试样例

自己测试案例：

```
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall/src/lexer$ flex test.l
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall/src/lexer$ gcc lex.yy.c -lfl
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall/src/lexer$ ./a.out
if
if 279 1 1 3
oo
oo 285 2 1 3
a[1] a[]
a 285 3 1 2
[ 274 3 2 3
1 286 3 3 4
] 275 3 4 5
a 285 3 6 7
[] 288 3 7 9
```

遇见换行符能够 lines 加一，遇到 a[]和 a[1]能够正确识别。

```
ifnooo if
ifnooo 285 4 1 7
if 279 4 8 10
```

能够识别出关键字和标识符

实验反馈

本次实验了解到正则表达式的运用，以及对 flex 和 linux 的使用，以及对.l 文件的基本编写，实验中难以处理的就是刚开始对代码的理解，不懂 yylex 的输入和输出，同时编写正则表达式的时候总是会出现换行和空格的错误，以及识别的错误，通过对实验指导书中的 test 文件指导，自己编写出了能够从终端输入立即得到反馈的 test.l 文件，帮助自己更好的编写正则表达式和验证编写的准确性，也非常方便的帮助自己测试正则表达式识别的顺序，减少了不必要的麻烦。并且通过自己的发现，我觉得这个 token 的识别应该是最长匹配。当我删除 identifier 的识别正则表达式，出现了如下情况。

```
elseif
else 278 1 1 5
if 279 1 5 7
```

但是当我加上之后

```
elseif
elseif 285 1 1 7
else if
else 278 2 1 5
if 279 2 6 8
```

说明，当有 else 和 if 以及 elseif 的正则表达式的定义，他会先选择 elseif 的正则表达式定义，而不是先识别成 else 和 if。所以识别的过程首先应该是最长匹配，然后根据正则表达式的顺序进行逐条匹配。

另外 yylex 应该隐含的把空格当作分隔符

```
if {printf("2");}
int {printf("1");}
```

```
ifint int if
21 1 2
```

如果加上一些其他的能够识别 ifint 的正则表达式

```
if {printf("2");}
int {printf("1");}
[a-zA-Z]+ { chars += strlen(yytext);words++;}
```

```
ifint if int
 2 1
look, I find 1 words of 5 chars
```

说明 yylex 应该把空格当作分隔符分割成了 ifint，int，if 然后对这些东西进行了最长匹配。（以上属于自己的实验猜测）