lab1 实验报告

学号：202008010415 姓名：温先武

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)。如：

文本输入：

```
int a;
```

则识别结果应为：

```
int     280     1       2       5
a       285     1       6       7
;       270     1       7       8
```

具体的需识别 token 参考 lexical_analyzer.h

特别说明对于部分 token，我们只需要进行过滤，即只需被识别，但是不应该被输出到分析结果中。因为这些 token 对程序运行不起到任何作用。

> 注意，你所需修改的文件应仅有[lexical_analyer.l]../../src/lexer/lexical_analyzer.l)。关于`FLEX`用法上文已经进行简短的介绍，更高阶的用法请参考百度、谷歌和官方说明。

1.1 目录结构

整个 repo 的结构如下

```
.
├── CMakeLists.txt
├── Documentations
│   └── lab1
│       └── README.md <- lab1 实验文档说明
├── README.md
├── Reports
│   └── lab1
│       └── report.md <- lab1 所需提交的实验报告（你需要在此提交实验报告）
├── include <-  实验所需的头文件
│   └── lexical_analyzer.h
├── src <-  源代码
│   └── lexer
│       ├── CMakeLists.txt
│       └── lexical_analyzer.l    <- flex 文件，lab1 所需完善的文件
└── tests <-  测试文件
    └── lab1
        ├── CMakeLists.txt
        ├── main.c    <- lab1 的 main 文件
        ├── test_lexer.py
        ├── testcase <-  助教提供的测试样例
        └── TA_token <-  助教提供的关于测试样例的词法分析结果
```

1.2 编译、运行和验证

lab1 的代码大部分由 C 和 python 构成，使用 cmake 进行编译。

编译

```
#  进入 workspace
$ cd cminus_compiler-2022-fall

#  创建 build 文件夹，配置编译环境
$ mkdir build
$ cd build
$ cmake ../

#  开始编译
#  如果你只需要编译 lab 1，请使用  make lexer
$ make
```

编译成功将在${WORKSPACE}/build/下生成 lexer 命令

运行

```
$ cd cminus_compiler-2022-fall
#  运行 lexer 命令
$ ./build/lexer
usage: lexer input_file output_file
#  我们可以简单运行下  lexer 命令，但是由于此时未完成实验，当然输出错误结果
$ ./build/lexer ./tests/lab1/testcase/1.cminus out
[START]: Read from: ./tests/lab1/testcase/1.cminus
[ERR]: unable to analysize i at 1 line, from 1 to 1
......
......

$ head -n 5 out
[ERR]: unable to analysize i at 1 line, from 1 to 1       258       1         1         1
[ERR]: unable to analysize n at 1 line, from 1 to 1       258       1         1         1
[ERR]: unable to analysize t at 1 line, from 1 to 1       258       1         1         1
[ERR]: unable to analysize     at 1 line, from 1 to 1       258       1         1         1
[ERR]: unable to analysize g at 1 line, from 1 to 1       258       1         1         1
```

我们提供了./tests/lab1/test_lexer.py python 脚本用于调用 lexer 批量完成分析任务。

```
# test_lexer.py 脚本将自动分析./tests/lab1/testcase 下所有文件后缀为.cminus 的文件，并将输出结果保存在./tests/lab1/token 文件下下
$ python3 ./tests/lab1/test_lexer.py
…
…
…
#上诉指令将在./tests/lab1/token 文件夹下产生对应的分析结果
$ ls ./tests/lab1/token
1.tokens 2.tokens 3.tokens 4.tokens 5.tokens 6.tokens
```

验证

我们使用 diff 指令进行验证。将自己的生成结果和助教提供的 TA_token 进行比较。

```
$ diff ./tests/lab1/token ./tests/lab1/TA_token
#  如果结果完全正确，则没有任何输出结果
#  如果有不一致，则会汇报具体哪个文件哪部分不一致
```

请注意助教提供的 testcase 并不能涵盖全部的测试情况，完成此部分仅能拿到基础分，请自行设计自己的 testcase 进行测试。

1.3 提交要求和评分标准

提交要求

本实验的提交要求分为两部分：实验部分的文件和报告，git 提交的规范性。

实验部分:

需要完善./src/lab1/lexical_analyer.l 文件;

需要在./Report/lab1/report.md 撰写实验报告。

实验报告内容包括:

实验要求、实验难点、实验设计、实验结果验证、实验反馈(具体参考 report.md;

实验报告推荐提交 PDF 格式。

git 提交规范：

不破坏目录结构(report.md 所需的图片请放在 Reports/lab1/figs/下);

不上传临时文件(凡是自动生成的文件和临时文件请不要上传，包括 lex.yy.c 文件以及各位自己生成的 tokens 文件);

git log 言之有物(不强制，请不要 git commit -m 'commit 1', git commit -m 'sdfsdf'，每次 commit 请提交有用的 comment 信息)

实验难点

（1）主要是对于一个特定的字符串，要写出它对应的正则表达式，我们首先要注意到“\”表示转义字符的意思，比如“\(”就表示左括号这一字符串，“\n”表示换行符，而“(”才表示左括号。

（2）主要是对于注释，注释的正则表达式很难懂，其实我们只要结合 C++的注释来理解就好了（cminus 中没有//这个注释）：C++的多行注释是/**/，/*匹配第一个它所遇到的*/，不管中间遇到什么字符（除了*/），因此我们特别要注意的一点就是/**/注释中间的内容还是可以出现/*的，但是不能出现*/（因为这相当于识别到结束符了）。因此，注释的正则表达式就是：\/*(?:[^*]|*+[^\/*])**+\/。前面的\/*表示“/*”，是注释的开始，然后中间是(?:[^*]|*+[^\/*])*，表示(?:[^*]|*+[^\/*])*可以有零个或多个，这很明显就是注释的内容，注释的内容零个或多个，就是注释可以写很多，也可以什么都不写，最后是*+\/，是“*+/”，很明显就是注释的结束，表示的是*必须有一个或多个，然后最后一个*和/匹配表示注释的结束。

（3）需要注意的是，C-Minus-f 的注释只能是/**/，而不能是//。

实验设计

首先创建一个 build 文件夹，然后配置编译环境：

```
root@click:/home/click/桌面/cminus_compiler-2022-fall-master/build# cmake ../
-- The C compiler identification is GNU 9.4.0
-- The CXX compiler identification is GNU 9.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found FLEX: /usr/bin/flex (found version "2.6.4")
-- Found BISON: /usr/bin/bison (found version "3.5.1")
-- Found LLVM 10.0.0
-- Using LLVMConfig.cmake in: /usr/lib/llvm-10/cmake
-- Configuring done
-- Generating done
-- Build files have been written to: /home/click/桌面/cminus_compiler-2022-fall-master/build
root@click:/home/click/桌面/cminus_compiler-2022-fall-master/build#
```

```
root@click:/home/click/桌面/cminus_compiler-2022-fall-master/build# make lexer
[ 20%] [FLEX][lex] Building scanner with flex 2.6.4
Scanning dependencies of target flex
[ 40%] Building C object src/lexer/CMakeFiles/flex.dir/lex.yy.c.o
lexical_analyzer.l: In function ‘analyzer’:
lexical_analyzer.l:52:11: warning: suggest parentheses around assignment used as truth value [-Wparentheses]
At top level:
/home/click/桌面/cminus_compiler-2022-fall-master/build/src/lexer/lex.yy.c:1130:16: warning: ‘input’ defined but n
ot used [-Wunused-function]
 1130 |     static int input  (void)
      |                ^~~~~
/home/click/桌面/cminus_compiler-2022-fall-master/build/src/lexer/lex.yy.c:1087:17: warning: ‘yyunput’ defined but
 not used [-Wunused-function]
 1087 |     static void yyunput (int c, char * yy_bp )
      |                 ^~~~~~
[ 60%] Linking C static library ../../libflex.a
[ 60%] Built target flex
Scanning dependencies of target lexer
[ 80%] Building C object tests/lab1/CMakeFiles/lexer.dir/main.c.o
[100%] Linking C executable ../../lexer
[100%] Built target lexer
root@click:/home/click/桌面/cminus_compiler-2022-fall-master/build# ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  lexer  libflex.a  Makefile  src  tests
```

配置号编译环境之后，就可以开始做实验了。

首先，我们先找到各个 token 符号所对应的字符及它们所对应的数字，根据它们的英文名字及参考资料可以很容易地找出它们所代表的符号：

```
typedef enum cminus_token_type {
    //运算
    ADD = 259,// +
    SUB = 260,// -
    MUL = 261,// *
    DIV = 262,// /
    LT = 263,// <
    LTE = 264,// <=
    GT = 265,// >
    GTE = 266,// >=
    EQ = 267,// ==
    NEQ = 268,// !=
    ASSIN = 269,// =
    //符号
    SEMICOLON = 270,// ;
    COMMA = 271,// ,
    LPARENTHESE = 272,// (
    RPARENTHESE = 273,// )
    LBRACKET = 274,// [
    RBRACKET = 275,// ]
    LBRACE = 276,// {
    RBRACE = 277,// }
                                                    26,1          19%
```

```
    LBRACE = 276,// {
    RBRACE = 277,// }
    //关键字
    ELSE = 278,
    IF = 279,
    INT = 281,
    FLOAT = 281,
    RETURN = 282,
    VOID = 283,
    WHILE = 284,
    //ID和NUM
    IDENTIFIER = 285,// 变量名
    INTEGER = 286, // 整型
    FLOATPOINT = 287,// 浮点型
    ARRAY = 288,// 数组
    LETTER = 289,// 单个字母
    //others
    EOL = 290,// 换行符
    COMMENT = 291,// 注释
    BLANK = 292,// 空格
    ERROR = 258// 错误
                                                          35,13         63%
```

然后我们根据上面所对应的符号写出对应的正则表达式，例如，字母（不论大小写）的正则表达式可以是/[a-zA-Z]，这个正则表达式可以代表任意一个大写或小写字母。

我们可以在这个网站测试所写的正则表达式是否正确：

https://c.runoob.com/front-end/854/

所编写的正则表达式如下：

```
运算符号
ADD +号：
\+
SUB -号：
\-
MUL *号：
\*
DIV /号：
\/
LT <号：
\<
LTE <=号：
\<\=
GT >号：
\>
GTE >=号：
\>\=
EQ ==号：
\=\=
NEQ !=号：
\!\=
ASSIN =号：
                                                          1,10-7        顶端
```

```
ASSIN =号：
\=

语法符号
SEMICOLON ;号：
\;
COMMA ,号：
\,
LPARENTHESE (号：
\(
RPARENTHESE )号：
\)
LBRACKET [号：
\[
RBRACKET ]号：
\]
LBRACE {号：
\{
RBRACE }号：
\}
```

```
关键字
ELSE:
else
IF:
if
INT:
int
FLOAT:
float
RETURN:
return
VOID:
void
WHILE:
while
```

```
数字和字母
IDENTIFIER 标识符：
[a-zA-Z]+ //标识符只能有字母组成，这里+号表示一个或多个
INTGER 整型：
0|[1-9]+[0-9]* //注意这里不能用[0-9]*和[0-9]+，因为这样的话000000也会被识别成整型，事实上这不是一个合法的数字
FLOATPOINT 浮点型：
(0|[1-9]+[0-9]*)\.[0-9]* //这里表示一个整型后面加上0个或多个小数
ARRAY 数组：
\[\]
LETTER 单个字母：
[a-zA-Z]
```

```
其他
EOL 换行符：
\n
COMMENT 注释：
\/\*(?:[^\*]|\*+[^\/\*])*\*+\/ 注意：/**/是一个完整的注释，中间可以有多个/*，同时，中间无论出现什么字符都可以识别>
，直到遇到第一个*/
BLANK 空格：
[ \f\n\r\t\v]
ERROR 错误：
无需匹配，文件中已经帮我们填好了
                                                                79,16-11      底端
```

然后，我们根据给出的示例编写文件，给出的示例如下：

```
%{
//在%{和%}中的代码会被原样照抄到生成的lex.yy.c文件的开头，您可以在这里书写声明与定义
#include <string.h>
int chars = 0;
int words = 0;
%}

%%
 /*你可以在这里使用你熟悉的正则表达式来编写模式*/
 /*你可以用C代码来指定模式匹配时对应的动作*/
 /*yytext指针指向本次匹配的输入文本*/
 /*左部分（[a-zA-Z]+）为要匹配的正则表达式，
        右部分（{ chars += strlen(yytext);words++;}）为匹配到该正则表达式后执行的动作*/
[a-zA-Z]+ { chars += strlen(yytext);words++;}


. {}
 /*对其他所有字符，不做处理，继续执行*/

%%

```

如图，左边是正则表达式，右边是识别出应该执行的操作。

```
click@click:~/桌面/test$ ./a.out
Hello World

look, I find 2 words of 10 chars
click@click:~/桌面/test$
```

示例程序能够识别单词的词数。

有了正则表达式，我们就可以根据正则表达式写出对应的代码。

在 src/lexler/lexical_analyzer.l 下照下面示例的格式填写代码：

```
%{
//在%{和%}中的代码会被原样照抄到生成的lex.yy.c文件的开头，您可以在这里书写声明与定义
#include <string.h>
int chars = 0;
int words = 0;
%}

%%
 /*你可以在这里使用你熟悉的正则表达式来编写模式*/
 /*你可以用C代码来指定模式匹配时对应的动作*/
 /*yytext指针指向本次匹配的输入文本*/
 /*左部分（[a-zA-Z]+）为要匹配的正则表达式，
        右部分（{ chars += strlen(yytext);words++;}）为匹配到该正则表达式后执行的动作*/
[a-zA-Z]+ { chars += strlen(yytext);words++;}


. {}
 /*对其他所有字符，不做处理，继续执行*/

%%

```

要识别的 token 值就是上面的那一串正则表达式，在这里填写 flex 的动作：

```
%%

 /*****************TODO*******************/
 /****请在此补全所有flex的模式与动作  start******/
 //STUDENT TO DO

. {return ERROR;}



 /****请在此补全所有flex的模式与动作  end******/
%%
```

在这里填写 C 代码：

```
    while(token = yylex()){
        switch(token){
            case COMMENT:
                //STUDENT TO DO
            case BLANK:
                //STUDENT TO DO
                break;
            case EOL:
                //STUDENT TO DO
                break;
```

题目要求我们返回这些字符串的这些信息：

```
int lines;
int pos_start;
int pos_end;
```

行数，开始位置和结束位置，当然还有一个 token，但是那里已经在 /include/lexical_analyzer.h 头文件定义好了，我们只需要更新 lines，pos_start，和 pos_end 就行。

填写的代码如下：

```
/*运算符*/
/*ADD*/
\+ {pos_start = pos_end; pos_end++; return ADD;}
/*这里的意思是pos_start下一个字符的开始位置等于上一个识别出来的字符的结束位置，我们可以保证上一个字符的结束位置一定是下一个字符
的开始位置，因为我们可以识别空格，空格的结束位置，就是下一个字符的开始位置，这里因为只有一个符号，所以pos_end+1，下面的同理*/
/*SUB*/
\- {pos_start = pos_end; pos_end++; return SUB;}
/*MUL*/
\* {pos_start = pos_end; pos_end++; return MUL;}
/*DIV*/
\/ {pos_start = pos_end; pos_end++; return DIV;}
/*LT*/
\< {pos_start = pos_end; pos_end++; return LT;}
/*LTE*/
\<\= {pos_start = pos_end; pos_end += 2; return LTE;}
/*两个字符，加2*/
/*GT*/
\> {pos_start = pos_end; pos_end++; return GT;}
/*GTE*/
\>\= {pos_start = pos_end; pos_end += 2; return GTE;}
/*EQ*/
\=\= {pos_start = pos_end; pos_end += 2; return EQ;}
/*NEQ*/
!\= {pos_start = pos_end; pos_end += 2; return NEQ;}
/*ASSIN*/
\= {pos_start = pos_end; pos_end += 1; return ASSIN;}
```

```
/*语法符号*/
/*SEMICOLON*/
\; {pos_start = pos_end; pos_end += 1; return SEMICOLON;}
/*和上面的运算符一样*/
/*COMMA*/
\, {pos_start = pos_end; pos_end += 1; return COMMA;}
/*LPARENTHESE*/
\( {pos_start = pos_end; pos_end += 1; return LPARENTHESE;}
/*RPARENTHESE*/
\) {pos_start = pos_end; pos_end += 1; return RPARENTHESE;}
/*LBRACKET*/
\[ {pos_start = pos_end; pos_end += 1; return LBRACKET;}
/*RBRACKET*/
\] {pos_start = pos_end; pos_end += 1; return RBRACKET;}
/*LBRACE*/
\{ {pos_start = pos_end; pos_end += 1; return LBRACE;}
/*RBRACE*/
\} {pos_start = pos_end; pos_end += 1; return RBRACE;}
```

```
/*关键字*/
/*这里也跟上面一样，pos_end加上关键字的字符长度就行*/
/*ELSE*/
else {pos_start = pos_end; pos_end += 4; return ELSE;}
/*IF*/
if {pos_start = pos_end; pos_end += 2; return IF;}
/*INT*/
int {pos_start = pos_end; pos_end += 3; return INT;}
/*FLOAT*/
float {pos_start = pos_end; pos_end += 5; return FLOAT;}
/*RETURN*/
return {pos_start = pos_end; pos_end += 6; return RETURN;}
/*VOID*/
void {pos_start = pos_end; pos_end += 4; return VOID;}
/*WHILE*/
while {pos_start = pos_end; pos_end += 5; return WHILE;}
```

```
/*数字和字母
IDENTIFIER*/
[a-zA-Z]+ {pos_start = pos_end; pos_end += strlen(yytext); return IDENTIFIER;}
/*跟上面一样，pos_start等于上一个字符串的结束位置，pos_end要加上现在识别到的字符串的长度，由示例我们知道长度是
strlen(yytext)*/
/*INTGER*/
0|[1-9]+[0-9]* {pos_start = pos_end; pos_end += strlen(yytext); return INTEGER;}
/*FLOATPOINT*/
(0|[1-9]+[0-9]*)\.[0-9]* {pos_start = pos_end; pos_end += strlen(yytext); return FLOATPOINT;}
/*ARRAY*/
\[\] {pos_start = pos_end; pos_end += strlen(yytext); return ARRAY;}
/*LETTER*/
[a-zA-Z] {pos_start = pos_end; pos_end += 1; return LETTER;}
```

```
/*其他字符，下面的C代码已经给出了要处理的动作，现在这里我们直接返回对应的字符就好了*/
/*EOL*/
\n {return EOL;}
/*COMMENT*/
\/\*(?:[^\*]|\*+[^\/\*])*\*+\/ {return COMMENT;}
/*BLANK*/
[ \f\n\r\t\v] {return BLANK;}
/*ERROR*/
. {return ERROR;}
```

下面是 C 代码：

```
        case COMMENT:
            //STUDENT TO DO，遇到注释的动作
            pos_start = pos_end;
            for(int i = 0; i < strlen(yytext); i++){
                    if(yytext[i] == '\n'){//注意，换行符的长度是1，只是用两个符号表示而已
                            pos_end = 1;
                            pos_start = 1;
                            lines++;
                    } else{
                            pos_end++;
                    }
            }
            break;
```

```
        case BLANK:
            //STUDENT TO DO
            pos_start = pos_end;
            pos_end += strlen(yytext);
            //strlen(yytext)获取文本长度，这里可以获取到全部的空格长度
            break;
        case EOL:
            //STUDENT TO DO
            pos_start = 1;
            pos_end = 1;
            lines += strlen(yytext);
            //获取有多少个换行符，换行之后重新计数
            break;
```

ERROR 的情况已经写好了，我们不用写。

实验结果验证

请提供部分自己的测试样例

首先是助教的测试样例：

```
click@click:~/桌面/cminus_compiler-2022-fall-master$ sudo su
[sudo] click 的密码：
root@click:/home/click/桌面/cminus_compiler-2022-fall-master# cd build/
root@click:/home/click/桌面/cminus_compiler-2022-fall-master/build# make lexer
[ 60%] Built target flex
[100%] Built target lexer
root@click:/home/click/桌面/cminus_compiler-2022-fall-master/build# cd ..
root@click:/home/click/桌面/cminus_compiler-2022-fall-master# ./build/lexer ./tests/lab1/testcase/1.cminus out
[START]: Read from: ./tests/lab1/testcase/1.cminus
[END]: Analysis completed.
root@click:/home/click/桌面/cminus_compiler-2022-fall-master# python3 ./tests/lab1/test_lexer.py
Find 6 files
[START]: Read from: ./tests/lab1/testcase/6.cminus
[END]: Analysis completed.
[START]: Read from: ./tests/lab1/testcase/1.cminus
[END]: Analysis completed.
[START]: Read from: ./tests/lab1/testcase/4.cminus
[END]: Analysis completed.
[START]: Read from: ./tests/lab1/testcase/5.cminus
[END]: Analysis completed.
[START]: Read from: ./tests/lab1/testcase/3.cminus
[END]: Analysis completed.
[START]: Read from: ./tests/lab1/testcase/2.cminus
[END]: Analysis completed.
root@click:/home/click/桌面/cminus_compiler-2022-fall-master# diff ./tests/lab1/token ./tests/lab1/TA_token
root@click:/home/click/桌面/cminus_compiler-2022-fall-master#
```

下面是自己编写的测试样例：

```
/*//*/
/* /** */
int test[];
int test[10000];
int test[abc];
```

查看输出：

```
root@click:/home/click/桌面/cminus_compiler-2022-fall-master# ./build/lexer ./tests/lab1/testcase/7.cminus out
[START]: Read from: ./tests/lab1/testcase/7.cminus
[END]: Analysis completed.
root@click:/home/click/桌面/cminus_compiler-2022-fall-master# cat out
int     280     3       1       4
test    285     3       5       9
[]      288     3       9       11
;       270     3       11      12
int     280     4       1       4
test    285     4       5       9
[       274     4       9       10
10000   286     4       10      15
]       275     4       15      16
;       270     4       16      17
int     280     5       1       4
test    285     5       5       9
[       274     5       9       10
abc     285     5       10      13
]       275     5       13      14
;       270     5       14      15
```

可以看到，注释已经被识别到了，因为没有输出。

实验反馈

这个实验总体来说不是很难，重点是根据字符串，写出对应的正则表达式，然后 pos_start 和 pos_end 的更新其实都大同小异。重点需要注意注释的正则表达式。可以在上述我提供的网站中练习正则表达式的编写。其实实验环境也可以用乌班图 20.04，不一定要用 WSL，WSL 的图形化界面总有些奇奇怪怪的 BUG。还有，

学会了一些开发中实用的 git 操作，同时，了解了一些 C-Minus-f 的语法规则。