1. **源语言的描述**

基于C语法设计的源语言LittleC

（1）结构和语义说明：

a. 该语言的一个程序由且仅由一个块结构组成，该块中包含两部分：可选的声明语句和必须的执行语句。声明语句用于声明变量，执行语句用于表达计算过程，过程中使用这些变量来存储内部数据和计算结果。

b. 该语言支持两种数据类型：整型和布尔型。整型数据占4个字节，布尔型占1个字节。整型数据的常量是整常数，布尔型数据的常量是 true 和 false 。

c. 整型数据支持 +、-、\*、/ 四种算术运算，运算结果为整型值。整型数据可与>、>=...等六个关系运算符结合，运算结果为布尔型值。

d. 布尔型数据支持 || 、&& 、！三种逻辑运算，运算结果为布尔常量。

e. 整型的变量和常量可以用算术运算符组合成算术表达式，其值为整常数；布尔型的变量和常量可以用逻辑运算符组成成布尔表达式，其值为布尔常量；算术表达式可以加上关系运算符构造出布尔表达式。

f. 整常数和算术表达式可以用于整型变量赋值，布尔常量和布尔表达式可以用于布尔型变量赋值；两者不兼容（文法中使用不同的赋值运算符以示区别）。

g. 各种运算符的优先级和结合规则参照C语言语法理解。

h. 控制流语句中的控制条件限定为1个布尔型变量，该量应该在引用之前定值。

i. 该语言不支持数组、结构体、指针等复杂数据类型。

j. 该语言不含有子程序结构，也不支持过程调用。

k. 该语言有简单的I/O功能：使用read指令能够从键盘读入1个整常数，使用write指令能向屏幕打印输出1个整常数。

（2）文法定义：

PROG        →    **{**  DECLS  STMTS  **}**

DECLS       →    DECLS  DECL    **|**empty

DECL         →    **int**NAMES  **;  |  bool**NAMES  **;**

NAMES     →    NAMES **,**  NAME  **|**  NAME

NAME       →    **id**

STMTS    →    STMTS  STMT  **|**   STMT

STMT      →    **id  =**  EXPR **;    |   id :=** BOOL **;**

STMT      →    **if  id**  **then** STMT

STMT      →    **if   id**  **then** STMT  **else** STMT

STMT      →    **while   id** **do** STMT

STMT      →    **{**  STMTS   STMT  **}**

STMT      →    **read  id  ;**

STMT      →    **write  id  ;**

EXPR    →    EXPR  ADD  TERM  **|**  TERM

ADD     →   **+ | -**

TERM    →    TERM  MUL NEGA  **|** NEGA

MUL     →    **\* | /**

NEGA   **→**FACTOR  **|  -** FACTOR

FACTOR→    **(**  EXPR **) |  id  |** **number**

BOOL    →    BOOL  **||**  JOIN    **|**    JOIN

JOIN     →    JOIN   **&&**   NOT **|**  NOT

NOT      →    REL   **|  !**REL

REL       →    EXPR   ROP  EXPR

ROP      →     **>  |  >=  |  <  |  <=  |  ==  |   !=**。

1. **词法规则描述和分析与设计**

a. 标识符：由字母打头后跟字母、数字任意组合的字符串；长度不超过8；不区分大小写；把下划线看作第27个字母；

b. 整常数：完全由数字组成的字符串；正数和0前面不加符号，负数在正数前面加-构成；长度不超过8；

c. 布尔常量：true 和 false 。

d. 关键字、运算符、分隔符仅包含在文法定义中出现过的终结符。关键字保留。

e. 字母表定义为上述规则中出现的字符的集合；不在该集合中的符号都以非法字符对待；

f. 源程序中可以出现单行注释和多行注释，参考C语言设计。

词法分析的主要任务是根据给定文本，产生相应的单词符号文本。单词符号应该是一个二元组，二元组中第一个元素是识别出的字符串，第二个元素应该是字符串的类型。词法分析器其本身就相当于一个源程序语言的预处理器和扫描器

要实现词法分析器我们要首先明确有哪几种单词符号。

从文法定义上来看文法符号主要分为以下几种

【1】关键词

根据语法定义关键词有以下九个：

Int bool if then while do read write else

【2】运算符

根据语法定义运算符有以下十四个：

+ - \* / || && ! >= > <= < == := =

【3】分隔符

根据语法定义分隔符有以下六个：

( ) { } , ;

【4】布尔类型

根据语法定义布尔类型有以下两种:  
 true false

【5】标识符

如果上述所有类型都无法匹配字符串，而且字符串以字母开头，由字母和数字组成，就将其识别为标识符

【6】整数常数

如果一个字符串由数字打头，且由数字组成则将其识别为整数常数

【7】无效词

上述所有类型都无法识别，则识别为无效词

在明确上述七个类型和每个类型的字符串组成后就可以开始对文本进行分类。遇到空格的时候直接跳过。遇到字符串就对字符串进行分类处理。

最后将所有分类结果输出到指定文件当中

1. **程序的主要流程**

【1】先通过枚举类型列出所有字符串类型和固定的字符串（关键字）。

【2】通过识别函数识别每个字符串所属的类型和进行省略符号的省略操作

【3】添加驱动模块与核心识别模块封装成一个类。这个类的实例化函数的参数是两个字符串，第一个字符串表示源程序的路径，第二个字符串表示结果的输出路径。

【4】对实现的字符串进行测试。

1. **关键算法**

【1】KMP算法：通过KMP算法的实现对省略字符串的匹配以跳过中间所有代码。通过通过字符串匹配找到相应的关键字类型

1. **总体结构**

【1】定义了一个名为Token的枚举类型来列举固定字符串的类型

Token类中有七个类型：  
 IDENTIFIER, //标识符

NUMBER, //整常数

BOOL, //布尔常量

KEYWORD, //关键字

OPERATOR, //运算符

SEPARATOR, //分隔符

INVALID\_TOKEN, //无效词

【2】定义了一个名为lexical\_analyzer的类

public部分只有三个函数:

lexical\_analyzer(const string& input\_file\_name, const string& output\_file\_name);

~lexical\_analyzer();

void analyze();

private部分先定义了四个无序关联式容器：

unordered\_map<string, Token> keyword\_map //存所有关键字

unordered\_map<string, Token> operator\_map //存所有操作符

unordered\_map<string, Token> separator\_map //存所有分割符

unordered\_map<string, Token> bool\_map //存所有bool类型，true和false

private部分又定义了多个实现analyzer所必须的中间函数

Token get\_keyword(const string& str); //判断是否是关键字

Token get\_operator(const string& str); //判断是否是运算符

Token get\_separator(const string& str); //判断是否是分隔符

Token get\_number(const string& str); //判断是否是整数常量

Token get\_bool(const string& str); //判断是否是字符常量

Token get\_explanatorynote(const string& str); //判断是否是注释

void write\_token(Token token, const string& word);//输出到文件

ifstream input\_file; //输入

ofstream output\_file; //输出

string find\_operator(const string& str, int i); //从文件当前位置匹配操作符

string find\_bool(const string& str, int i); //从文件当前位置匹配bool

string find\_separator(const string& str, int i); //从文件当前位置匹配分隔符

【3】在类之外定义了以后可能会用到的字符串匹配算法：

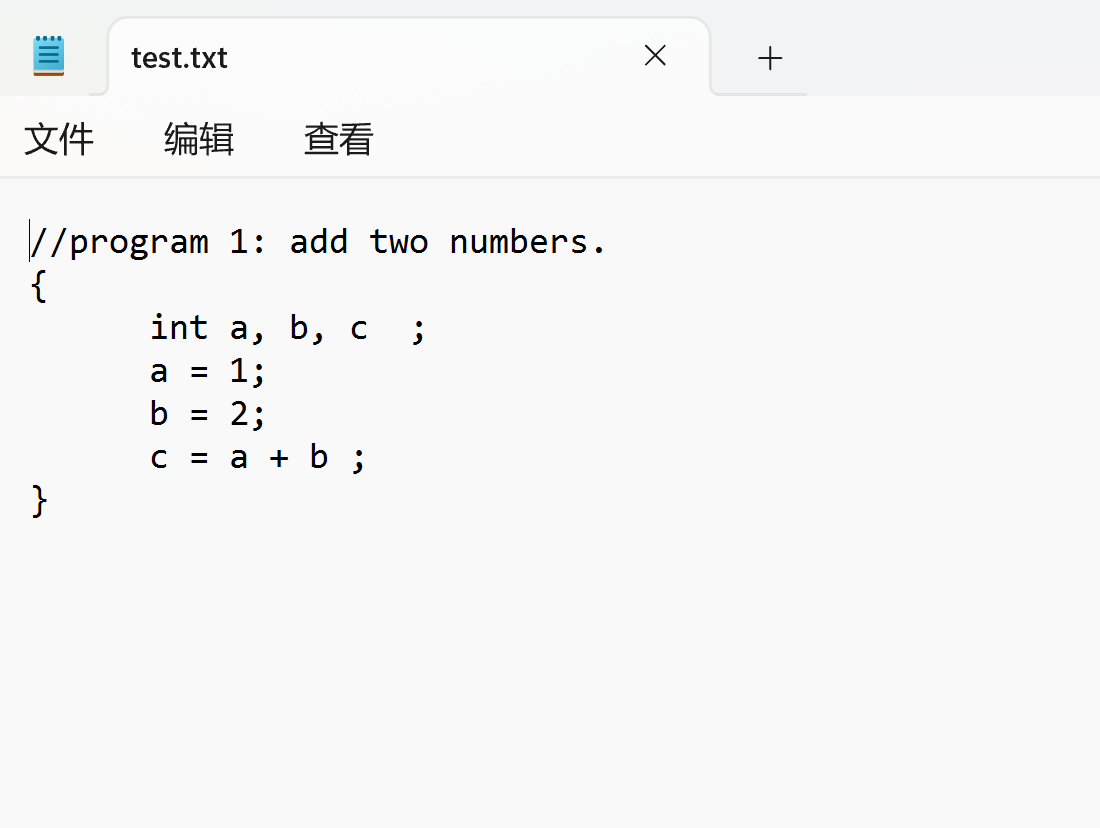
void CreatNext\_E(string p, int next[])

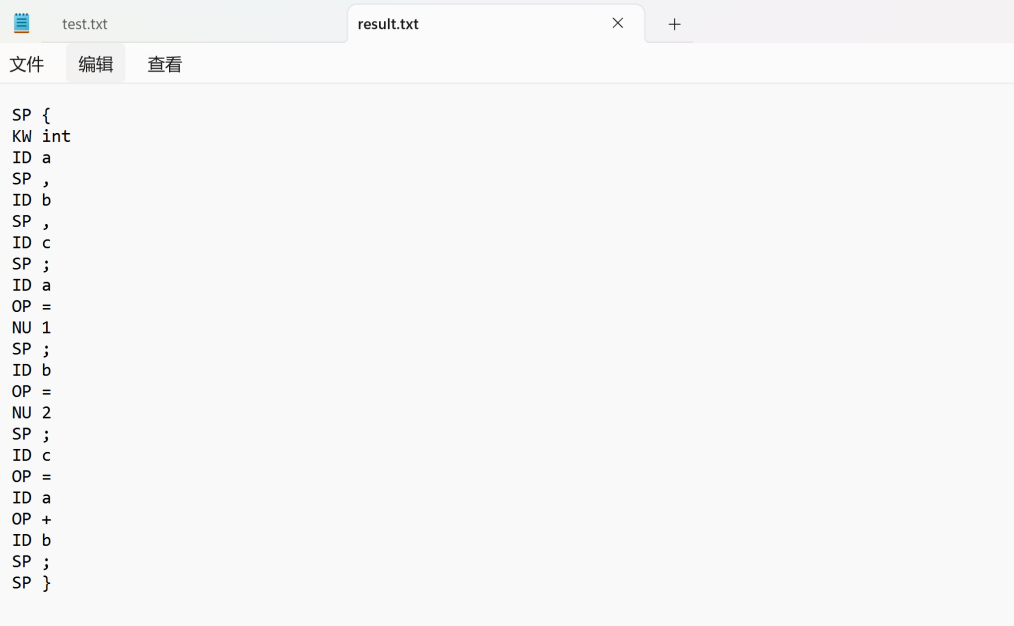
int KMP\_find\_string\_in\_string(string t, string p,int current) //上面两个函数实现KMP

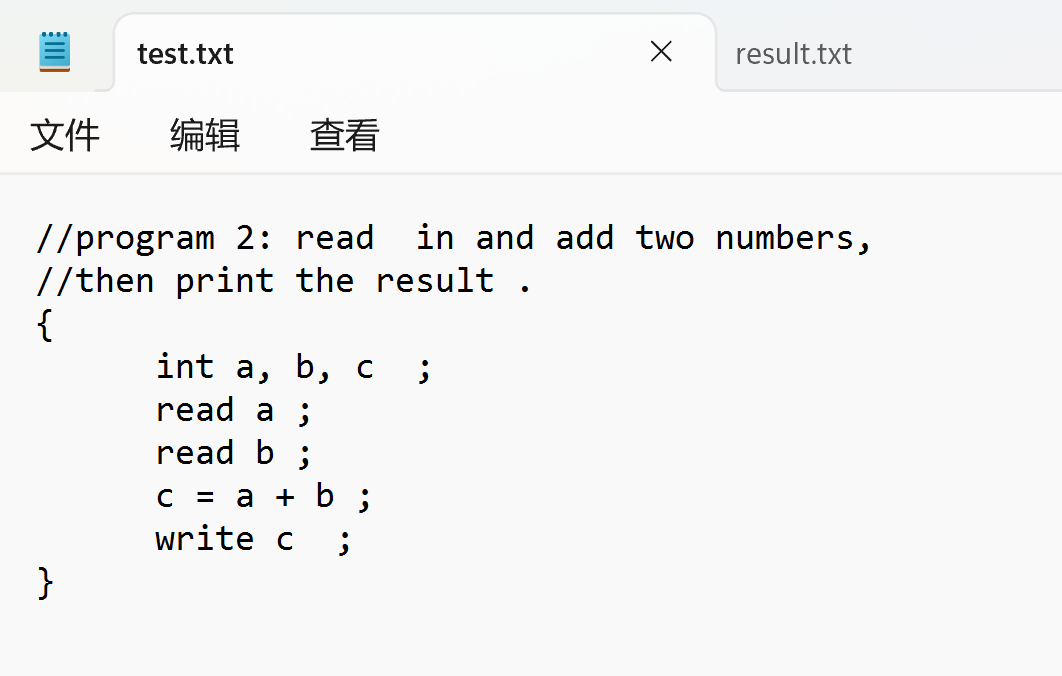
int BF\_find\_string\_in\_string(string t, string p,int current) //实现暴力字符串匹配算法

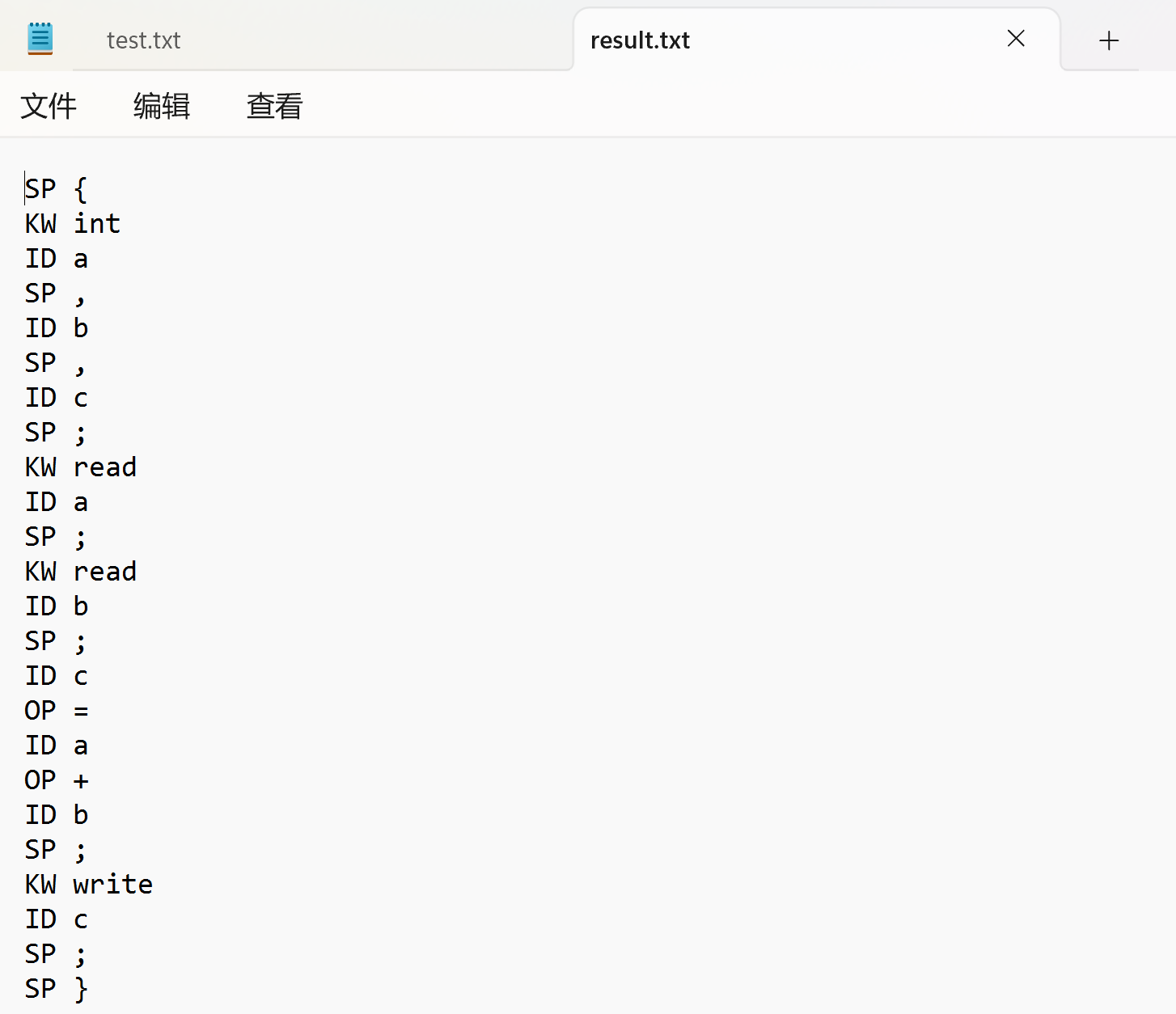
整体代码在文件包中

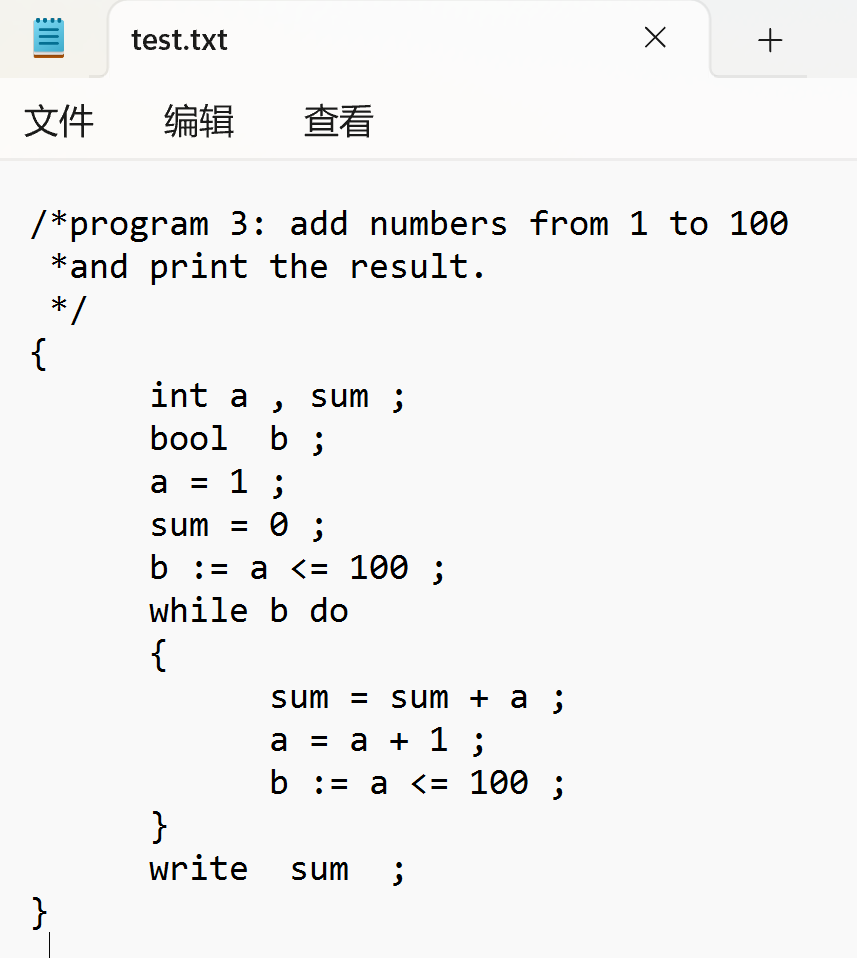
**六．测试用例与程序运行截图**











结果：

SP {

KW int

ID a

SP ,

ID sum

SP ;

KW bool

ID b

SP ;

ID a

OP =

NU 1

SP ;

ID sum

OP =

NU 0

SP ;

ID b

OP :=

ID a

OP <=

NU 100

SP ;

KW while

ID b

KW do

SP {

ID sum

OP =

ID sum

OP +

ID a

SP ;

ID a

OP =

ID a

OP +

NU 1

SP ;

ID b

OP :=

ID a

OP <=

NU 100

SP ;

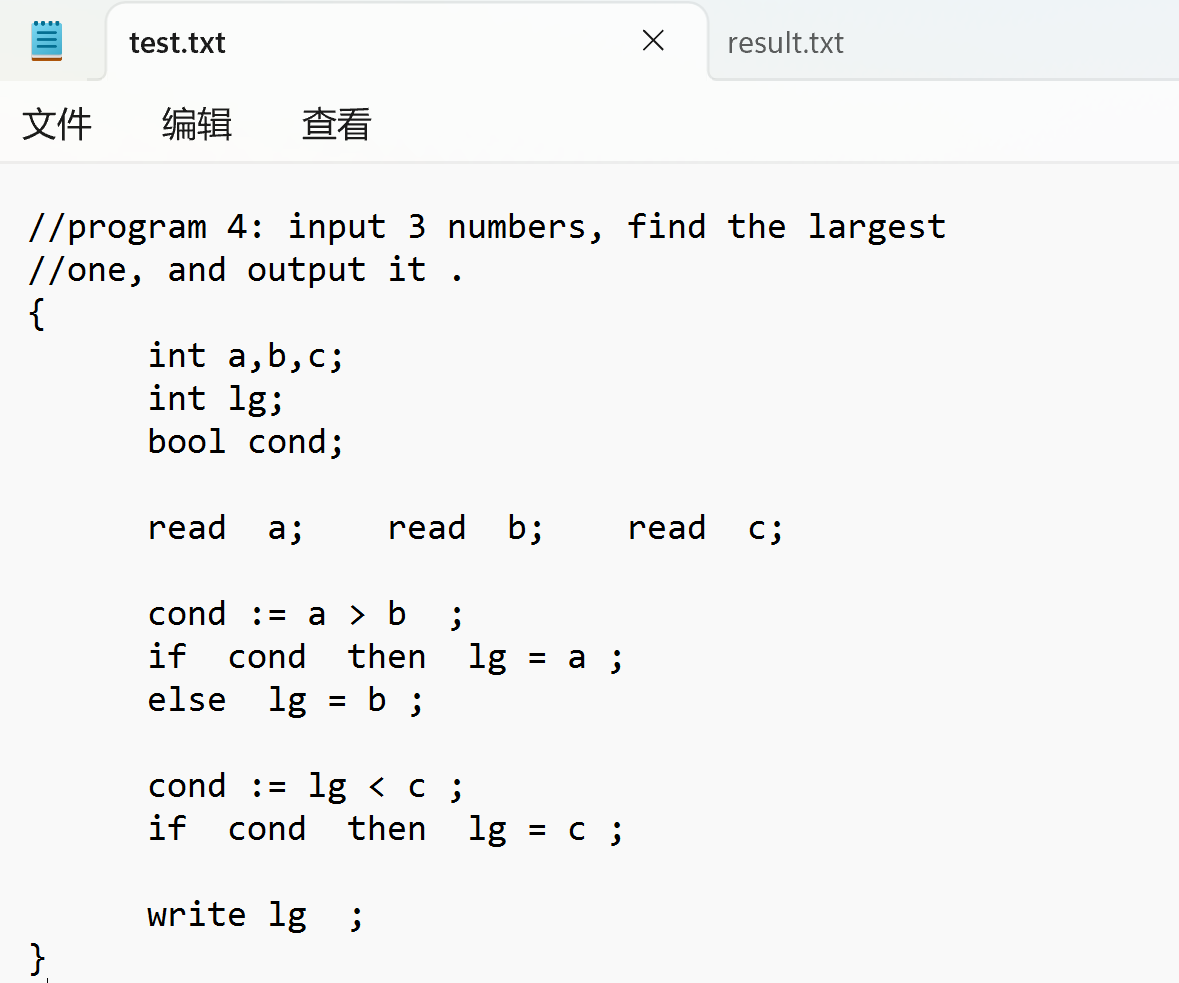
SP }

KW write

ID sum

SP ;

SP }



SP {

KW int

ID a

SP ,

ID b

SP ,

ID c

SP ;

KW int

ID lg

SP ;

KW bool

ID cond

SP ;

KW read

ID a

SP ;

KW read

ID b

SP ;

KW read

ID c

SP ;

ID cond

OP :=

ID a

OP >

ID b

SP ;

KW if

ID cond

KW then

ID lg

OP =

ID a

SP ;

KW else

ID lg

OP =

ID b

SP ;

ID cond

OP :=

ID lg

OP <

ID c

SP ;

KW if

ID cond

KW then

ID lg

OP =

ID c

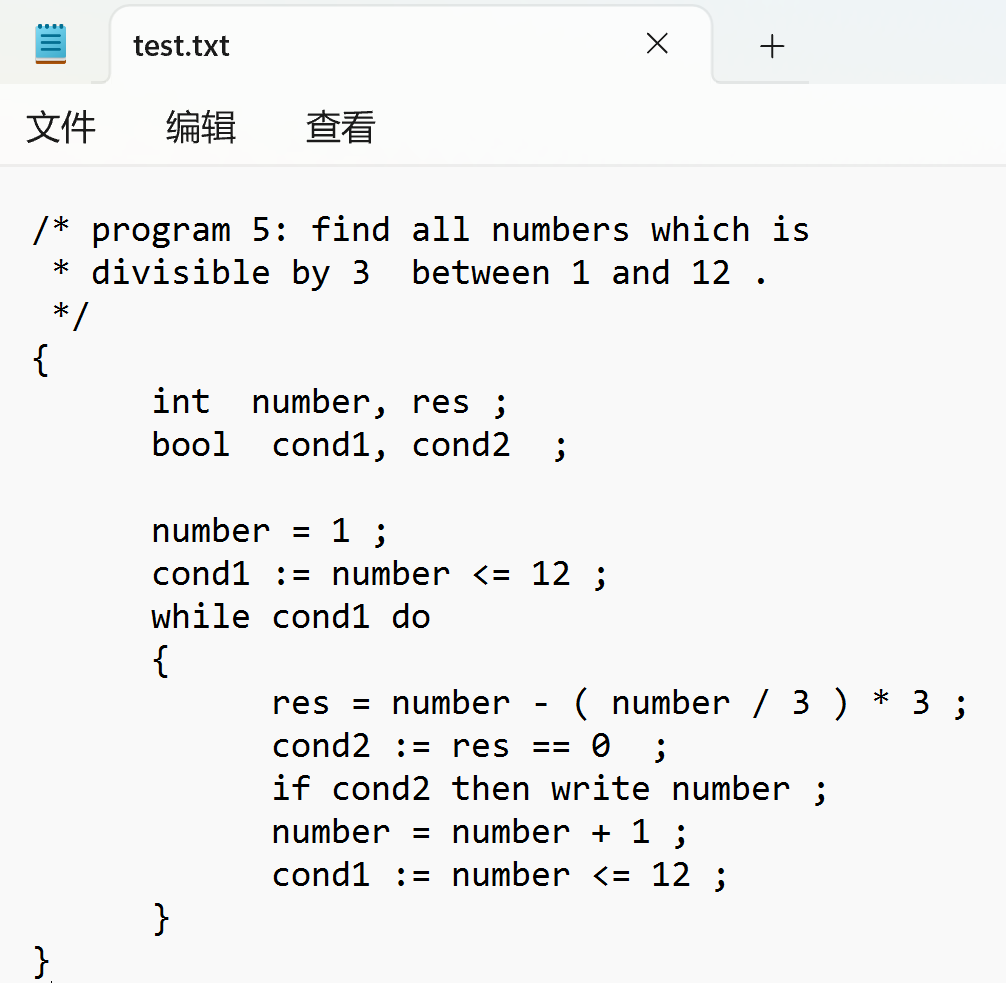
SP ;

KW write

ID lg

SP ;

SP }



结果：

SP {

KW int

ID number

SP ,

ID res

SP ;

KW bool

ID cond1

SP ,

ID cond2

SP ;

ID number

OP =

NU 1

SP ;

ID cond1

OP :=

ID number

OP <=

NU 12

SP ;

KW while

ID cond1

KW do

SP {

ID res

OP =

ID number

OP -

SP (

ID number

OP /

NU 3

SP )

OP \*

NU 3

SP ;

ID cond2

OP :=

ID res

OP ==

NU 0

SP ;

KW if

ID cond2

KW then

KW write

ID number

SP ;

ID number

OP =

ID number

OP +

NU 1

SP ;

ID cond1

OP :=

ID number

OP <=

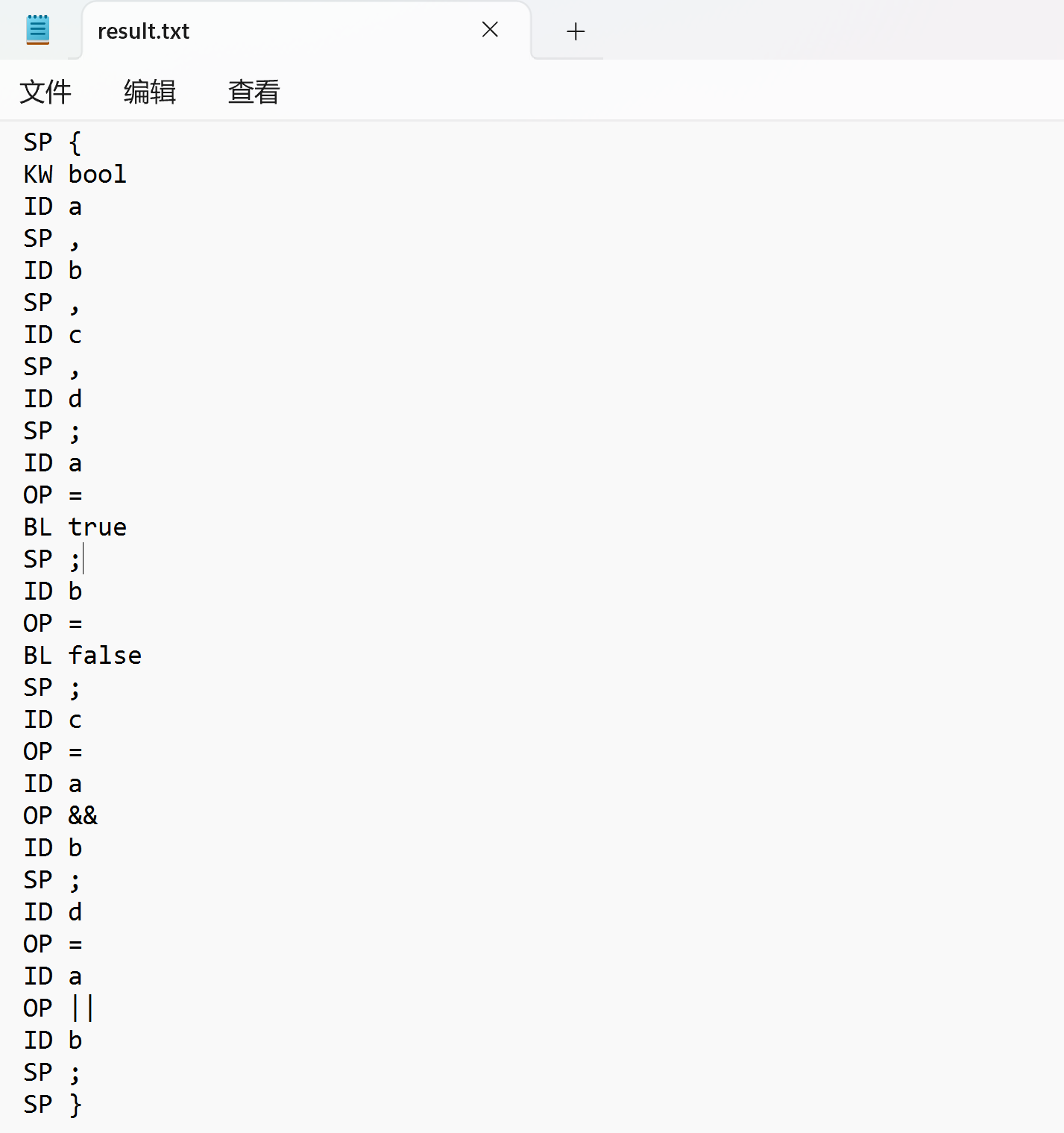
NU 12

SP ;

SP }

SP }





1. **实验总结**

【1】语言环境：C++

【2】编译环境：

#include<iostream>

#include<fstream>

#include<string>

#include<unordered\_map>

【3】自我分析（遇到的问题，如何解决等）

开始我使用字符串匹配算法。发现省略字符的识别和多字符字符串的整体识别实现起来很麻烦，于是我就使用了BP算法。程序就很好地运行了。

不过我在实现完成后发现使用暴力字符串匹配算法后程序的运行效率不高，于是我使用了改进的字符串匹配算法KMP。

改进之后程序的整体运行时间明显降低。最后完成了词法分析的程序