# 1.编译运行环境说明

开发语言：C语言

开发环境：windows 7 64bit 下的Code::Blocks 13.12 with TDM-GCC compiler, version 4.7.1

程序运行环境：已在windows 7 64bit测试通过。

主函数所在文件为main.c 。工程文件为Code::Blocks project files类型的regex.cbp文件。

# 2.设计结构和各模块功能

模块关系图如图1所示。主模块为正则表达式主模块（regexSearch.h、regexSearch.c），它包含了三个子模块：词法分析模块（头文件lexicalAnalysis.h、lexicalAnalysis.c）、元字符匹配模块（头文件match.h、match.c）和扫描匹配模块（nfa.h、nfa.c）。

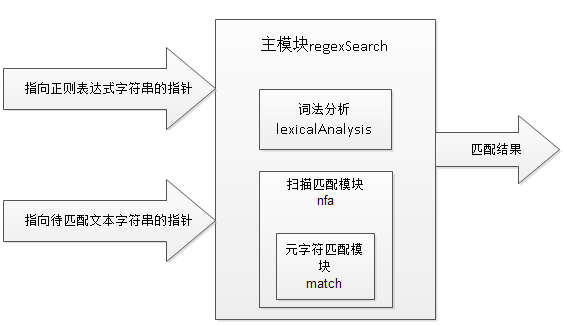


图 1 模块关系图

该regexSearch模块的输入输出定义让它可以方便的拿到其他程序中应用。

此次比赛的IO规范由main.c完成。main.c会读取同目录下的regex.txt ，并判断有多少行，每读取一行正则规则，就打开对应的textN.txt文件读入一维数组缓冲区，调用主模块regexSearch得到结果，输出到对应的outputN.txt ，然后释放textN.txt缓冲区，关闭textN.txt文件，继续读取下一行，依次循环。

## 2.1 主模块

该模块用于根据正则表达式匹配文本并输出匹配结果。

### 2.1.1 对外提供的函数

主模块主要提供了函数int regexSearch(char \* regex\_text, char \* text, char \* result) 。输入参数依次为指向正则表达式字符串的指针，指向带匹配文本字符串的指针，以及用来存放匹配结果的缓冲区指针。该函数的返回值是0匹配失败 1匹配成功 其他 表达式有错误。

另外，还对外提供了int charType(const char \* ch)函数，供三个子模块调用。该函数输入一个字符指针，返回的是该字符的类型。字符类型编码如下

普通字符用1表示。其他字符按照要求分为4组。第一组为元字符，第二组为重复次数，第三组为分支符号，第四组为分组和后项引用。具体编码见表1.

表 1 类型编码

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 组别 | 编码 | 表示类型 | 组别 | 编码 | 表示类型 |
| 1 | 11 | . | 2 | 22 | {n,m} |
| 1 | 12 | ^和$ | 3 | 31 | | |
| 1 | 13 | \b \B | 41 | 41 | ( |
| 1 | 14 | \w \s \d \W \S \D | 42 | 42 | ) |
| 1 | 15 | [x] | 43 | 43 | \2后项引用 |
| 2 | 21 | ？ \* + | 无 | 1 | 普通字符 |

### 2.1.2 内部函数

static BOOL destroyList(struct regex\_list \* head)。用于销毁本次匹配正则表达式分词后生成的数据结构。

## 2.2词法分析模块（lexicalAnalysis）

该模块用于将正则表达式数组处理后生成一个保存了一个个正则单元的链表结点。例如正则表达式(\d|[abc]\*|d)+\3 经过分词处理后变成图2这样

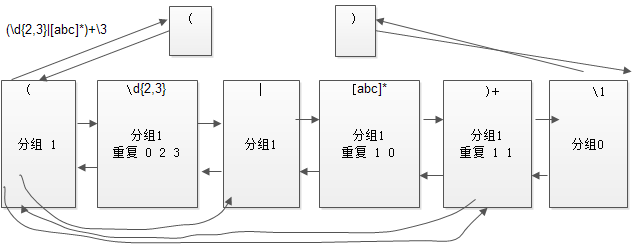


图 2 词法分析举例

其中每个结点均记录了所属的所有分组，以group[]存储，下标值n从0开始，如果group[n]==1，则该结点属于n组。group\_id记录了所属最小的分组。

每个左括号结点记录了指向属于该组的分支符号的指针数组，配对的左右括号互相指向。关于结点结构体的设置请参见2.5 数据结构说明。

### 2.2.1 对外提供的函数

int lexicalAnalysis(char \* arr, struct regex\_list \* head)函数。

该函数输入为指向正则表达式字符串的指针 和 保存分词后信息的链表头结点

该函数会将分词的结果存入链表

该函数返回值 如果正则表达式有错误 返回错误码 否则返回1。如果分配结点所需内存失败，程序会退出。

### 2.2.2 内部函数

该模块内部有三个函数。

1. charToInt()用于将字符串变为整数返回，供setRate()函数使用。
2. static void setRate(char\* pchar, int rate[3])

将pchar指向的字符串（可能的重复次数信息）编码为3位数字 保存到rate数组。

重复次数和3位数字的编码关系如下

第一位0表示出现次数是rate[1]到rate[2]的闭区间

第一位1表示出现次数是 rate[1]次及以上

例如

\* 编码为 1 0

+ 编码为 1 1

? 编码为 0 0 1

{n} 编码为 0 n n

{n,} 编码为 1 n

{n,m} 编码为 0 n m

其中\*等没有上限可以支持超过1024个字符。

1. static struct regex\_list \* saveToList(struct regex\_list \* head, struct regex\_list \* p, char \* arr, int low, int high, int posi\_start, int \* group, int top)

函数的功能是将一个正则单元的所有信息保存到链表结点 供lexicalAnalysis()函数调用

输入参数：

struct regex\_list \* head 链表头结点

struct regex\_list \* p 保存当前正则单元的结点

char \* arr 正则表达式数组

int low 该单元在数组中起始偏移量

int high 该单元在数组中结尾偏移量

int posi\_start 将元字符和重复次数信息分开的偏移量

int \* group 用于处理分组的栈数组

int top 栈数组的栈顶位置

返回值：返回保存这些单元正则的结点指针

## 2.3 nfa扫描匹配模块

该模块对分词后的正则链表和文本进行匹配。

### 2.3.1 对外提供的函数

BOOL scanListWithCharPosi(struct regex\_list \* const plow, struct regex\_list \* const phigh, const char \* text, const int start\_char, int \* presult)

该函数的输入参数

struct regex\_list \* const plow 一串正则单元的开始结点

struct regex\_list \* const phigh 一串正则单元的结束结点

const char \* text 指向文本

const int start\_char 文本的从哪个字符开始匹配

int \* presult 指向一个整数 保存当前状态匹配到了文本中的字符的偏移量

该函数的返回值是是否匹配成功。

该函数是本程序中最复杂的一个程序，实现的简要思路如下，具体流程请看源代码：

i指向文本，p指向正则，两个交替前进。p没有指向链表结尾的时候，一直循环如下内容：遇见左括号，将括号中的各个分支的内容（链表结点范围）依次传入自己进行递归调用。判断字符单元时，调用match函数，匹配后，根据是否是零宽度决定是否增加i。根据单元的重复次数决定是否继续判断当前结点p和位置i 重复次数达到要求时，p指向下一个正则单元，继续匹配。当匹配失败的时候（不管是左括号还是普通单元），判断重复次数是否可以为0，为0的话也是成功，否则该结点和文本匹配失败，调用canGreedyBack判断是否可以回退（如果该结点所属分组已经重复多次则不需回退，只需该次重复记为不匹配），如果可以回退，调用greedyBack函数将p前移，如果有重复了多次且大于最低要求的，将该结点匹配次数减一，i置为该结点去掉一次匹配后的位置，p后移，回到循环继续判断。

### 2.3.2 内部函数

1. static BOOL canGreedyBack(struct regex\_list \* p)

该函数用于判断一个结点不匹配的时候是否需要对之前的结点进行回退。

2. static BOOL greedyBack(struct regex\_list \*\* p, int \* i)

该函数用于贪婪回退 会修改p和i为回退后下一个结点和文本位置。

## 2.4 元字符匹配模块 （match）

该模块用于判断元字符和字符是否匹配，例如\d和数字5匹配，[a-f]和b匹配,^和行首匹配等。

该模块的输入输出见图3.

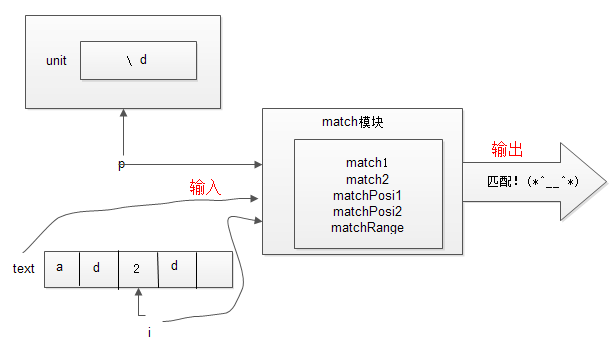


图 3 match模块输入输出

### 2.4.1 对外提供的函数

BOOL match(const struct regex\_list \* p, const char \* text, int i);

用于判断某字符和表达式是否匹配。p结点的unit域存放了正则规则。它是通过判断类型后，调用内部的5个函数实现的。

### 2.4.2 内部函数

内部函数分别为判断不同类型是否匹配的函数。

1. BOOL matchPosi1(const char\* pchar, const char\* text, int i); 用于匹配位置^ $
2. BOOL match1(const char \* pchar, const char ch); //用于匹配.和其他普通字符
3. BOOL matchPosi2(const char\* pchar, const char\* text, int i); 判断 两个字符的位置\b \B
4. BOOL match2(const char\* pchar, const char ch); 判断2个字符的元字符 \w \s \d \W \S \D
5. BOOL matchRange(const char \*pchar, const int len, char ch); 判断方括号形式[a] [a-z]

## 2.5 数据结构说明

分词后的信息保存在一个双向链表中。所有的模块均使用这种结构体。该链表的结点结构体声明如下

struct regex\_list

{

char unit[UNIT\_LEN]; //保存单位正则字符

int posi\_start; //分割元字符和出现次数的下标。unit[0]到unit[posi\_start-1]是元字符

int rate[3]; //三位数字 记录重复次数 编码见2.1.1小节

int group[MAX\_GROUP]; //记录分组信息。保存该结点所属的所有分组 group[n]==1则属于第n组

int is\_posi; //记录该元字符是不是匹配位置信息（零宽度）

int is\_back\_ref; //记录该元字符是不是分组后的后项引用

struct regex\_list \* pred;

struct regex\_list \* next; //构成双向链表

int low;

int high; //low到high闭区间是 该单元匹配到的字符的下标范围

int matched\_num; //仅用于右括号和后项引用 记录该组匹配了几次 用于贪婪回退

int matched\_char\_posi[MAX\_MATCHED\_CHAR\_POSI][2]; //仅用于右括号和后项引用单元 记录每次匹配位置 第一次为[ matched\_arr[0],matched\_arr[1] ) 最后一次为 [matched\_arr[matched\_num-1], matched\_arr[matched\_num]) 。第二维度 记录每次匹配是用的该分组内的哪个分支

int branch\_num; //如果是左括号或头结点 记录分支符号|的数量

struct regex\_list \* pbranch[MAX\_BRANCH]; //如果是左括号或头结点 记录该组内的所有分支|的位置 从1开始

int group\_id; //如果是括号记录该括号是第几个分组 或该单元所属的最小分组

struct regex\_list \* pbracket; //如果是括号 这个指针指向和该括号配对的另一个括号

};

# 3. 功能实现情况

## 3.1 完成的功能

1、支持所有元字符 . \w \s \d \W \S \D ^ $ \b \B [x] [^x]

例如： [a-f4-7]\w\d\s\W\D\S 可以匹配6d3（换行）#!

a$\s \babc匹配a（换行）（空格）abc会输出a（换行）（空格）abc

2、支持任意元字符和重复量词（\* + ? {n} {n,m}{n,}）组合

例如： [a-d]\*\d{3}-\d{7}匹配sdfabcd321-123456789得到abcd321-1234567

1. 支持元字符和重复量词的一重贪婪回退

\d+.+12 匹配adsfs123abc3abc12ashdfi会得到123abc3abc12

1. 支持分支符号

a+|c 匹配aaaad会得到aaaa 匹配dc会得到c

1. 分组和后向引用

分组和后项引用： (\d+)\1匹配781541544得到154154

1. 分组和分支的结合

(ab|cd)(ef|gh)\1\2匹配111abghabghijo得到abghabgh

1. 分组、分支和重复量词的结合

(ab|cd){2,3}\1匹配ergafabcdabcdabrg得到abcdab

(a|p)\*apapp匹配apaapappppappapapappaapp得到apaapappppappapapapp

## 3.2 额外的功能：

1、支持一些较简单的需多次回退的表达式

.\*\w\*\d\*45匹配abcd123456会得到abcd12345 。其中\d\* \w\* 和.\*分别回退了。

(\d+)(.+)(\2\1)匹配adsfs123abc3abc12ashdfi得到123abc3abc12。其中.+和\d+分别回退

2. 支持括号不匹配错误的检测。括号不匹配时，会在屏幕输出wrong regex: no matched brackets!不生成结果outputN.txt文件。暂不支持其它错误检测，不过预留了实现检测更多错误后返回的接口。

# 4. 其他补充说明

## 4.1一些功能补充：

1. 打开文本文件时动态申请内存，可以打开大文件。匹配结束后会释放内存。
2. 处理\* +等没有上限的重复的时候不是采用<1024 而是没有上限，只要可以会一直重复。

## 4.2 关于代码的补充

1、将正则表达式词法分析后生成的链表设置头结点和尾结点 分别看作左括号和右括号，整个表达式相当于分组0 。这种设计可以减少特殊情况的处理。

2、分组信息的处理。分组信息用到了栈。栈顶指针top，设置一个变量i=0，遇到左括号，++i并将i入栈，遇见右括号 top--，这样栈中始终能够保存每个节点所属的所有分组。这种处理十分简单有效。