第一章　第四届国际奥林匹克信息学竞赛
中国组队赛试题分析

§1.1　表达式求值

一、试　　题

我们自己设计了一种名为 STR 的程序设计语言，定义如下：

一个 STR 程序　::=<STR 指令序列> <,>

<STR 指令序列>　::=<STR 指令> | <STR 指令序列>

<STR 指令>　::=<字符串 1,字符串 2>

字符串 1　::=　字符串

字符串 2　::=　字符串 | 空串

字符串　::=　除","“< ”“> ”以外的任何可显示 ASC 字符所组成的字符序列

空串　::=　（不包含任何字符）

其中：：=　读作“定义为”，　| 读作“或”，表示或的关系。

例如下面所列的一个 STR 程序包含七条指令：

```
<aa,b>
<ba,a>
<bc,a>
<c,start>
<d,>
<b,finish>
<,>
```

STR 指令完成字符替换操作，字符串 2 替换字符串 1。

执行一个 STR 程序的流程如下：

1. 事先输入一个待处理的字符串，这里称之为原串；
2. 从 STR 程序开始处取第一条指令；
3. 若该指令为 <,> ，则退出 STR 程序，否则做 4；
4. 若该指令中的字符串 1 在原串中不出现，则按顺序取下一条指令，并转 3。否则做 5；
5. 将原串中的从左至右查找到的第二个（如有多个的话）字符串 1 替换成字符串 2，（注意　① 在这一步如有多个可替换处，仅替换第一个；② 替换后不留空格），被替换后的原串仍称原串。做完这一步后转 2。

例如　输入待处理字符串 abcabcd，用前面给出的包含七条指令的 STR 程序来处理，分步结果如下：

	abcabcd
使用 <bc,a>	aaabcd

使用 <aa,b>　　babcd
使用 <ba,a>　　abcd
使用 <bc,a>　　aad
使用 <aa,b>　　bd
使用 <d,>　　b
使用 <b,finish>　　finish
最后会遇到 <,> 退出 STR 程序。

现有三个任务需你来做：

任务 1：用你熟悉的程序设计语言编一个能够逐条解释执行任意一个 STR 程序的程序。STR 程序放在有 ".STR" 为后缀的文本文件中。作为待处理的字符串(原串)，用你所用的程序设计语言由键盘输入，要求将执行 STR 程序对原串进行处理的每一步都显示出来。格式为：

	原串
使用 <STR 指令>	变化后的原串
使用 <STR 指令>	变化后的原串
⋮	⋮

注意：任务 1 要求编写一个对于任何一个 STR 程序都适用的通用的解释执行程序。

任务 2：用 STR 语言编一个程序，完成如下功能：

对于形如

数字字符串 1 + 数字字符串 2 =

的字符串(视为原串，其中数字字符串 1 和数字字符串 2 分别对应两个 10 进制正整数，)用你所编制的 STR 程序对原串做替换，使得程序结束时的原串恰为两个 10 进制正整数相加的结果。

例如　1990 + 123 = 视为原串，而所编的 STR 程序名为 add10.str，应能对这个原串进行替换处理，最后使原串变为 2113，恰为两数相加结果。将 add10.str 文件保留起来，用任务 1 的程序来检验 add10.str 能否正确实现任意的 10 进制正整数加法运算。

任务 3：用你所熟悉的程序设计语言编写一个程序，该程序的功能为：如由键盘输入一个 2…16 的正整数 n，程序会输出一个名为 addn.str 文件，该文件是一个 STR 程序，这个程序能够如任务 2 那样，完成任何一个 n 进制正整数加法运算。

二、算法分析

本题要求设计一个 STR 语言，这个语言能够对任何形如：

数字字符串 1 + 数字字符串 2 =

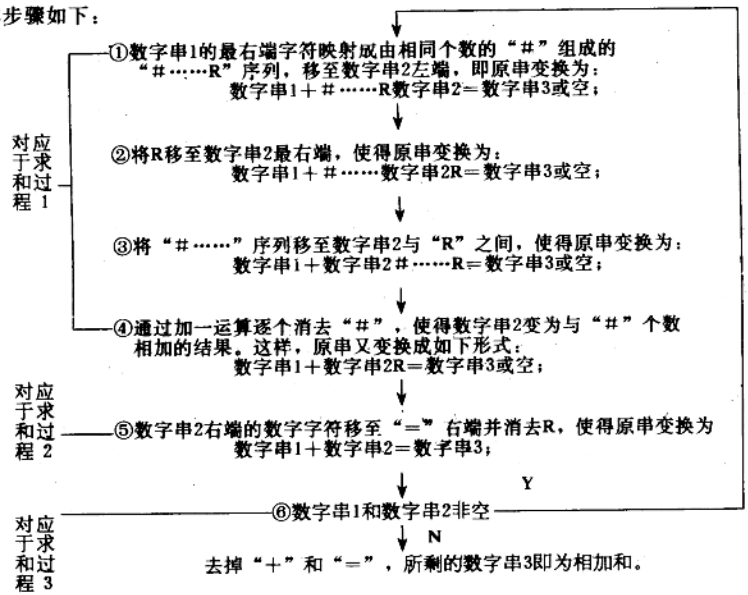
(数字字符串 1 和数字字符串 2 分别对应两个 n 进制正整数)

的字符串进行处理，使得处理结束后的原串恰为两数相加的结果。这里，特别值得提醒的是，解释执行的过程不像通常算术中的加法运算那样一蹴而就，而是反复使用 STR 的指令序列一次一次的对表达式串进行替换处理，最终达到求和目的的。那么，这个 STR 语言应该包含哪些指令呢？

首先，让我们来粗略地分析一下求和过程：

1. 将数字串 1 的最右端字符移出该串，并通过替换处理，使数字串 2 变为与该数相加的结果；
2. 数字串 2 的最右端字符移至等号右端，产生和的新一位数；
3. 若数字串 1 非空，转(1)；否则，将数字串 2 移至等号右端；并消去‘+’和‘=’号，所剩数字串即为和。

具体步骤如下：



其中“#”为数字映射，“R”为运算标志。

有了上述算法流程，我们就可以为每一运算步骤设计 STR 指令了。以十进制为例：

步骤①的指令为：

<0+,+R>,<1+,+#R>,…,<9+,+#########R>；

步骤②的指令为：

<R0,0R>,<R1,1R>,…,<R9,9R>

步骤③的指令为：

<#0,0#>,<#1,1#>,…,<#9,9#>

步骤④的指令为：

<0#,1>,<1#,2>,…,<8#,9>,<9#,#.0>,<#,1>,

<.,>；

其中<9#,#.0>,<#,1>是进位，<.,>消进位符号‘.’。

步骤⑤的指令为：

<0R=,=0>,<1R=,=1>,…,<9R=,=9>,<R=,=0>；

步骤⑥的指令为：<+,>,<=,>整个指令序列以<,>为结束标志。

例如　我们输入原串“17＋26＝”。用包含上述指令的 STR 程序来进行替换处理，分步结果如下：

input the original state:

	17+26=
<7+,+#######R>	1+#######R26=
<R2,2R>	1+#######2R6=
<R6,6R>	1+#######26R=
<#2,2#>	1+######2#6R=
<#2,2#>	1+#####2##6R=
<#2,2#>	1+####2###6R=
<#2,2#>	1+###2####6R=
<#2,2#>	1+##2#####6R=
<#2,2#>	1+#2######6R=
<#2,2#>	1+2#######6R=
<#6,6#>	1+2######6#R=
<#6,6#>	1+2#####6##R=
<#6,6#>	1+2####6###R=
<#6,6#>	1+2###6####R=
<#6,6#>	1+2##6#####R=
<#6,3#>	1+2#6######R=
<#6,6#>	1+26#######R=
<6#,7>	1+27######R=
<7#,8>	1+28#####R=
<8#,9>	1+29####R=
<9#,#.0>	1+2#.0###R=
<0#,1>	1+2#.1##R=
<1#,2>	1+2#.2#R=
<2#,3>	1+3.2#R=
<2#,3>	1+3.3R=
<.,>	1+33R=
<3R=,=3>	1+3=3
<1+,+#R>	+#R3=3
<R3,3R>	+#3R=3
<#3,3#>	+3#R=3
<3#,4>	+4R=3
<4R=,=4>	+=43
<+,>	=43
<=,>	43

三、程序的求精分析

第一层

一、初始化

1.1 定义和说明

```
begin
  write ('n= ');
```

二、生成 addn.str 文件

```
  str (digit,filename);
    { digit 转化为字串 filename }
  filename ← 'add' + filename + '.str';
```

三、n 进制正整数相加

```
repeat
  readln (state);
    { 读入表达式串 }
```

```
readln (digit);
{ 读入进制数 }
                    { 生成文件名 }
                  assign (f,filename);
                    { 设置和打开逻辑文件 f }
                  rewrite (f);
                    { 写文件准备 }
                2.1   make_new_text;
                    { 生成所有指令存入文件 f }
                  close (f);
                    { 关闭文件 }
```

```
3.1   finding_process
  (state); { 解释执行 }
  writeln ('another
    problem? (Y/N)');
      readln (more);
  until  (more='N')
       or (more='n');
  end.
```

第二层

1. 求精　1.1 —— 定义和说明

```
const max_stack  =  150;        { 最多指令数 }
var
  f          : text;            { 文件变量 }
  filename   : string[20];      { 文件名串 }
  digit      : integer;         { 进制数 }
  state      : string;          { 原串 }
  more       : char;
  { more = 'N',程序结束;否则继续解释执行下一原串 }
```

至于过程和函数说明，则在下面逐一给出。

2. 求精　2.1 —— make_new_text 的过程说明

```
procedure make_new_text;
var
  digit_code : array [0..20] of char;
   { digit_code[i] — digit 进制的数字序列中第 i 个数字字符 (0≤i≤20);
  i, j : integer; { 循环变量 }
begin
  for i:=0 to 9 do digit_code[i]← chr (ord ('0')+i);
   { digit_code[0..9]='0'..'9' }
  if digit>10 then
   for i:=10 to digit-1 do digit_code[i] ← chr (ord ('a')+i-10);
        { digit_code[10..digit-1] = 'a'..[chr (ord ('a')+digit-11] }
  for i:=0 to digit-1 do writeln (f,'<R',digit_code[i],',',
                                  digit_code[i],'R>');
        { 生成指令序列<Ri,iR>并存入 f 文件 (0≤i≤digit-1) }
  for i:=0 to digit-1 do writeln (f,'<#',digit_code[i],',',
                                  digit_code[i],'#>');
        { 生成指令序列<#i,i#>并存入 f 文件 (0≤i≤digit-1) }
  for i:=0 to digit- 2 do writeln (f,'<',digit_code[i],'#,',
                                  digit_code[i+1],'>');
        { 生成指令序列<i#,i+1>并存入 f 文件 (0≤i≤digit-2) }
  writeln (f,'<',digit_code[digit-1],'#,#.0>');
```

```
    writeln (f,'<#,1>');
    writeln (f,'<.,>');
          { 生成指令< (digit-1)#,#.0>,<#1,1>,<.,>并存入 f 文件 }
    for i:=0 to digit-1 do writeln (f,'<',digit_code[i],'R=,=',
                                    digit_code[i],'>');
          { 生成指令序列<iR=,=i>并存入 f 文件 (0≤i≤digit-1) }
    writeln (f,'<R=,=0>');
    writeln (f,'<0+,+R>');
          { 生成指令序列 <R=,=0> 和 <0+,+R> 并存入 f 文件 }
    for i:=1 to digit-1 do
     begin
      write (f,'<',digit_code[i],'+,+');
      for j:=1 to i do write (f,'#');
      writeln (f,'R>');
     end; { for }
          { 生成指令序列 <i+,+#……#R> 并存入 f 文件 (0≤i≤digit-1)
                                    i个'#'序列                    }
    writeln (f,'<+,>');
    writeln (f,'<=,>');
    writeln (f,'<,>');
          { 生成指令 <+,>、<=,>、<,> 并存入 f 文件 }
    end; { make_new_text }
```

3. 求精 3.1 —— finding_process (state)的过程说明

```
  procedure finding_process (var first_s : string);
  var s_str : string;                  { 当前指令 }
      stack : array[1..max_stack]of string[30];   { 暂存所有 STR 指令 }
        i : integer;                   { 指令序号 }
    position : integer;   { 原串 first_s 中被替换串 (即<a,b>中的 a) 的首位置 }
  com-pos : integer;                   { 当前指令中 ',' 的位置 }
  begin
    writeln (' ':20 , first_s);        { 打印表达式串 }
    reset (f); i ← 0;                  { 读文件准备 }
    repeat                             { 从 f 文件中读所有指令至 stack 数组 }
     i ← i + 1;
     readln (f,stack[i])
    until stack[i]= '<,>';
    i ← 1; s_str ← stack[i];           { 从第一条指令开始解释执行 }
    while s_str <> '<,>' do            { 若还剩指令未解释执行,则继续下述过程 }
     begin
      com-pos ← pos (',',s_str);{ 求当前指令<a,b>中',' 的位置 com_pos}
      if length (first_s) >= com_pos-2 then
       { 若原串长度大于等于被替换串 a 的长度 }
```

```
        begin
          position ← pos (copy (s_str,2,com_pos－2),first_s);
          { 在原串 first_s 中求被替换串 a 的首位置 position }
          if position<>0 then      { 若 first_s 串中存在 a 子串 }
           begin
           delete (first_s, position, com_pos－2);{ 在 first_s 串中删去子串 a}
          if com_pos＋1<length (s_str) then   { 在当前指令中,替换串 b 存在}
          insert(copy(s_str,com_pos＋1,
                          length(s_str)－com_pos－1),first_s,position);
           { 在 first_s 的 position 位置开始,插入替换子串 b }
          writeln (s_str:15,' ':5,first_s);
           { 打印指令和被替换后的原串 }
          i ← 0    { 指令序号初始化，表示下一替换从头开始 }
          end      { then }
          end;     { then }
          i ← i＋1; s_str ← stack[i];    { 搜索下一条 STR 指令 }
        end;     { while }
      close (f);    { 关闭文件 }
end;    { find_process }
```

§1.2 子串匹配

一、试　题

一种表达式的定义如下：

元素　:=　一个或多个小写英文字母组成的序列
包项　:=　元素 | (表达式)
连项　:=　包项 | 包项 * | 包项#
或项　:=　连项 | 连项 ＋ 或项
表达式　:=　或项 | 或项，表达式

其中，“:=”表示“定义为”；“|”表示“或”关系；表达式是字符串的集合，

表达式1 =	表达式1表示的集合元素
(表达式2)	与表达式2相同
表达式2 *	由表达式2的元素出现零次或任意多次而得
表达式2#	由表达式2的元素出现一次或任意多次而得
表达式2 ＋ 表达式3	由表达式2的任一个元素与表达式3的任一个元素顺接而得
表达式2，表达式3	由表达式2或表达式3的任一个元素

例：　若表达式为　((c) * ＋(ae)#) * ＋f

则　cccccaeaeaef

aeaeaeaeaeaeaeaeaef

aef

aecccaeccaef

f

都是该表达式所表示的集合中的元素。

要求编程完成这种表达式的匹配功能。由键盘输入一个表达式，并在一个名为 LH. TXT 的文本文件中查找第一次匹配到的字串，并输出该子串及其位置。

该文件可认为是一个很长的字符串，所谓子串位置指该子串第一个字符位置。

例：　若 LH. TXT 内容为

abcacbbccaeaefdlkfjsdalljghiersdag

输入表达式为　((c)*+(ae)#)*+f

则输出子串为　ccaeaef

子串位置为　8

二、算法分析

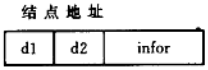
由题意可以看出，一个表达式由小写英文字母、运算符'*'、'#'、'+'、','和括号组成。从这个表达式出发，反复使用表达式的定义对运算符进行推导和替换，就可以得到这个表达式所表示的集合中的每个元素，这些元素是由一个或多个小写英文字母组成的字符串。而题意仅要求查找其中的一个元素，即第一次与 LH. TXT 中的某个子串匹配的元素，并指出子串位置。

我们用结点来表征解题过程中每一步的特点及其关联方式。那么对于该题来说，应如何定义结点的数据结构呢?这还得从表达式的定义分析起。

设 S1，S2，S 是表达式。表达式 'S1，S2' 表示的集合元素既可以是 S1 的任一个元素，也可以是 S2 的任一个元素；表达式 'S*' 表示的集合元素既可以空，也可以由任意多个 S 的元素组成。表达式 'S#' 表示的集合元素既可以是一个 S 元素，也可以由任意多个 S 的元素组成。由此可见，每一个结点最多应有两个后继状态。从这一点出发，我们对结点设计了如下数据结构：

```
Type
  NodePtr = ↑ Node;                { 结点的地址指针类型 }
  Node = Record                    { 结点类型：
           D1, D2  :NodePtr;       指向两个后继结点的地址指针
           Infor   :String[10];    存贮元素值的信息域 }
         End;
```

有了结点形式后，如何使用表达式定义，对运算符进行推导和替换呢?我们先从最简单的情况开始分析，即设表达式 S1，S2，S 由仅含小写英文字母的元素组成。结点的图形表示为：



D1，D2 指针域中的 ^ 表示为空，即 nil.

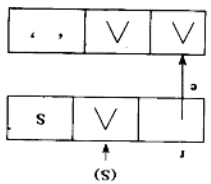


图1.2-2

从 r 结点入口，得到表达式值 S；

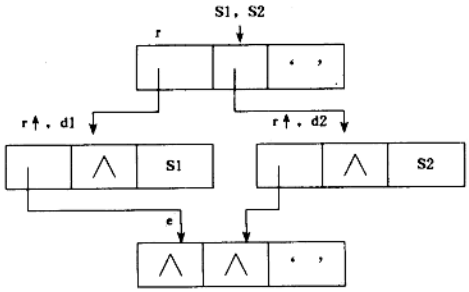


图1.2－3

从r结点入口，通过d1指针得到表达式值S1，或者通过d2指针得到表达式值S2.

从r结点入口，通过d1指针和r↑.d1结点的d1指针得到表达式值S1，S2；

从r结点入口，即可以通过d1指针出口，使表达式值为空，也可以沿r，r↑.d2、b结点组成的环环绕多次，然后由r的d1指针出口，得到多个S元素；

从r结点入口，即可以经r↑.d1、b↑.d1指针出口，使表达式值为S，也可以沿r，b组成的环环绕多次，然后出口，得到多个S元素；

上述各个简单图形都有一个确定的入口结点r、一个确定的出口结点e，即每输入一个表达式，都会有一个表达式集合中的元素输出。

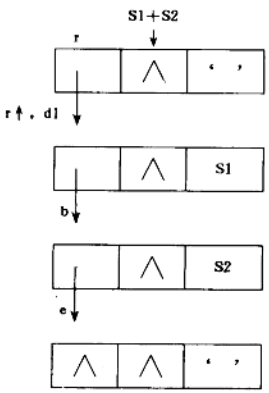


图1.2－4

但问题是，如果表达式S1，S2，S本身含运算符，则求解过程将构成一个更为复杂的、带环的有向图，这个图也得有一个入口、一个出口。我们设计一个递归过程make(r,e,s)来构造这个图。其中参数r为指向入口结点的地址指针，e为指向出口结点的地址指针，s为表达式串。根据上述思想和运算符由()→*、#→+→，的优先顺序，make(r,e,s)可递归定义如下：

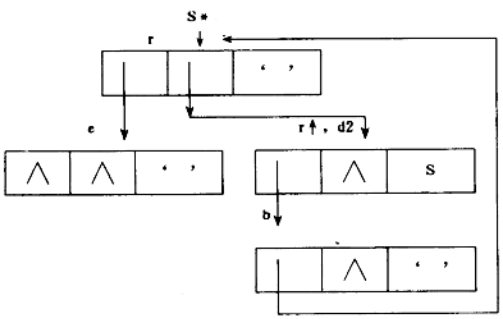


图1.2－5

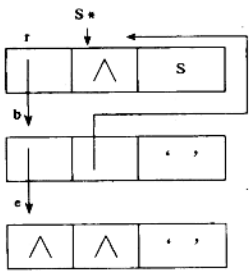


图1.2－6

```
make(r,e,s);                                  当表达式为'(s)'形式时;

r↑.infor ← '';                                当表达式为's1,s2'形式时;
生成r的子结点r↑.d1和r↑.d2;
make (r↑.d1,e,s1);
make (r↑.d2,e,s2);

r↑.d2 ← nil;  r↑.infor ← '';                  当表达式为's1+s2'形式时;
生成r的子结点r↑.d1和中间结点b;
make (r↑.d1,b,s1);
make (b,e,s2);

r↑.d1 ← e;  r↑.infor ← '';                    当表达式为's*'形式时;
生成r的子结点r↑.d2和其子结点b;
b↑.d1 ← r; b↑.d2 ← nil;
b↑.infor ← '';
make (r↑.d2,b,s);

生成r的子结点b;                               当表达式为's#'形式时;
b↑.d1 ← e; b↑.d2 ← r;
b↑.infor ← '';
make (r,b,s);

r↑.d1 ← e; R↑.d2 ← nil;                       当表达式S为不含运算符的英文
r↑.infor ← s;                                 字母元素时,到达递归边界。
```

过程 make(r,e,s) 调用返回后，置出口结点 e

```
e.d1 ← nil;
e.d2 ← nil;
e.infor ← nil;
```

例如现有表达式 "(a#+b*),c"，我们通过 make 过程求这个表达式所表示的集合中的每个元素，具体过程如下(见图1.2-7)：

由此可见，上述递归定义是确定的。因为每递归一次，运算符都被消去一个，并替换成相应的字母序列。所以，递归若干次后，一定会求出表达式S所表示的集合的所有元素。只要输入的表达式格式正确，都可以由递归边界明确定值。接下去的问题是如何在这个图中搜索匹配子串。同样地，我们设计一个递归函数 find(r,p,s)，函数值的类型为布尔型。该函数从结点 r 出发搜索由 make 过程构造的图。如果发现有与原串(LH.TXT 文本文件)第 p 个位置开始的子串匹配的表达式元素 str(若存在多个匹配的表达式元素，则函数在第一次匹配后便返回)，则返回 true，否则返回 false。

find(r,p,s)函数递归定义如下：

例如，LH.TXT 文件的内容为'cbac'。我们通过 find 函数，在 make(@start, @stop, '(a#+b*),c')构造的带环有向图中搜索匹配子串，具体过程如下(见图1.2-8)：

最后输出子串'c'，子串位置1。

find 函数的递归定义也是确定的，每递归一次，函数会沿当前 r 结点的 d1指针和 d2 指针搜索下去，递归若干次后，一定会到达上述三个递归边界之一。如果我们想确定原串第 i 个位置开始的子串是否与表达式所表示的集合中的任一元素匹配，只要调用 find(@start, i, '')，从入口结点@start 出发，按 find 函数的递归定义搜索图，便能得出问题的答案。

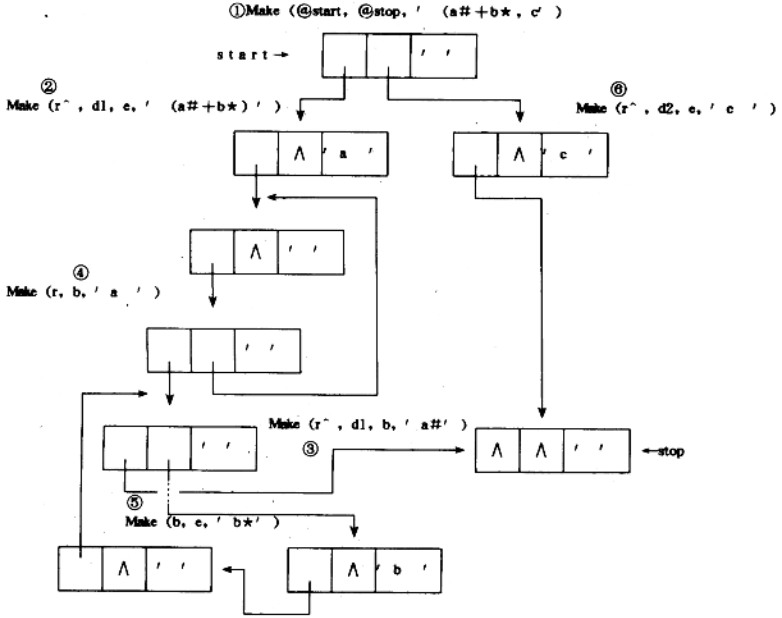


图1.2－7

注：make 上方的数字表示递归顺序

find (r,p,s)＝

- true　且 str ← s；　若r＝@stop，即到达出口，求出了子串 str 和子串位置 p；
- true　若 (r↑.infor＝'')and(find(r↑.d1,p,s) ＝true)or(find(r↑.d2,p,s)＝true))时；
- true　若 (r↑.infor<>'')and(原串 p 位置开始的子串与 r↑.infor 匹配)and((find(r↑.d1, p＋length(r↑.infor), s＋r↑.infor)＝true)or(find(r↑.d2, p＋length(r↑.infor), s＋r↑.infor)＝true)；
- false　若 r＝nil，无法再匹配下去；
- false　若r↑.infor<>''但原串 p 的位置开始的子串与 r↑.infor 不匹配，匹配失败；

——→递归边界

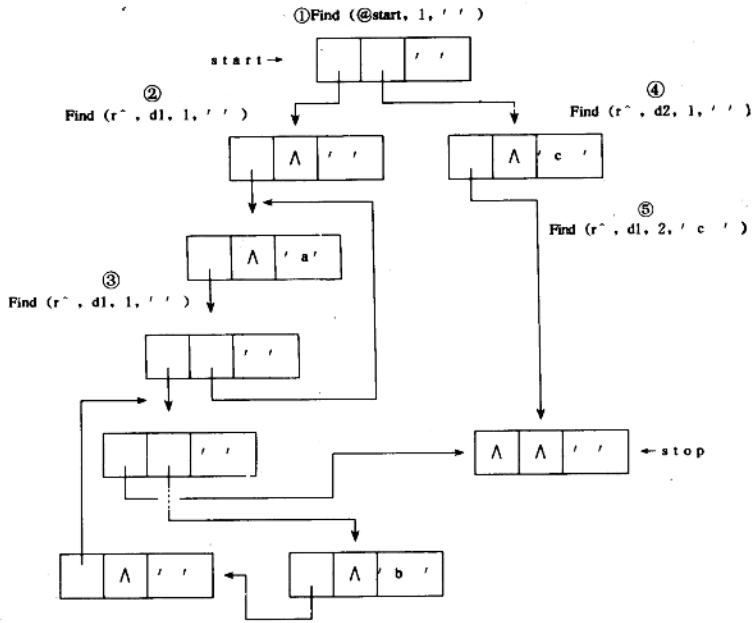


图1.2－8

注：find 上方的数字表示递归顺序

三、程序的求精分析

第一层

```
一、初始化
1.1  定义和说明;
     begin
  1.2  Getstring;
{从 LH.TXT 中读入原串}
  write('expression:');
  readln(str);
  {输入表达式串 str};
```

```
二、构造图
2.1   make (@start, @stop,
str);
  {构造图}
    stop.d1 ← nil;
    stop.d2 ← nil;
    stop.infor ← '';
  {置出口结点的指针域空,
        信息域空}
```

```
三、搜索匹配子串
  i ← 1; {从原串第一个位置
  开始搜索匹配子串}
  while list[i]<>#0 do
{只要原串未搜索完,则进行如下处理}
3.1  if find(@start,i,'') then
  {若原串第 i 个位置开始的子串与
    表达式集合中的某元素匹配}
      begin
        writeln('sub string:'
        ,str);
        {打印匹配子串}
  if length(str)=0 then i ← 0;
```

```
        { 若匹配子串为空,
          则置位置值为0 }
        writeln ('position:',i);
          { 输出位置值 }
        readln;
        halt(0); { 退出程序 }
      end { then }
      else i ← i+1; { 匹配失败,则
        从原串下一位置开始继续匹配 }
      writeln('No sub-string found');
      { 原串搜索完毕,则输出匹配失败信息 }
      end. { main }
```

第二层

1. 求精 1.1 ——定义和说明

```
Const
  max=7000;                          { 原串最多字符数 }
Type
  nodeptr=↑node;                     { 结点地址的指针类型 }
  node=record                        结点类型:
        d1,d2:nodeper;               指向两个后继结点的地址指针
        infor:string[10];            存储元素值 }
       end;
Var
  list          : array[1..max] of char;   { 原串 }
  str           : string;                  { 表达式串或匹配子串 }
  start, stop : node;                      { 图的入口结点和出口结点 }
  i             : integer;                 { list 数组的位置指针 }
  can           : boolean;                 { 括号配对标志 }
```

至于函数和过程说明,则在下面逐一给出。

2. 求精 1.2 —— Getstring 的过程说明

```
Procedure Getstring;
var f : text;                    { 文本文件 }
begin
assign(f,'c:\pascal\data\lh.txt'); { 将 LH.TXT 的文件路径赋给文件变量 f }
reset(f);                        { 读准备 }
i ← 0;                           { 指针初始化 }
while not eoln(f) do             { 将文件中的每一字符依次读入 list 数组 }
 begin
    i ← i+1; read(f,list[i])
 end;                            { while }
list[i+1] ← #0;                  { 置原串结束标志#0 }
close(f);                        { 关闭文件 }
```

```
  end;                                    { Getstring }
```

3. 求精 2.1—— make(@start,@stop,str)的过程说明

```
Procedure make(r,e:nodeptr; s:string);
{ r — 入口结点指针; e — 出口结点指针; s — 表达式串 }
var
  p : integer; { 表达式的位置指针 }
  b : nodeptr; { 中间结点指针 }
begin
  p := MyPos ('',copy(s,2,length(s)-2)); { 对括号配对标志 can 赋值 }
  if (s[1]='(') and (s[length(s)]=')') and (can)
  { 若表达式为(s)形式且括号配对,则递归处理 S 表达式,退出 }
  then begin
    make (r,e,copy(s,2,length(s)-2));
    exit;
  end;                    { then }
2.1.1  p ← mypos(',',s); { 查找表达式 S 中位于括号外的',' 的位置值 }
  if p<>0 then { 若表达式 S 为's1,s2'形式 }
  begin
    r↑.infor ← '';                        { 置 r 结点的信息域为空 }
    new(r↑.d1);new(r↑.d2);                { 生成 r 的两个后继结点 }
    make(r↑.d1,e,copy(s,1,p-1));          { 沿 r 的 d1指针方向递归处理 s1 }
    make(r↑.d2,e,copy(s,p+1,length(s)-p));
    { 沿 r 的 d2指针方向递归处理 s2 }
    exit;                          { 退出 }
  end;                             { then }
  p ← mypos('+',s);                { 查找表达式 S 中位于括号外的'+'的位置值 }
  if p<>0 then                     { 若表达式 S 为's1+s2'形式 }
  begin
    r↑.infor ← '';r↑.d2 ← nil;          { 置 r 的信息域为空、d2指针空 }
    new(r↑.d1);new(b);                  { 生成 r 的后继结点 r.d1和 b 结点 }
    make(r↑.d1,b,copy(s,1,p-1));
    { 沿 r 的 d1指针方向递归处理 s1,由 b 结点出口 }
    make(b,e,copy(s,p+1,length(s)-p));
    { 从 b 结点出发,递归处理表达式 S2 }
    exit; { 退出 }
  end; { then }
  if s[length(s)]='*' then { 若表达式 S 为's1*'形式 }
  begin
    r↑.infor ← '';r↑.d1 ← e; { r 的信息域空、d1指针指向出口 }
    new(r↑.d2);new(b); { 生成 r 的后继结点 r↑.d2和中间结点 b }
    b↑.d1 ← r;b↑.d2 ← nil;b↑.infor ← '';
    { b 的 d1 指针返回 r,d2指针和信息域为空 }
    make(r↑.d2,b,copy(s,1,length(s)-1);
    { 从 r↑.d2入口递归处理 S1,处理结束后指向 b 结点 }
```

```
  exit;{ 退出 }
end;{ then }
if s[length(s)]='#' then { 若表达式 S 为's1#'形式 }
  begin
    new(b);{ 生成中间结点 }
    b↑.d1 ← e;b↑.d2 ← r;b↑.infor ← '';
    { 可沿 b 的 d1指针出口,也可沿 b 的 d2指针返回 r,b 的信息域为空 }
    make(r,b,copy(s,1,length(s)-1));
    { 从 r 结点出发递归处理 S1,处理结束后指向 b }
    exit;                { 退出 }
  end;                   { then }
r↑.d1 ← e;
{ 表达式 S 为字母序列,到达了递归边界,由 r 的 d1指针出口 }
r↑.d2 ← nil;             { r 的 d2指针置空 }
r↑.infor ← s;            { 字母序列 S 存入 r 的信息域 }
end;                     { make }
```

4. 求精 3.1 —— find(@start,i,'')的函数说明

```
function find (r:nodeptr;p:integer;s:string):boolean;
{ r — 当前被搜索的结点指针,调用初始时设为图的入口指针@start;
p — 子串在 list 数组的开始位置;
s — 求得的匹配子串,调用初始时设为'' }
begin
  find ← false;
  if r=nil then exit;{ 当前搜索的结点地址指针为空,失败退出 }
  if r=@stop then
  { 搜索到出口结点指针则成功,匹配子串 S 赋 str,返回 true }
  begin
    find ← true;
    str ← s;
    exit;
  end;{ then }
  if r↑.infor='' then
{ 若 r 的信息域为空,则从原串 list 的 p 位置开始,分别沿 r 的 d1指针方向和 d2指针方向递归搜索。若其
  中一个方向匹配成功,则返回 true }
  begin
    if find(r↑.d1,p,s)or find(r↑.d2,p,s)
    then find ← true;
  end                    { then }
else
  begin                  { 若 r 的信息域非空,则分别按下述两种情况处理 }
    for i := 1 to length(r↑.infor) do
{若 list[p]开始的长度为 length(r↑.infor)的子串信息与 r↑.infor 不匹配,则失败退出 }
    if list[p+i-1]<>r↑.infor[i] then exit;
    if find (r↑.d1,p+length(r↑.infor),s+r↑.infor)
```

```
      or find(r↑.d2,p+length(r↑.infor),s+r↑.infor)
  {否则,从list[p+length(r↑.infor)]位置开始沿r的d1指针方向和d2指针方向递归搜索,若其中一个
   方向匹配成功,则返回true}
      then find ← true;
    end;                    {else}
  end;                      {find}
```

第三层

最后求精 2.1.1 —— mypos(',',s)的函数说明

```
Function mypos (sub:char; S:string):integer;
var
    b : integer;          {表达式串S中括号的配对情况,b=0括号配对}
    i : integer;          {S串的位置指针}
  begin
    b ← 0;mypos ← 0;can ← true;
    for i ← 1 to length(s) do
    if s[i]='(' then b ← b+1
    else if s[i]=')' then begin
    b ← b-1;
    if b<0 then begin
    {若括号不配对,can置false,退出}
      can ← false;
      exit;
    end
  end
  else if (s[i]=sub)and (b=0) then begin
    mypos ← i;
    exit;
  end;{then}
end;{mypos}
```

输入表达式串S和运算符sub,若在S串中sub出现在括号外,则返回其位置值。若在S串中sub出现在括号内或括号不配对,则返回0

§1.3 称　　重

一、试　　题

有6个物体分别用1－6编号,其中5个重量相同。现在有一架台秤,1次能称出放在上面的若干物体的总重。要求编一个程序,让计算机找出一种称法,只要称3次就可以知道每个物体的重量。具体使用时由操作者默想6个物体各自的重量,而每次由计算机用编号提问若干物体的总重量,操作者由键盘回答。如此3次后程序应能输出每个物体的重量。

二、算法分析

本题的关键是寻求一种可行的称重方案(即确定每次分别称哪几个物体的总重，如此称3次后就可得知每个物体的重量)。该题本身不算太复杂，只要经周密的逻辑推理，便能直接求得问题的解。但我们不要求选手这样做，这是因为

1. 选手要在限定的竞赛时间内完成这一推理过程，几乎是不可能的；

2. 计算机解题与数学解题不能等同，不能像数学竞赛那样仅强求于人为的数学因素、直截了当的依据规则在纸面上推导求解，而是要求选手设计一种能在计算机上实施的算法，即一一列举各种可能的称重方案，用检验条件判定哪些方案称3次就能得知每个物体的重量，哪些方案是无法办到的。

那么，这个判定称重方案所用的检验条件究竟是什么呢？

我们知道，称重次数限定为3次，分析每次称重结果的判断条件不外乎有以下6个：

1. 第1、2次称重的每个物体重量相同；
2. 第1次称重的每个物体重量相同；
3. 第2、3次称重的每个物体重量相同；
4. 第2次称重的每个物体重量相同；
5. 第3次称重的每个物体重量相同；
6. 连续3次称重中都含那个重量不同的物体；

为了说明简便，下面我们将重量不同的那个物体称作“次品”，其它5个重量相同的物体统称“正品”。

为了得出一个完美的称重方案，必须连续成功地判断六次。每判断一次，就在次品所能产生的集合范围内除去该判断条件所表明的那些正品。若仅剩下一个元素，那么这个元素就是次品的一个可能编号。我们在原集合范围里除去这个元素后继续下一个判断……直至通过六次判断，将这个次品的六个可能编号全部求出，次品可能产生的集合范围变空为止。若在某次判断中得出不止一个的次品编号，则说明该称重方案是无用的，应予剔除，重新枚举下一称重方案。

具体的判断过程如下(见图1.3－1)：