国际国内青少年信息学（暨计算机）竞赛试题解析

（1992～1993）

電子工業出版社

国际、国内青少年信息学(暨计算机)竞赛试题解析(92～93)

吴文虎　王建德　编著

电子工业出版社

（京）新登字 055 号

内 容 提 要

本书作者是国际奥林匹克中国队总教练兼领队和培训我国选手的优秀教师。本书是他们辛勤培育我国竞赛选手的经验总结。主要内容有：第四届、第五届国际奥林匹克信息学竞赛中国组队赛试题分析，第四届、第五届国际奥林匹克信息学竞赛试题分析，第九届、第十届全国奥林匹克信息学竞赛试题分析；此外，还附有基础试题、图论试题、组合数学应用试题、搜索试题和综合试题等。

本书适合中学计算机教师和广大中学生阅读，也可作为计算机竞赛选手培训教材。

国际、国内青少年信息学（暨计算机）竞赛试题解析（92～93）

吴文虎　王建德　编著

责任编辑：徐轲

*

电子工业出版社出版

北京市海淀区万寿路 173 信箱（100036）

电子工业出版社发行　各地新华书店经售

北京天利电子出版技术公司排版

北京市顺义县天竺颖华印刷厂印刷

*

开本：787×1092 毫米 1/16　印张：12.25　字数：360 千字

1994 年 6 月第 1 版　1994 年 6 月第 1 次印刷

印数：0,001－4,000 册　定价：14.00 元

ISBN 7－5053－2392－X/TP・684

序　言

国际信息学奥林匹克(International Olympiad in Informatics)是计算机知识在世界范围青少年普及的产物,从1989年到现在已成功地举行了五届竞赛。在这五届竞赛中,中国队共派出15名选手(其中一名选手连续参加两届,另一名选手连续参加三届)夺得金牌7块,银牌5块,铜牌6块,在第三届和第四届获总分第一。

举办国际信息学奥林匹克的目的是,通过竞赛形式对有才华的青少年起到激励作用,促其能力得以发展;让青少年彼此建立联系,推动知识与经验的交流,促进合作与理解;宣传新兴学科信息学,给学校这一类课程增加动力,启发新的思路;建立教育工作者与专家档次上的国际联系,推进学术思想交流。

从学科国际奥林匹克来看,只有信息学中国是首届派团参赛的。其中一个重要原因就是计算机的普及受到了党和政府的重视,得到了老师、家长和社会各界的支持,许多有识之士认识到"计算机的普及要从娃娃做起"的战略意义,并为之操劳;科学技术是第一生产力,未来人才的全面素质,包括科学素养,是发展科学技术,增强综合国力的重要因素。

电子计算机是现代科技的基础,它的出现和发展,把社会生产力水平提到前所未有的高度,开创了一个技术革命的新时代。计算,跟语言一样,是人类社会每时每刻都离不开的工具。计算机可帮助人类进行复杂计算。以往历史上的技术革命,只能起到创造或改进工具,用机器代替人的体力的作用;而计算机则能把人从重复性的或有固定程式的脑力劳动中解放出来,使自己的智能获得空前的发展。普及计算机教育是时代发展的需要。在只有算盘的时代,学生要学珠算;在有计算尺的时候,要学拉算尺;出现了电脑,要学用计算机,这本来是顺理成章的事。但这样理解却远远不够。因为计算机远非一般的计算工具,它是"人类通用智力工具",它在开发人类智能方面起着无与伦比的作用。著名计算机科学家,图灵奖获得者,美国斯坦福大学教授G.伏赛斯在〈What to do tell the computer scientist comes〉一文中曾预言:计算机科学将是继自然语言、数学之后,而成为第三位的,对人的一生都有大的用途的"通用智力工具"。用还是不用智力工具,对人的智能的发挥与发展肯定大不一样。正因为如此,计算机与基础教育相结合已成为当今世界的大趋势。谁不重视计算机教育,谁就会在人才的激烈竞争中败下阵来,现在必须将有关计算机的基本知识和应用计算机的能力纳入到学生必备的知识结构中。谁比较快地认识到这一点并真的付诸实践,谁就能取得主动。国家教委柳斌副主任在论述"为什么要重视计算机教育"时说:"经验证明,计算机技术越是高度发展,计算机人才就越重要,计算机教育就越重要。只有培养一批又一批掌握现代化已经成熟的电子计算机技术的人才,并不断发展和提高我国的计算机技术水平,我们才能加速我国走向现代化,走向世界,走向未来的历史进程。"

普及计算机教育要从娃娃做起。对青少年，从学一门简单的计算机语言入手来了解计算机，粗懂计算机是怎么工作的，它能够帮助我们做什么，可能是一条捷径。通过学习程序设计的思路与方法，还可以学到现代的、科学而高效的思维方式，提高逻辑思维、做规划、抽象化、形式化描述问题和科学计算以及分析问题与解决问题的能力，从已经学过计算机知识的青少年成长情况看，学与不学大不一样。特别是我们看到参加各级计算机奥林匹克竞赛的选手，到了大学阶段从学习能力上显示出明显的优势。

普及计算机要有层次意识，分层次符合因材施教原则。有些孩子课堂教学吃不饱，希望吃点"小灶"，希望学习更高层次的编程解题思路、方法与技巧，而且更盼望自己也能在全国大赛或世界大赛上一显身手。王建德老师和我将92、93两年国际、国内青少年信息学竞赛的题目做了归纳总结，写成了这本课外读物，供中学师生阅读参考。这里必须声明：要真的看懂有一定难度的试题，首先要有知难而进的勇气；第二，计算机不上机实践是学不会的，理论联系实际很重要；其次，书上的思路与方法，仅仅是抛砖引玉，也许你的方法更好，我们当然高兴。引发创造精神是我们写这本书的初衷。

中国计算机学会普及委员会主任
国际信息学奥林匹克中国队总教练
北京计算机奥林匹克学校校长
清华大学计算机科学与技术系教授
吴文虎

一九九四年二月于清华园

目　录

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 =

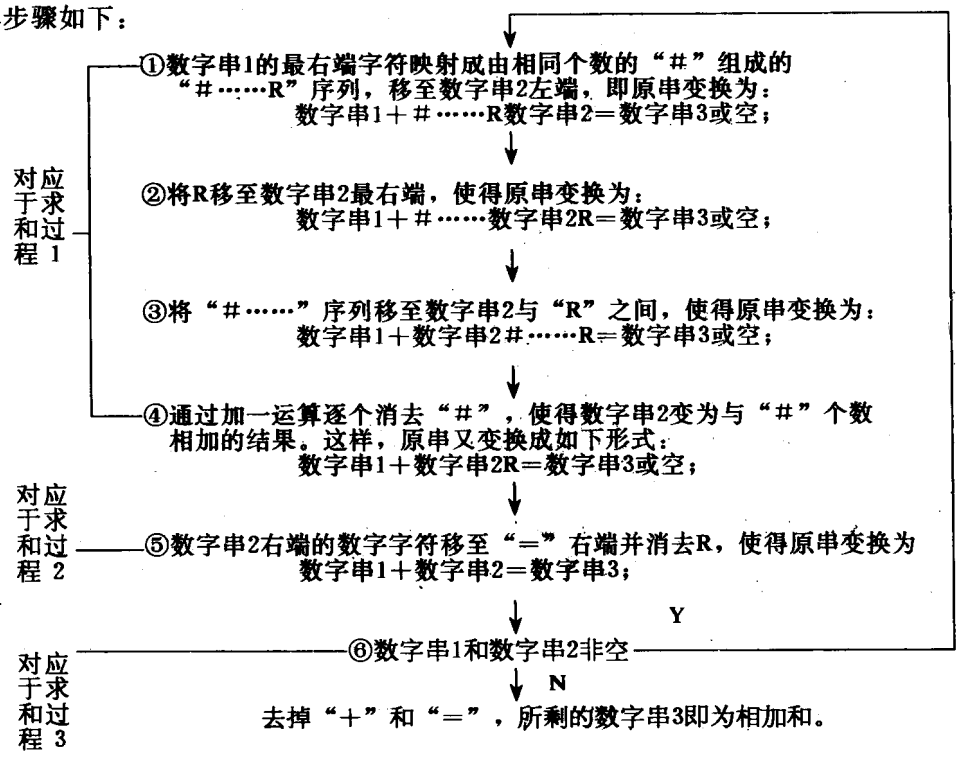
(数字字符串 1 和数字字符串 2 分别对应两个 n 进制正整数)

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=
<#6,6#>	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：

结点地址

d1	d2	infor

D1，D2 指针域中的 ^ 表示为空，即 nil.

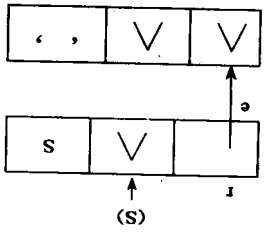


图1. 2－2

从 r 结点入口，得到表达式值 S；

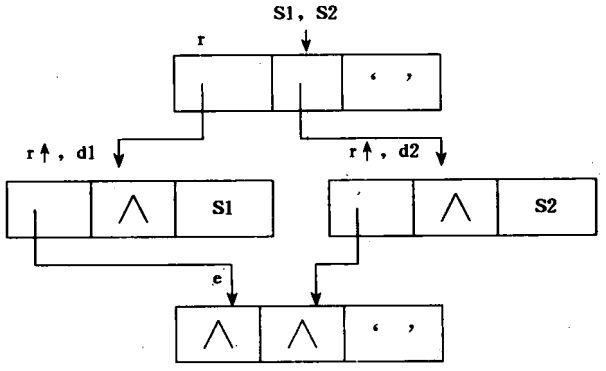


图1.2－3

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。

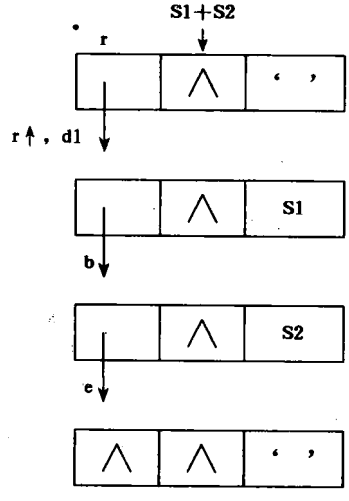


图1.2－4

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：

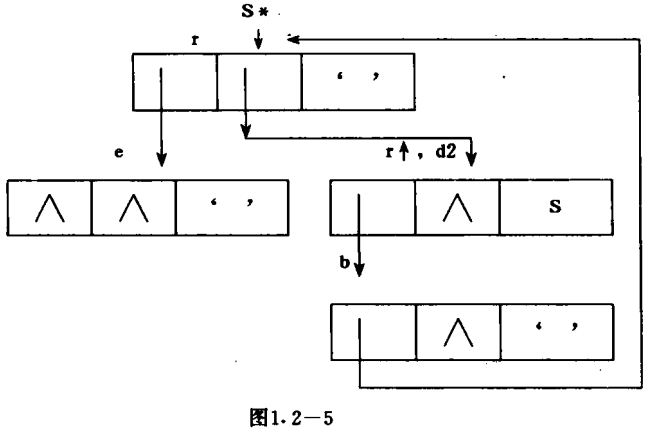


图1.2－5

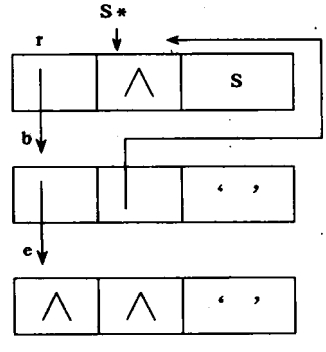


图1.2－6

```
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。

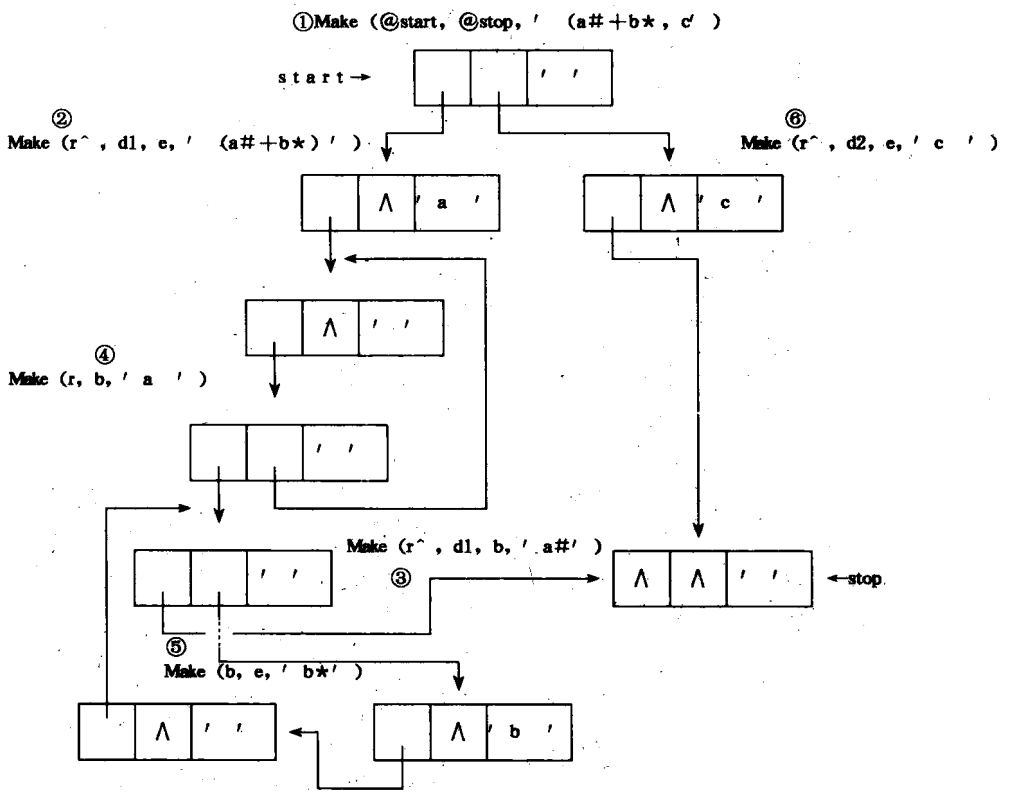


图1.2—7

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