Program Promenade

連載

木の図示

寺田　実（電気通信大学情報通信工学科）
terada@ice.uec.ac.jp

■問題

今回の問題は，1998 年 11 月に早稲田大学で行われたアジア地区予選の問題 G, "BUT We Need a Diagram" である．問題文は以下から入手できる．

`http://icpc.baylor.edu/past/icpc99/Regionals/Tokyo98/node7.html`

BUT というのは問題文によれば "Binary and/or Unary Tree" のことで，枝の数が 0, 1, 2 に制限された木構造である．その木構造を平面上に図示する，というのが問題の骨子である．ただし，図示するにあたってはいくつかの制限事項があり，それらを満たした上でスペースを最小としなければいけない．木構造の読み込みと図示という記号処理的な問題であるが，よくある処理手法だけではうまくいかないところがなかなか面白い問題である．

問題は，式として記述された BUT を構文解析によって読み込み，スペースを最小となるように各節点の配置を計算し，その結果を図による表現として出力するものである．各表現の詳細を述べる前に，問題文にある入出力の例をあげておく．

```
式表現（入力）:
a(B(c(D),E),f(g(H,i)))
図表現（出力）:
D  H i
-  ---
c E g
--- -
 B  f
 ----
  a
```

問題文はまず BUT の再帰的な定義から始まる．BUT には部分木の個数によって，0-BUT, 1-BUT, 2-BUT の 3 種類がある．

0-BUT　木の末端（葉）に相当し，英字 1 字のラベルのみを持つ．

1-BUT　1 つだけ部分木を持つもので，英字 1 字のラベルと，BUT1 つを持つ．

2-BUT　2 つの部分木を持つもので，英字 1 字のラベルのほかに左右 2 つの BUT を持つ．

次に，BUT の式表現が定義される．これは，BUT の木構造を深さ優先，行きがけ順 (pre-order) で走査したもので，1-BUT の場合にはラベルの後にかっこで括った部分式を置く．2-BUT では，かっこの中に左と右の部分式をコンマで区切って並べる．BNF で記述すると以下の通り：

```
BUT ::= 0-BUT | 1-BUT | 2-BUT
0-BUT ::= LABEL
1-BUT ::= LABEL '(' BUT ')'
2-BUT ::= LABEL '(' BUT ',' BUT ')'
LABEL ::= 'a'|'A'|...|'z'|'Z'
```

最後に，BUT の図表現が定義される．木構造を，根を下として表す（つまり，計算機業界の木ではなく，自然界の木と同じ）．1-BUT では，ラベル（英字 1 字）のすぐ上にハイフンを置き，そのすぐ上に部分木の根（つまりはラベル）がくる．2-BUT では，ラベルの上にハイフンの列を置き，その列の左右の端のすぐ上に各部分木の根を置く．このとき，はじめのラベルは，ハイフン列の「中央」にこなければならない．ここで「中央」とは，ハイフン列の長さが奇数であればちょうど真中とし，偶数の場合には中心点の左のハイフンとすることになっている．

ここで，図表現における制限事項は以下の 2 つである：

1. 同じ行にあるラベルやハイフン（列）は 1 つ以上の空白で分離する

2. 0-BUT を表現するラベルのすぐ上にハイフンがこない

また，図表現のスペースが最小，という条件は，2-BUT のハイフンの長さの総和が最小であることを意味していると解釈しよう[☆1]．また，行数と空白の文字数も最小でなければならない．必要とする行数は木の深さによって一意に定まるので，前者は「空行を含まないこと」と同じである．後者は，全体をできるだけ左詰めに印字することに相当する．

入力は，式表現による BUT をセミコロンで区切ってならべ，最後の BUT の後だけはピリオドを置く．出力は BUT の番号を示す行（1: など）のあとに図表現を置く．

サイズに関する制限として，式表現におけるラベルの個数は 80 以下とされている．

また，式表現には構文エラーがないことも保証されている（つまりエラー発見や処理は不要である）．

■式表現の読み込み－構文解析

最初にすべきことは式表現の読み込みである．まず，読み込んでメモリ上に作り出す内部表現を決めよう．BUT のノードに対応する構造体として，以下を使うことにする．とりあえず内部表現に必要な要素だけを定義し，今後必要になる要素についてはおいおい追加することにしよう（オブジェクト指向の継承のメカニズムの原型）．

```
struct node {
  char label;
  char arity;   /* 0,1,2 */
  struct node *left, *right;
  /* other members ... */
};
```

ノードの種類は要素 arity で区別し，1-BUT の場合には要素 left だけを使用することとした．

構文解析の方法にはいろいろあるが，プログラミングコンテストの会場では構文解析器生成系（yacc など）は使えないし，そもそもごく単純な文法だから再帰降下法（recursive descent, いわゆるトップダウン方式）を使うことにしよう．

構文解析にはまず字句解析が必要であるが，幸いなことにこの式表現ではすべての構文要素（トークン）が 1 文字なので事実上字句解析は不要である．また，トークンを 1 つ先読みしておく必要があるが，これも 1 文字先読みでよい．その文字入力の部分は以下の通り：

```
int ch;
void nextch(void)
{
  do {
    ch = getchar();
    if(ch == EOF){
      fatal("EOF");
    }
  } while(isspace(ch));
}
```

ここで，isspace は改行も含めた空白文字をチェックする標準のマクロである．

構文解析の関数は以下の通り．

```
struct node *parse(void)
{
  struct node *result =
    (struct node *)malloc(sizeof *result);
  memset(result, 0, sizeof *result);
  if(!isalpha(ch)){
    fatal("letter expected");
  }
  result->label = ch;
  nextch();
  if(ch == '('){
    nextch();
    result->left = parse();
    if(ch == ')'){
      result->arity = 1;
      nextch();
    } else if(ch == ','){
      result->arity = 2;
      nextch();
      result->right = parse();
      if(ch != ')'){
        fatal("')' expected");
      }
      nextch();
    } else {
      fatal("',' or ')' expected");
    }
  } else {
    result->arity = 0;
  }
  return result;
}
```

1 文字先読みして ch に格納されている文字から始まる式表現を解析し，内部表現を作成してその根のポインタを返す仕様である．解析が終了した段階で，次の文字が ch に先読みされている．

なお，式表現の構文エラーはないと保証されているが，プログラムのエラーの早期発見の観点から，構文エラーの検出を行っている．関数 fatal はエラーメ

☆1 このことは問題文中には明示されていない．ほかに，「必要とする領域（空白も含めた文字数）が最小」といった解釈も可能であるが，本稿では採用しない．

ッセージを表示してプログラムを異常終了させるだけである．

全体の main 関数はおおよそ以下のようになる．関数 print_but は木構造の配置計算と印字のためのものである．その中の配置計算についてこれ以降で説明する．

```
int main(int ac, char **av)
{
  struct node *root;
  int qno = 1;
  nextch();
  while(1){
    root = parse();
    print_but(root, qno);
    qno++;
    if(ch == ';'){
      nextch();
    } else if(ch == '.'){
      break;
    } else {
      fatal("';' or '.' expected");
    }
  }
  return 0;
}
```

■解法 1 －左寄り戦略

BUT を図示するときに変化し得るのは，2-BUT のハイフン列の長さだけであり，それを最小になるように決定するのがこの問題の本質である．

最初に紹介する解法は，「できるだけ節点を左寄りに置く」という方針に基づくものである．

木全体を深さ優先でトラバースし，末端（下位）の節点の（水平）位置を定めてから，根に近い（上位）節点の位置を定める．深さ優先のトラバースの結果として，図表現は左上からできあがっていく．したがって，ある節点の位置を定める際に，その左隣に表示される節点はすでに決定済みである．そのために大域的な配列 v[] を用意し，v[d] には，深さ d の位置での現在配置済みの右端位置を記録すれば，この情報を利用して「できるだけ左寄り」が実現できる（**図 -1** の左側の太線が v[] を示し，そこに右側の木を寄せようとしている）．

ただし，下位の節点は上位と無関係に左に寄れるわけではない．たとえば 1-BUT が続く場合，図表現は垂直線状になる．その上位の節点のすぐ左に別の節点が配置されているとしたら，下位の節点群はいかに左に空きがあろうと寄るわけにはいかない．

これを解消するために，配置の手続きには「その節点が左に寄ってもよい限界位置」を引数 lim として渡すことにする．各節点はこの条件のもとで自分の「できるだけ左寄り」の位置を決定し，それを返り値 r とする．つまり，親の位置を「仮決め」して子の位置を決定し，それを使って親の位置を「本決め」するわけである．

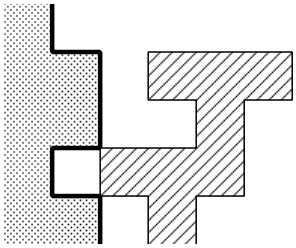

図 -1　左限界を保持する配列

0-BUT では r は max(v[d]+2, lim) である．1-BUT では r は部分木に max(v[d]+2, lim) を渡して再帰した結果となる．

2-BUT の場合がやや複雑である．まず左の部分木を処理するが，そのときの引数は max(v[d]+2, lim)-1 である．これは注目節点の限界の 1 つ左を限界とすることに相当する．もちろん左部分木はもっと左に寄っても構わないのだが，そうすると中央の位置が引数 lim より左には寄れないため，バランスをとるために右部分木の位置が逆に右に寄ることになって，要するスペースを大きくしてしまい，いいことがないからである．左部分木の処理が済むと，その返り値 +2 を引数として右部分木を処理する．左右の部分木の位置が決定すれば，注目節点の位置も決定できる．

具体的なコードを示す．まず，節点の構造体に，その（水平）位置を記録する要素 pos を追加する．配置の関数は以下の通り：

```
int p(struct node *n, int lim, int d)
{
  int m  = max(v[d]+2, lim);
  int r;
  int rl, rr;

  switch(n->arity){
  case 0:
    m = max(m, v[d+1]+1); /* ! */
    r = m;
    break;
  case 1:
    r = p(n->left, m, d+1);
    break;
  case 2:
    rl = p(n->left, m-1, d+1);
    rr = p(n->right, rl+2, d+1);
```

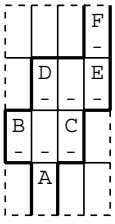

図 -2　木構造の外形の表現

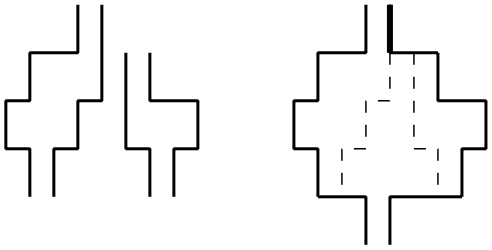

図 -3　部分木の外形の融合

```
    r = (rl + rr)/2;
    break;
  }
  n->pos = r;
  v[d] = r;
  return r;
}
```

0-BUT の場合に行っている処理（/* ! */ の行）は，前述した制約条件 2（0-BUT のすぐ上は空白であること）のための処理である．

大域配列を 1 つ使うほかは局所的な情報のやりとりだけで済み，なかなかよいプログラムといえるのではないだろうか．問題文の中に示されるテストデータは正しく処理できた．

しかし，意外なことに，審判団が使用した本番データでは正解と異なる結果を出した問題が見つかった．実はこの解法は間違っていたのである．どこが間違っているかは本稿の最後に述べることにしよう．

■解法 2 －部分木接近戦略

もう 1 つのアプローチは，2-BUT における 2 つの部分木を独立に配置した後でそれらをできるだけ接近させる方法である．

2 つの木構造を隣接させるには，それぞれの外形を知らないといけない．そのために，左右の端を記録する整数配列を部分木ごとに用意する．実際に記録するのは中心線（その木の根の位置）を 0 とした端の位置で，左右とも幅方向に正とする（**図 -2** では，左端配列 : {0, 1, 0, -2}, 右端配列 : {0, 1, 2, 2} となる）．

配置を行う関数はこれらの配列も引数として受けとり，配置結果に応じて該当する要素に値をつめて返す．接近の際には，左部分木の右端の配列と右部分木の左端の配列を深さごとに調べ，（それぞれの値の和 +1）の最大値が 2 つの木の最短間隔となる．ただし，はじめに述べた制約条件（0-BUT のすぐ上は空白であると）は別途考える必要がある．

間隔を決定したのち，全体としての左右の端を計算する．左部分木の左端を，間隔の半分だけ左に寄せる（つまり値を増やす）．右部分木の右端についても同様である．ただ，この段階で注意しなければならないのは左右の部分木の深さが異なる場合で，浅い方の配列が尽きたその先は，深い方の内側の端（左部分木なら右端）を引き継いでおかないといけない（**図 -3** に 2 つの部分木の融合を示す．右の部分木の深さが少ないため，右端の配列は上方で元の左部分木の右端の一部（太線）を引き継いでいる）．

以上を考慮したコードを示す．節点の構造体には，前節の pos にかわって，2-BUT における部分木の根の間隔を示す dist を追加した．

関数 dist は，2 つの「端配列」とそれぞれのサイズを受け取って，その 2 つの端を最も接近させたときの距離を求めるものである．0-BUT に関する制約条件もここで考慮している．

関数 merge_v は部分木の外形の融合の手続きである．外側の端配列 v とそのサイズ n, 内側の端配列 vrest とそのサイズ m, 端をずらす量 d と内側の端が引き継がれる場合（前述）のずらし量 drest が引数である．

関数 p が配置を決定する．n が処理対象の節点，lv と rv が配置結果となる左右の端配列を格納すべき領域である．返り値は部分木の深さとなる．

```
#define NNODE (80+10)   /* ノード数上限 + マージン(10) */

/* n1(length of v1) >= n2 */
int dist(int *v1, int *v2, int n1, int n2)
{
  int i, r = 0;
  for(i=0; i<n2; i++){
    r = max(r, v1[i]+v2[i]+2);
    if(i < n1-1) r = max(r,
v1[i+1]+v2[i]+1);
    if(i > 0) r = max(r, v1[i-1]+v2[i]+1);
  }
```

```
  return r;
}

void merge_v(int *v, int *vrest, int n,
             int m, int d, int drest)
{
  int i;
  for(i=0; i<n; i++){
    v[i] += d;
  }
  for(; i<m; i++){
    v[i] = vrest[i] - drest;
  }
}

int p(struct node *n, int *lv, int *rv)
{
  int d, dl, dr, di;

  lv[0] = 0;
  rv[0] = 0;
  switch(n->arity){
  case 0:
    d = 0;
    break;
  case 1:
    d = p(n->left, lv+1, rv+1);
    break;
  case 2:
    {
      int lrv[NNODE], rlv[NNODE];
      int rl, rr;

      dl = p(n->left, lv+1, lrv);
      dr = p(n->right, rlv, rv+1);

      if(dl >= dr){
        di = dist(lrv, rlv, dl, dr);
      } else {
        di = dist(rlv, lrv, dr, dl);
      }

      n->dist = di;

      d = max(dl, dr);
      merge_v(lv+1, rlv, dl, d,
              (di/2), (di+1)/2);
      merge_v(rv+1, lrv, dr, d,
              (di+1)/2, (di/2));
    }
    break;
  }
  return d+1;
}
```

■結果の印刷

配置結果を印刷するについては，1 行ごとに木構造を走査して印刷する方法と，2 次元の文字配列のバッファを用意してその上に木構造を配置して一度に印刷する方法がある．

前者の方法をとると，木構造の走査の順など面倒が大きい．今回の問題では，サイズの制約条件から節点数の上限（80）が決まっており，バッファのサイズも容易に見積もれるので後者の方法をとることにした．左右方向にはたかだか 80 文字，上下方向にも 160 行で十分である．

木構造を深さ優先で走査し，節点ごとに対応するバッファ位置に文字を埋め込んでいけばよい．最後に上と左のマージンを除いて印刷すれば完了である．プログラムコードは省略する．

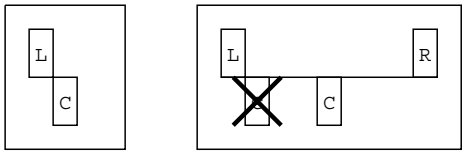

図 -4　解法 1 の問題点

■解法 1 の問題点

最後に，本文中で述べた解法 1 のどこに問題があったかを述べておく．解法 1 において，2-BUT の左部分木の限界位置を決める際に，「注目節点の限界位置の 1 つ左」としたのが問題点である．

図 -4 では，C をラベルとする節点を処理しようとしている．左部分木 L の限界位置を C の 1 つ左に設定し，再帰的に処理にとりかかる（図左）．左部分木の処理が完了し，L の位置が正式に決まると，続いて右の部分木の処理にかかる．このとき，左部分木と右部分木が左右に広がっていると，右部分木の正式位置 R はかなり右にずれることになる．ここで，C の正式位置は L と R の中間であるから，C は当初の限界位置よりは右によった位置にきてしまい，最も左に寄せるという目的に反する結果となる．

もし L の限界位置としてもっと左寄りに与えておけば，C は当初の限界位置どおりの場所に置くことができたはずであるが，そのことは L の配置が済むまでは分からないのである．

C が一度定まった後で，当初の限界位置からずれていた場合に再配置を行うことも可能ではあろうが，呼出の中に 2 回以上の再帰呼出を含む場合には指数関数オーダの時間を必要としてしまうから，現実的な解法にならないおそれがある．

（平成 15 年 9 月 12 日受付）