ARBRES BINAIRES

La structure d'arbre binaire exposée ci-dessous a été utilisée pour un compilateur d'automate qui a été publié dans la revue Pascalissime à partir du numéro 61.
A ce qui a déjà été dit dans l'article consacré au compilateur d'automate, j'apporte ici un complément sur l'utilisation de l'arbre binaire pour mémoriser les identificateurs du langage.

Minimiser le temps de recherche d'un identificateur parmi ceux du langage est la principale raison d'emploi d'une structure d'arbre binaire pour mémoriser les identificateurs du langage dans l'objet TListIdMgr.

L'arbre binaire est une structure de données liée à une relation d'ordre procurant un nombre moyen de comparaison pour la recherche d'un élément de l'ordre de log2 n. C'est à dire que l'on effectue de l'ordre de 15 comparaisons pour trouver un élément parmi 32768 (215). Bien sur, dans le meilleur des cas, on effectue une seule comparaison si l'élément est le premier de l'arbre, et malheureusement 32768 si l'arbre est complètement déséquilibré (voir figure 1 et 2).

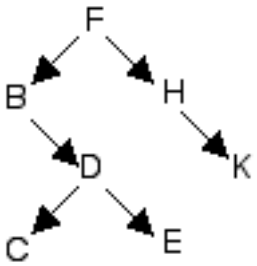

Figure 1

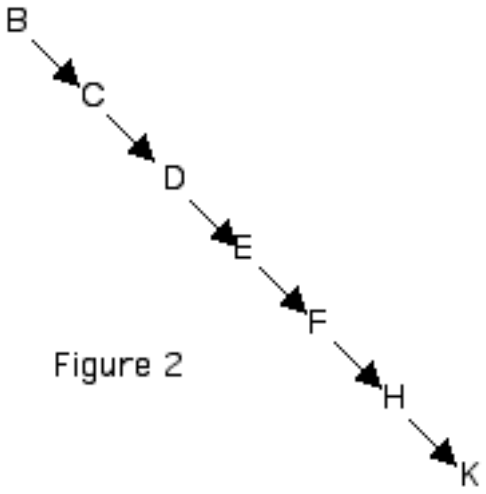

Figure 2

Pour l'arbre de la figure 1, il faut 3 comparaisons pour trouver l'élément "K" alors qu'il en faut 7 pour l'arbre de la figure 2.

Voila, le mot est lâché : la performance d'un arbre binaire est lié à sa construction équilibrée ou non.

L'ordre d'ajout des éléments est donc très important lors de la construction de l'arbre.

Dans la plupart des cas on ne connait pas à priori la valeur des éléments par avance, alors on doit se contenter du facteur "de l'ordre de log2 n", ou mettre en place des algorithmes de rééquilibrage de l'arbre ce que l'on verra ensuite.

Par contre, dans le cas qui nous intéresse, on a l'avantage de connaître tous les identificateurs du langage. Nous allons donc les ordonner correctement pour obtenir un arbre le plus équilibré qui soit.

L'algorithme de classement se résume à prendre à chaque fois l'élément central d'une suite d'éléments ordonnés, par dichotomie.

Pour nos identificateurs cela donne :

"action automate call default end error event from gosub init out to same".

L'élément central est "event".

Puis on constitue deux sous-listes :

"action automate call default end error" et "from gosub init out to same", dont les éléments centraux sont "call" et "init". (On remarque que si le nombre d'éléments est pair le choix de l'une ou l'autre possibilité n'affecte pas la performance finale, bien que les arbres soient différents).

Puis on constitue les sous-listes :

"action automate", "default end error", "from gosub", "out to same", dont les éléments centraux sont "action", "end", "from", "to".

Puis on constitue les sous-listes :

"automate", "default error", "gosub", "out same", dont les éléments centraux sont "automate", "default", "gosub", "out".

Puis on constitue les sous-listes :

"error", "same", dont les éléments centraux sont "error", "same".

Ce qui donne de l'ordre de construction suivant :

"event call init action end from to automate default gosub out error same".

Et l'arbre binaire suivant :

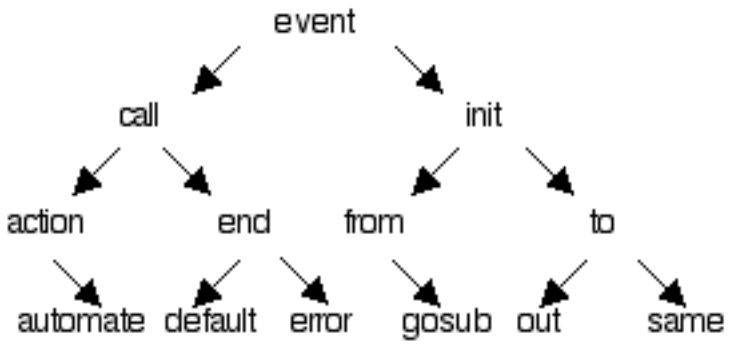

Figure 3

L'arbre est bien équilibré, le nombre de comparaisons maximum pour une recherche sera de 4 soit log2 de 16.

Le codage de la construction ci-dessus peut s'obtenir en appelant le méthode "Add" dans l'ordre indiqué. Cependant, ce n'est pas toujours possible ou alors cette contrainte est trop forte si la liste n'est pas figée.

Pour équilibrer notre arbre nous allons donc ajouter une méthode dans l'objet que nous nommons "Balance", dont le code est le suivant :

```
const
      C_Nb_Elmt_Max = 512;
type
      TNbElmt = 0..C_Nb_Elmt_Max;
var
      Tab: array[1..C_Nb_Elmt_Max] of PListId;
```

```
{ Procédure qui balance l'arbre de façon à minimiser le temps de
recherche }
    procedure TListIdMgr.Balance;
        var
            NbElmt: TNbElmt;

        procedure PlaceDansTab (Noeud: PListId);
        begin
            if Noeud^.G <> nil then
                PlaceDansTab(Noeud^.G);
            NbElmt := NbElmt + 1;
            Tab[NbElmt] := Noeud;
            if Noeud^.D <> nil then
                PlaceDansTab(Noeud^.D);
        end;

        procedure PlaceDansArbre (var Noeud: PListId;
                                  Premier, Dernier: TNbElmt);
            var
                Index: TNbElmt;
        begin
            Index := (Premier + Dernier) div 2;
            Noeud := Tab[Index];
            if Premier <> Index then
                PlaceDansArbre(Noeud^.G, Premier, Index - 1)
            else
                Noeud^.G := nil;
            if Dernier <> Index then
                PlaceDansArbre(Noeud^.D, Index + 1, Dernier)
            else
                Noeud^.D := nil;
        end;
    begin
        NbElmt := 0;
        PlaceDansTab(Liste);
        PlaceDansArbre(Liste, 1, NbElmt);
    end;
```

On peut appeler "Balance" chaque fois qu'il est nécessaire. C'est à dire, bien souvent soit après une série d'ajout d'éléments soit avant une recherche.

Le codage de "Balance" suit la démarche décrite précédemment en plaçant en premier lieu les éléments de l'arbre suivant la relation d'ordre puis en les replaçant dans l'arbre en suivant la dichotomie.

L'objet de gestion de l'arbre binaire devient assez important pour que l'on puisse le placer de façon autonome dans une unité de compilation nommée : "ArbreMgr".

(source de l'unité arbremgr.p)

Cette unité fait appelle à l'unité "BasicDef" où l'on définit le type de l'élément manipulé, la constante de l'élément non défini et le type de la clef qui doit être un type simple que l'on peut utiliser avec les comparateurs du Pascal. L'utilisation de l'unité "BasicDef" donne la possibilité de garder le même source pour l'unité "ArbreMgr" qu'elle que soit l'utilisation, dans la limite d'une utilisation par programme.

(source de l'exemple essai.p et basicdef.p)
(texte du fichier essai.out)

Le langage Ada permet de s'affranchir des deux précédentes limitations en utilisant un "package generic" dont la définition pourrait être :

```
{ Package assurant la gestion de l'arbre binaire. }
generic
      type TClef is private;
      type TElement is private;

      with function "<" (Left, Right : TClef) return Boolean is <>;

package ArbreMgr is

      procedure Init;
      procedure Ajoute (Clef: in TClef; Element: in TElement);
      procedure Balance;
      procedure Recherche (Clef: in TClef; Element: out TElement);
      procedure AfficheListe;
      procedure AfficheArbre;
      procedure Done;

end ArbreMgr;
```

Et l'appel est le suivant :
package MonArbre is new ArbreMgr (Integer, Integer, "<");

Voilà une bonne idée de départ d'utilisation de l'ADA ...

Pascal Pignard (Novembre 1999).