TP 13 : arbres préfixes (commentaire du programme)

Gilles Peskine

17 janvier 2003

Voir le code des bibliothèques dans les fichiers `liste_assoc.c` et `arbre_prefixe.c`, les en-têtes `liste_assoc.h` et `arbre_prefixe.h`, et un exemple d'utilisation des arbres préfixes dans `demo13.c`.

1 Listes d'associations

On commence par définir `struct cellule`. En tant que cellule de liste, il y a un champ `s` qui pointe vers la cellule suivante de la liste, et les autres champs forment l'élément de la liste. S'agissant de listes d'associations, il y a deux autres champs : `c` pour la clé, et `donnee` pour la valeur associée.

Le champ `donnee` est de type `void*`. N'importe quel pointeur de type `toto*` peut être converti en `void*`, puis reconverti en `toto*` sans perte d'information. On utilise donc le type `void*` quand on veut manipuler des données dont on ne connaît pas le type (il y a un certain nombre d'exemples dans la bibliothèque standard : `malloc`, `qsort`, `memcpy`...). On n'a pas besoin de *cast* pour convertir entre `void*` et un type pointeur sur données. Évidemment, il faut faire attention à respecter le type original lorsqu'on déréférence le pointeur : on peut écrire « `int i = 3; void *p = &i; float *f = p;` » mais cela ne fera pas de `*f` un flottant valant 3 (la valeur de `f` est celle de la suite de `sizeof(float)` octets stockée à partir de l'adresse de `i`, ce qui peut d'ailleurs très bien ne pas être un flottant valide).

Pourquoi `void*` ? Parce qu'il existe des machines sur lesquelles tous les pointeurs n'ont pas la même représentation. Sur beaucoup de machines, la mémoire est considérée comme un vaste tableau d'octets et un pointeur est tout simplement le numéro de l'octet. Cependant, sur certaines machines où l'unité naturelle est un mot de plusieurs octets, les pointeurs sur des données de taille multiple d'un mot (`int`, pointeurs, flottants...) sont le numéro du mot où débute l'objet (donc typiquement 2, 4 ou 8 fois le pointeur sur octet correspondant).

Enfin, un `void*` permet de représenter tout pointeur sur une donnée, c'est-à-dire que les pointeurs sur des fonctions sont exclus. Il n'y a pas de manière portable d'autoriser une case mémoire à contenir un pointeur sur fonction de n'importe quel type (s'il y a un nombre fini de types possibles, on peut bien sûr avoir recours à une `union`).

La fonction `cons` alloue la mémoire pour une nouvelle cellule de liste. S'il n'y a pas de mémoire disponible, elle appelle `abort`. On remarque que la fonction `cons` est déclarée `static` : elle n'est donc pas visible depuis un autre fichier source.

La fonction `aliste_recherche` recherche la donnée associée à un caractère `c` dans une liste d'associations `*l`. Elle renvoie non pas la valeur du champ `donnee` associé à `c`, mais son adresse, ce qui a deux avantages. D'une part, ceci permet de distinguer le cas où `c` est introuvable (la fonction renvoie `NULL`) du cas où la donnée se trouve être le pointeur nul (la fonction renvoie un pointeur valide, l'objet pointé étant un pointeur nul dans le type `void*`). D'autre part, ceci permet au code qui appelle `aliste_recherche` de modifier le champ `donnee` lui-même.

La fonction `aliste_recherche` prend en argument un pointeur sur un pointeur sur la première cellule de la liste, ce qui permet de coder une optimisation classique (non faite ici) : quand on trouve un élément, on le déplace au début de la liste ; il peut y avoir un gain de temps conséquent si on recherche fréquemment les mêmes éléments.

La fonction `aliste_ajoute_au_debut` ajoute une association d'un caractère `c` à une donnée `d` à une liste d'association `*l`. Elle ne vérifie pas si la clé est déjà présente. Il peut donc y avoir plusieurs cellules dans la liste qui ont la même clé, et c'est la première en partant du début, donc celle ajoutée le plus récemment, qui compte (c'est elle que `aliste_recherche` trouve et que `aliste_supprime` retire ; en revanche `aliste_parcours` trouve toutes les associations). La fonction `aliste_ajoute_au_debut` renvoie l'adresse du nouveau champ `donnee` créé (dont la valeur initiale est `d`).

La fonction aliste_supprime retire une association d'une liste. Si la clé est présente, elle supprime la cellule correspondante et renvoie 1 (si la clé est présente plusieurs fois, seule la première occurrence est supprimée). Si la clé est absente, aliste_supprime ne fait rien et renvoie 0.

La fonction aliste_parcours applique la fonction f à toutes les paires (clé, valeur) présentes dans la liste. Le paramètre supplémentaire p est passé à chaque appel de f. Il permet de passer des arguments supplémentaires à la fonction, et aussi de lui faire (indirectement) retourner un résultat. Voici par exemple une fonction qui renvoie la longueur d'une liste d'associations :

```
void aliste_incr(char c, void **d, void *p) {
    size_t *pn = p;
    ++*pn;
}
size_t aliste_longueur(aliste l) {
    size_t n = 0;
    aliste_parcours(l, aliste_incr, p);
    return n;
}
```

2 Arbres préfixes

À chaque nœud de l'arbre, on stocke une liste des sous-arbres (liste d'association indexée par le caractère suivant du mot, avec pour donnée un pointeur vers le nœud fils correspondant). On stocke également le mot correspondant au chemin vers ce nœud, mais seulement si le mot en question est dans l'ensemble représenté : sinon le champ mot contient NULL. Si le mot est dans l'ensemble, le champ donnee permet de lui associer une donnée.

L'arbre vide est représenté par un pointeur nul.

La fonction cree_noeud alloue la mémoire pour un nœud, qui démarre vide.

La fonction ap_insere ajoute un mot à l'arbre. Elle parcours l'arbre en partant de la racine et en descendant suivant les lettres du mot. Si elle tombe sur un nœud qui n'a pas de fils correspondant à la lettre courante, elle ajoute une entrée à la liste des fils et elle crée le fils correspondant. Enfin elle ajoute le mot et la donnée dans le nœud qui vient peut-être d'être crée.

La fonction ap_recherche parcourt l'arbre à la recherche d'un mot. Si l'arbre est vide, ou si la lettre courante n'est pas présente dans la liste des fils, la recherche s'arrête et la fonction ap_recherche renvoie NULL. Si on est arrivé au nœud correspondant au mot recherché, on teste s'il est effectivement présent, et on renvoie l'adresse du champ donnee ou NULL selon si c'est ou non le cas. Sinon la fonction se rappelle elle-même sur le nœud fils et sur le mot privé de sa première lettre.

On pourrait combiner les fonctions ap_recherche et ap_insere en une seule fonction prenant quatre paramètres : un pointeur sur un arbre, un mot, un booléen indiquant s'il faut ajouter le mot s'il est absent, et une donnée à insérer en cas de besoin.

La fonction ap_supprime parcourt elle aussi l'arbre à la recherche d'un mot. Cependant elle est un peu plus compliquée, parce qu'elle peut devoir supprimer non seulement un nœud, mais aussi tout une partie de l'arbre qui ne sert qu'à stocker ce mot et aucun autre. Par exemple, si l'arbre contient les mots bonjour et bonsoir et qu'on retire bonsoir, il faut supprimer les nœuds correspondant aux lettres s, o, i et r du mot bonsoir.

La fonction ap_parcours appelle f(n->mot, &n->donnee, p) pour tous les nœuds n de l'arbre. Le travail essentiel est dans la fonction parcours_branche. Parcourir un arbre, c'est parcourir le sommet, puis parcourir chaque sous-arbre. Et parcourir chaque sous-arbre, c'est appeler la fonction parcours_branche sur chaque élément de la liste des fils.

Le paramètre supplémentaire passé à la fonction aliste_parcours contient deux morceaux : le paramètre supplémentaire passé à ap_parcours (extra->p), et la fonction à appliquer à chaque mot (extra->f).

Noter que le parcours de l'arbre ne s'effectue pas en ordre alphabétique. Toutefois on traite toujours un mot avant ses éventuels suffixes. Pour obtenir un parcours alphabétique, il suffit de commencer par trier dans chaque nœud la liste des fils.

Le parcours d'un arbre est naturellement récursif. Alors qu'avec des listes on a le choix entre boucles et récursivité, beaucoup d'opérations sur les arbres sont difficiles à écrire sans utiliser de fonction récursive (on est amené à manipuler une pile à la main, donc à refaire le travail que fait le compilateur si on a une fonction récursive).