Structures de données en langage C

Notes de cours pour le module de programmation C dispensé en L2 Maths (option Informatique) à l'université d'Evry (deuxième semestre 2011).

- TD1 pdf latex solution
- TD2 pdf latex solution
- TD3 pdf latex
- DL1 pdf latex corrigé
- DS pdf latex corrigé

Table des matières

1 Rappels

1.1 Constitution d'un programme

Un exemple : fonction exponentielle

```
// Commentaire d'une ligne
/* Commentaire
   sur plusieurs lignes */

// Utilisation de bibliothèques
#include <stdlib.h>
#include <stdio.h>

// Définition d'une fonction
// Valeur de retour, nom, argument, corps
float f(float x, int n) {
  // Déclaration de variable
  int j;
  // Déclaration de variable avec initialisation
  float r = 1;

  // Structure de contrôle : boucle
  for(j = 0; j < abs(n); j++) {
```

```
    r = r * x;
  }

  // Structure de contrôle : branchement conditionnel
  if(n < 0) {
    // Valeur de retour de la fonction
    return (1.0 / r);
  }
  else {
    return r;
  }
}

// Fonction principale, où commence l'exécution.
int main() {
  // appel des fonctions printf et f
  printf("%f\n",f(2,-5));
}
```

Quelques définitions

Un programme est constitué d'instructions, organisées en blocs, structures de contrôle et fonctions. Une **instruction** est un ordre simple exécuté par la machine (le plus souvent, pour modifier la valeur d'une variable en mémoire). Un **bloc** est une suite d'instructions ou de structures de contrôle exécutées séquentiellement. Une **structure de contrôle** permet d'organiser l'exécution des blocs : les **structures conditionnelles** (if, switch) exécutent un bloc ou un autre en fonction d'une condition ; les **boucles** (for, while) permettent de répéter un bloc tant qu'une condition est vérifiée.

NB: Les blocs peuvent contenir des structures de contrôle qui contiennent des blocs qui peuvent à leur tour contenir une structure de contrôle, etc … Les programmes sont donc des exemples d'objets définis **récursivement**.

Une **fonction** est un bloc que l'on peut appeler depuis une instruction en donnant des paramètres. Une fonction est définie par son **nom**, ses **arguments** (ou paramètres) et son **corps** (qui est un bloc où on utilise les arguments, voire le nom). Lors de l'appel d'une fonction, l'exécution saute depuis le bloc d'appel vers le corps de la fonction, et les arguments de la fonction sont initialisés avec les valeurs données à l'appel.

Un **programme C** est une collection (ordonnée) de fonctions. L'une d'entre elles doit s'appeler **main**. L'exécution commence par l'appel de cette fonction.

1.2 Variables et types de données

Les **variables** sont des emplacements en mémoire auxquels on a donné un nom et dont on peut modifier le contenu. Une variable a un **type**, qui détermine :

- les valeurs qu'elle peut prendre (une variable entière ne peut pas contenir de valeur flottante)
- le codage en mémoire (c'est-à-dire comment une valeur est représentée par une séquence de bits)
- la sémantique des opérations (par exemple, la division est toujours représentée par le symbole `/`, mais est définie différement sur les entiers et les flottants).

  Pour créer une variable (et l'utiliser par la suite), il faut la déclarer, en donnant son type et éventuellement une valeur initiale.

```
int compteur = 0;
float x;
```

Les variables ont une **portée** limitée, c'est-à-dire qu'elles existent seulement dans une portion bien définie du programme. On distingue les **variables globales**, définies en dehors d'un bloc, et les **variables locales**, définies au début d'un bloc. Les variables globales sont accessibles dans toutes les fonctions (et donc tous leurs blocs) définies après la déclaration de la variable. Une variable locale n'est accessible qu'à l'intérieur du bloc où elle a

été définie. Enfin, les arguments d'une fonction se comportent exactement comme des variables locales, qui sont initialisées automatiquement lors de l'appel de la fonction.

Exercice : Qu'affiche le programme suivant ?

```
#include <stdlib.h>
#include <stdio.h>

int x = 0;

void f(int x) {
  printf("%d\n",x);
  {
    int x = 2;
    printf("%d\n",x);
  }
}

void main() {
  printf("%d\n",x);
  f(1);
}
```

Les types vus au premier semestre:

- les types primitifs (ou types simples) tels que les **entiers** (`int`, `short`, `long`, etc …), les **flottants** (`float`, `double`), les **caractères** (`char`)
- le type `void` qui représente l'absence de valeur
- les **structures** (aussi appelées records, champs d'enregistrement) qui correspondent au produit cartésien. Elles servent à rassembler des données qui sont logiquement liées commme par exemple, les valeurs réelles et imaginaires d'un nombre complexe :

```
struct { re : float ;
         im : float }
```

- les **tableaux**, qui sont des séquences de variables de taille spécifiée dans le programme (pour le moment)

```
int t[42];
```

2 Gestion dynamique de la mémoire

2.1 Problématique

Les variables locales permettent d'utiliser simplement la mémoire de la machine, parce que le compilateur C effectue deux tâches importantes :

- il génère une erreur si on utilise une variable sans l'avoir créée (via une déclaration).
- il libère la mémoire automatiquement lorsque la variable cesse d'être accessible (à la sortie du bloc où elle a été créée).

Le compilateur prend donc en charge la création et la destruction des variables dans un cadre bien délimité, à savoir l'utilisation des variables locales. Cette fonctionnalité simplifie le code et limite les erreurs, mais se révèle vite insuffisante. Un exemple : représentation des vecteurs de dimension quelconque, avec une structure

```
struct vector {
  int dim;
  float coord[ ?? ]; // la taille du tableau devrait être égale à dim ...
}
```

Cette structure ne peut être initialisée par le compilateur parce qu'on ne peut pas exprimer en C qu'il existe une relation entre le champ `dim` et le tableau `coord` (en l'occurrence que la longueur du tableau doit être égale à `dim`).

Le langage C propose des outils pour gérer manuellement la mémoire, c'est-à-dire pour réserver des espaces en mémoire, désigner ces espaces et les rendre lorsqu'ils ne sont plus utilisés.

2.2 Adresses et pointeurs

Les machines actuelles manipulent l'information sous la forme de séquences de **bits**, qui sont des cellules ayant deux états possibles (allumé ou éteint). La taille (ou capacité) d'une mémoire est exprimée généralement en **octets** qui sont des paquets de 8 bits. Chaque processeur peut opérer simultanément sur un nombre limité de bits ; cette unité "naturelle" du processeur et qui lui est spécifique est appelée un **mot**. Les machines actuelles (en 2011) ont des mots de 32 ou 64 bits (soit 4 ou 8 octets).

On peut se représenter la mémoire d'un ordinateur comme un grand tableau de mots, que doivent se partager les programmes s'exécutant sur la machine. Chaque case de ce tableau est associée à une **adresse**, qui ressemble moralement à un entier, représentant le numéro de la case.

Toute variable a une adresse en mémoire, c'est-à-dire un emplacement où est stocké le contenu de la variable. On peut l'obtenir grâce à l'opérateur `&`. Ainsi, si `a` est une variable, `&a` désigne son adresse en mémoire. Les adresses constituent aussi un type de données en C, et ont une notation spéciale : l'adresse d'une variable de type `t` se note `t*` (par exemple une adresse d'entier a pour type `int*`). On peut bien sûr créer des variables qui contiennent des adresses : ce sont les **pointeurs**. Le pointeur désigne une adresse en mémoire, et on peut inversement lire la valeur qui réside à cette adresse avec l'opérateur `*`. Si `p` est un pointeur (c'est donc une adresse en mémoire), `*p` représente la valeur se trouvant à l'adresse pointée par `p`.

Exercice : Quelle est la valeur de `b` à la fin de ce bloc ?

```
int a = 3;
int b;
int* p;

p = &a;
b = *p;
```

Les pointeurs de type `void *` constituent une exception : ils indiquent une adresse sur une valeur dont on ne connait pas (ou pas encore) le type. Une conséquence immédiate est que l'on ne peut pas regarder le contenu pointé par un pointeur de ce type : puisque l'on ne connait pas le type du contenu, on ne sait pas comment la valeur est codée. On dit que les pointeurs de type `void *` sont *non-déréférençables*.

Enfin en C, on définit une valeur particulière, appelée `NULL` qui ne pointe (par convention) sur rien. Elle correspond souvent à la valeur nulle, à savoir la toute première case mémoire. On se sert souvent de cette valeur particulière pour indiquer une erreur dans un traitement, ou un cas particulier d'une structure de données (voir chapitre sur les listes).

2.3 Retour sur les tableaux

Soit un pointeur `p` de type `t*`, `p` désigne un emplacement mémoire pouvant contenir une valeur de type `t`. L'expression `p + 1` désigne l'adresse immédiatement adjacente pouvant contenir une valeur de type `t`. Plus généralement on peut additionner un pointeur et un entier pour obtenir l'adresse du ième élément d'une séquence en mémoire (cette opération fait partie de ce qu'on appelle l'arithmétique des pointeurs). Les tableaux en C sont implémentés de cette manière : un tableau étant une séquence d'élements, ces éléments sont placés les uns à la suite des autres en mémoire.

Les variables de type tableau sont en fait des pointeurs. Si on définit le tableau `t` par `int t[10]`, alors `t` est un pointeur, de type `int*`, qui pointe au début du tableau. On a alors les équivalences suivantes (avec `i` une variables entière) :

Pointeur	Tableau

Pointeur	Tableau
t	&(t[0])
*t	t[0]
t + i	&(t[i])
*(t + i)	t[i]

Moralité : les pointeurs (et leur addition avec les entiers) sont le mécanisme de base pour implémenter les tableaux.

2.4 Retour sur l'appel de fonction : passage en valeur ou par référence

Lors d'un appel de fonction, avant d'exécuter le corps de la fonction, on crée une variable locale pour chaque argument, et on copie les arguments d'appel dans les variables locales. Une conséquence importante est que si l'on modifie les arguments d'une fonction, cela n'a pas d'impact en dehors de la fonction :

```
void f(int a) {
  a = 2;
}

void main() {
  int x = 0;
  printf("%d\n",x); // Affiche 0
  f(x);
  printf("%d\n",x); // Affiche encore 0
}
```

Si vraiment on veut modifier la valeur d'une variable donnée en argument d'un appel de fonction, on peut utiliser les pointeurs :

```
void f(int* a) {
  *a = 2;
}

void main() {
  int x = 0;
  printf("%d\n",x); // Affiche 0
  f(&x); // On transmet l'adresse de x
  printf("%d\n",x); // Affiche 2 !
}
```

La première méthode est appelée appel de fonction par **passage de valeurs**, elle implique une recopie et rend impossible la modification d'une variable qui serait passée en argument ; la deuxième est appelée appel de fonction par **passage de références**, ne demande que la copie du pointeur et permet la modification de la variable passée en argument.

NB La modification des arguments d'une variable grâce au passage de référence est notamment utilisée pour la fonction scanf, et permet de modifier plusieurs variables lors de l'appel

```
int x,y;
char z;

scanf("%d %d %c",&x,&y,&z); // Noter l'utilisation de l'opérateur &
```

2.5 Allocation mémoire

Jusqu'ici nous avons considéré des pointeurs sur des variables locales, donc gérées par le compilateur. D'ailleurs, voici une erreur courante, à ne pas reproduire bien sûr

```
(int *) f() { // la fonction f renvoie un pointeur sur entier
  int x = 42;
  return &x; // l'adresse de x est bien un pointeur sur entier mais ...
}

void main() {
  printf("%d\n",*(f())); // on imprime le contenu de la mémoire
                         // pointée par le résultat de f
}
```

Le souci dans cet exemple est que `x` est une variable locale, donc uniquement définie dans le corps de la fonction `f`. Quand l'appel à `f` se termine, la variable `x` n'existe plus et l'emplacement mémoire correspondant est rendu au système. Notre programme n'a plus le droit d'y accéder. S'il le fait quand même, il cause ce qu'on appelle une **faute de segmentation**, qui peut causer l'arrêt immédiat du programme (crash) ou des résultats incorrects.

Le langage C dispose de fonctions pour allouer de la mémoire et la rendre au système, ce qui permet une **gestion manuelle** de la mémoire.

Allocation On peut réclamer de la mémoire au système via la fonction **malloc**, dont le type est (en simpliant un peu) : `void* malloc(int size)`. La fonction `malloc` fournit un pointeur indiquant le début d'une zone mémoire de taille `size`. Le programme peut ensuite librement utiliser cette zone. Le pointeur retourné est de type `void*`, et dans la plupart des cas il faudra faire une **conversion de type** (cast, en anglais) explicite.

```
int* data = (int *) malloc(size);
```

La taille de la zone mémoire est à fournir en octets. Or bien souvent les variables ont une taille différente de 1 octet et cette taille peut varier en fonction de la machine sur laquelle le programme est exécuté. Pour connaître la taille mémoire occupée par une variable de type `t` en octets, on utilise l'opérateur `sizeof`. Pour l'allocation mémoire, on retrouvera très souvent des formules de la forme

```
t* data = (t *) malloc(nb_elements * sizeof(t));
```

Désallocation Une fois qu'un espace mémoire n'est plus utilisé, il faut le rendre au système, pour lui permettre de l'allouer à d'autres programmes. **Si une adresse a été obtenue par malloc**, alors on peut la libérer par la fonction **free**.

```
t* data = (t *) malloc(nb_elements * sizeof(t));
... // calculs sur data
free(data);
```

2.6 Mise en pratique

Nous illustrons maintenant ces fonctions en proposant une bibliothèque permettant de manipuler des vecteurs de flottants. Le premier choix à faire est la représentation en mémoire des vecteurs. Il faut manifestement simultanément représenter les coefficients du vecteur et sa taille, ce que nous faisons à l'aide d'une structure

```
struct vector_t {
  int dim;
  float* coef;
};
```

Plutôt que de manipuler la structure du vecteur directement, on utilisera un pointeur vers cette structure, c'est-à-dire des variables de type `struct vector_t *`. Oon utilisera donc le passage par référence. Pour alléger les notations, on a recours à des **alias de type**, avec l'opérateur **typedef**, qui permet de nommer un type.

```
typedef struct vector_t * vector;
```

Le type `vector` est maintenant synonyme de `struct vector_t *`. Comme il est très fréquent de manipuler une structure via un pointeur sur cette structure, il existe une notation spéciale pour accéder aux champs de la structure. Soit `v` une structure de type `struct vector_t`, on accède à ces champs avec la notation `v.dim` ; si on dispose d'un pointeur `p` sur `v`, on peut accéder au même champ avec la notation `p->v`.

La première fonction à créer pour notre bibliothèque est un constructeur qui permet de fabriquer des vecteurs d'une certaine taille. Pour cela, il faut allouer de l'espace pour une structure `vector_t` et initialiser ses champs. La dimension ne pose pas de problème, en revanche il faut aussi allouer de la mémoire pour stocker les coefficients.

```
vector vector_create(int n) {
  vector r = (vector) malloc(sizeof(struct vector_t));
  r->dim = n;
  r->coef = (float *) malloc(n * sizeof(float));
  return r;
}
```

La fonction `vector_create` commence par allouer de la mémoire pour la structure (cf utilisation de `sizeof`) via malloc. Noter également la conversion de type vers `vector`. La dimension est initialisée (opérateur `->`), puis les coefficients, avec un nouvel appel à `malloc`. On renvoie le pointeur pour finir.

La fonction `vector_create` n'initialise pas les coefficients : leur valeur est inconnue. On peut faire une version plus utile de `vector_create` qui produit un vecteur nul

```
vector vector_null(int n) {
  int i;
  vector r = vector_create(n);
  for(i = 0; i < r->dim; i++) {
    r->coef[i] = 0;
  }
  return r;
}
```

Quand le vecteur n'est plus utile, il faut libérer la mémoire qu'il occupe. On commencera par libérer la mémoire occupée par ses coefficients, puis par la structure

```
void vector_free(vector v) {
  free(v->coef);
  free(v);
}
```

Attention, il ne faut pas surtout pas procéder dans l'ordre inverse, qui causerait une faute de segmentation.

Enfin, voici d'autres fonctions éventuellement utiles, pour l'exemple.

```
// Affiche un vecteur à l'écran
void print(vector v) {
  int i;
  printf("[ ");
  for(i = 0; i < v->dim; i++) {
    printf("%f ", v->coef[i]);
  }
  printf("]");
}

// Calcule la norme d'un vecteur
float norm(vector v) {
  int i;
  float x = 0.;
  for(i = 0; i < v->dim; i++) {
    x += v->coef[i] * v->coef[i];
  }
```

```
  return sqrt(x);
}
```

Ainsi qu'un exemple d'utilisation des fonctions

```
void main() {
  vector v = vector_null(3);
  print(v);
  printf(" %f\n",vector_norm(v));
  vector_free(v);
}
```

L'ensemble du programme est accessible ici.

3 Listes

3.1 Prologue : fonctions récursives

Une fonction est récursive si son corps contient un appel ou plus généralement une référence à elle-même.

Exercice : Définir une fonction calculant le i^e terme de la suite de Fibonacci, définie par la récurrence suivante : \begin{align} F_0 & = 0
F_1 & = 1
F_{n+2} & = F_{n+1} + F_n
\end{align} Solution

```
int fibonacci(int n) {
  if(n <= 0) return 0; // On étend la suite aux entiers négatifs
  if(n == 1) return 1;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Les fonctions récursives permettent, comme les boucles `for` ou `while` de décrire des répétitions. Selon le cas, un calcul s'écrira plus naturellement dans un style **récursif** ou dans un style **itératif**. On peut toujours passer d'un style à l'autre, avec plus ou moins de difficultés.

Exercice : Ecrire une version itérative du calcul de la suite de Fibonacci.

Solution

```
int fibonacci(int n) {
  int f0 = 0;
  int f1 = 1;
  int i;

  if(n == 0) return f0;
  for(i = 2; i <= n; i++) {
    int x = f1;
    f1 = f0 + f1;
    f0 = x;
  }
  return f1;
}
```

Remarques :

- la version itérative est nettement moins simple à comprendre que la version récursive. Pour se convaincre qu'elle est correcte, il faut prouver un **invariant**, c'est-à-dire une propriété qui est vraie à chaque itération de la boucle `for`. Ici, on prouve qu'à l'issue de chaque répétition de la boucle `for`, la variable `f0` contient le

$(i - 1)^e$ nombre de Fibonacci et `f1` le i^e. En comparaison, la version récursive reproduit la définition de la suite.
- la version récursive demande beaucoup de calculs, ce dont on peut s'apercevoir en calculant le nombre d'appels récursifs à la fonction `fibonacci`. Soit $T(n)$ ce nombre, on a (grossièrement) la relation $T(n) = T(n-1) + T(n-2)$. Autrement dit, le nombre d'appels pour calculer les éléments de la suite de Fibonacci est une suite de Fibonacci. Avec un peu d'efforts, on trouve que F_n est asymptotiquement équivalente à φ^n où φ est le nombre d'or.
- la version itérative est en contrepartie beaucoup plus efficace que la version récursive, puisqu'elle demande seulement de l'ordre de `n` opérations (l'itération de la boucle).

Le style itératif est plus adapté lorsque l'on manipule des structures indexées par des entiers (tableaux, matrices) ; le style récursif est dans les autres cas souvent plus concis et plus proche de l'écriture d'une formule. Il est nettement plus simple à utiliser lorsque les données elle-mêmes sont de nature récursives.

3.2 Définition des listes

Soit $\mathbb{E}$ un ensemble d'éléments, une liste sur $\mathbb{E}$ est soit une liste vide, notée $\emptyset$, soit un couple constitué d'un élément de $\mathbb{E}$ et d'une liste sur $\mathbb{E}$. Autrement dit, un liste est soit vide, soit elle contient un premier élément, la **tête** de la liste, suivie du reste de ses éléments, sous la forme d'une liste appelée **queue**.

Exemples: $\emptyset$ est la liste vide, $(1,(2,\emptyset))$ est la liste contenant (dans ce ordre) 1 puis 2, $(3,(1,(4,(5,(9,\emptyset)))))$ est la liste contenant les premières décimales de π, $(\emptyset,(\emptyset, \emptyset))$ est une liste contenant deux listes vides, etc … Pour simplifier, on notera une liste non vide (h,t) h :: t.

Remarques :

- une liste est une structure **récursive** puisque les listes apparaissent dans leur propre définition.
- l'ensemble des listes $\mathbb{L}$ sur un ensemble $\mathbb{E}$ est la solution de l'équation

$$ \mathbb{L} = \{ \emptyset \} \cup \mathbb{E} \times \mathbb{L} $$

Une liste représente une *séquence finie* d'éléments, comme un tableau. La distinction porte sur *l'accès aux éléments* :

- tableau = accès direct à n'importe quel élément (avec l'index)
- liste = accès direct **seulement** au premier élément

On verra qu'en contrepartie, les listes permettent d'ajouter ou de supprimer plus simplement des éléments.

Nous allons maintenant définir le **type abstrait** des listes, c'est-à-dire donner des opérations sur les listes et leurs propriétés.

Opération
$\mbox{is}_{\text{empty}}$
Type
$\mathbb{L} \rightarrow \{ T, F \}$
Description
retourne T si la liste donnée en argument est vide et F sinon
Propriétés

$\mbox{is}_{\text{empty}}(\emptyset) = T$

$\mbox{is}_{\text{empty}}(h :: t) = F$

Comme on le voit sur cet exemple très simple, on parvient à donner une description formelle complète de l'opération en reprenant la définition des listes : on donne la valeur de l'opération pour chaque cas, d'abord le cas liste vide, ensuite le cas où la liste contient au moins un élément.

Opération
$\mbox{length}$
Type
$\mathbb{L} \rightarrow \mathbb{N}$
Description
retourne la longueur de la liste
Propriétés

$\mbox{length}(\emptyset) = 0$

$\mbox{length}(h :: t) = 1 + \mbox{length}(t)$

Pour connaître la taille de la liste, il faut la parcourir récursivement, jusqu'à arriver à la liste vide. Cette opération est une bonne illustration du principe "diviser pour régner" qu'on retrouve souvent en programmation, et particulièrement dans les structures récursives : si un problème est trop difficile à résoudre directement (ici le calcul de longueur d'une liste est trop difficile parce qu'on a accès directement qu'au premier élément), on le décompose en sous-problèmes plus simples (ici, une liste strictement plus courte), on résout récursivement, et on assemble les résultats à la fin.

Opération
$\mbox{member}$
Type
$\mathbb{E} \times \mathbb{L} \rightarrow \{ T, F \}$
Description
indique si un élément `e` est présent dans une liste `l`
Propriétés

$\mbox{member}(e,\emptyset) = F$

$\mbox{member}(h :: t) = T$ si $h = e$

$\mbox{member}(h :: t) = \mbox{member}(t)$ sinon

Opération
$\mbox{nth}$
Type
$\mathbb{N} \times \mathbb{L} \rightarrow \mathbb{E}$
Description
retourne un élément d'une liste en fonction de sa position
Précondition
$\mbox{nth}(i,l) \mbox{ si } i < \mbox{length}(l)$
Propriétés

$\mbox{nth}(0, h :: t) = h$

$\mbox{nth}(n, h :: t) = \mbox{nth}(n - 1, t)$

Nous voyons ici apparaître un élément capital pour les types abstraits, les **préconditions**. Ces préconditions sont des propriétés à vérifier avant d'appliquer l'opération. Les propriétés qui suivent peuvent supposer que les préconditions sur les arguments tiennent et ne sont valables qu'à cette condition. Ici, la précondition exprime simplement qu'on ne peut pas demander le 42^e élément d'une liste de longueur 4. Notons que cela dispense de définir l'opération `nth` pour la liste vide, puisqu'elle ne satisfait pas la précondition.

Opération
$\mbox{equals}$
Type

$\mathbb{L} × \mathbb{L} → \{T,F\}$

Description

teste si deux listes sont (structurellement) égales

Propriétés

$\mbox{equals}(∅, ∅) = T$

$\mbox{equals}(h_1 :: t_1, h_2 :: t_2) = equals(t_1,t_2)$ si $h_1=h_2$

$\mbox{equals}(l_1,l_2) = F$ dans tous les autres cas

Opération

$\mbox{cons}$

Type

$\mathbb{E} × \mathbb{L} → \mathbb{L}$

Description

ajoute un élément en tête de liste

Propriétés

$\mbox{cons}(e, l) = e :: l$

Opération

$\mbox{insert}$

Type

$\mathbb{E} × \mathbb{N} × \mathbb{L} → \mathbb{L}$

Description

$\mbox{insert}(e,i,l)$ ajoute l'élément e à la i^e position de la liste l.

Précondition

$\mbox{insert}(e,i,l)$ si $i < \mbox{length}(l)$

Propriétés

$\mbox{insert}(e, 0, l) = e :: l = \mbox{cons}(e,l)$

$\mbox{insert}(e, n, h :: t) = \mbox{insert}(e,n - 1,t)$

Opération

$\mbox{filter}$

Type

$\mathbb{E} × \mathbb{L} → \mathbb{L}$

Description

$\mbox{filter}(e,l)$ retire de l toutes les occurrences de e.

Propriétés

$\mbox{filter}(e, []) = []$

$\mbox{filter}(e, h :: t) = \mbox{filter}(e,t)$ si $h = e$

$\mbox{filter}(e, h :: t) = h :: \mbox{filter}(e,t)$ si $h ≠ e$

3.3 Implémentation

3.3.1 Représentation mémoire

En suivant la définition (inductive) des listes, on voit qu'il faut proposer une représentation pour chacun des cas, liste vide ou couple formé d'un élément puis d'une liste. Le deuxième cas doit immédiatement faire penser à des

structures, qui sont le moyen naturel en C de représenter des produits cartésiens (rappel : un couple est un cas particulier de produit cartésien). Cette structure doit contenir deux champs, à savoir la tête de la liste, qui est un élément, et la queue de la liste, qui est une liste. Comme pour les vecteurs vus au chapitre précédent, on utilisera un pointeur sur cette structure pour représenter les listes, avec deux conséquences :

- si la queue d'une liste est une liste, c'est donc un pointeur sur une structure
- si une liste est un pointeur, alors on peut conventionnellement choisir de représenter une liste vide par le pointeur `NULL`

Pour écrire du code C, il nous faut fixer le type des éléments de la liste, par exemple `char`. Les remarques précédentes nous conduisent à écrire

```
struct list_t {
  char head;
  struct list_t* tail;
};

typedef struct list_t* list;
```

Une liste `list` est un pointeur sur une `struct list_t`. Ce pointeur est nul si la liste est vide et sinon, on peut accéder à la tête et à la queue, qui est un pointeur sur `struct list_t`, c'est-à-dire une liste.

3.3.2 Constructeur et destructeur

Une fois la représentation choisie, il faut donner des fonctions permettant de construire des listes en mémoire, et de les libérer après utilisation. Le constructeur `cons` est le pendant de l'opérateur `::` vu plus haut

```
list cons(char e, list l) {
  list r = (list) malloc(sizeof(struct list_t));
  r->head = e;
  r->tail = l;
  return r;
}
```

Pour libérer la mémoire occupée par une liste, il faut libérer tous les éléments de cette liste. La formulation récursive aide ici à résoudre le problème simplement : soit la liste est vide et il n'y a rien à faire, soit elle ne l'est pas et dans ce cas on libère récursivement la queue de la liste, puis la structure abritant la tête

```
void free_list(list l) {
  if(!is_empty(l)) {
    free_list(l->tail);
    free(l);
  }
}
```

3.3.3 Opérations

Calculer la longueur d'une liste s'écrit en C quasiment comme dans le type abstrait vu plus haut

```
int length(list l) {
  if(is_empty(l)) return 0;
  else return 1 + length(l->tail);
}
```

tout comme la fonction `member`, qui teste la présence d'une valeur dans une liste

```
int member(char e, list l) {
  if(is_empty(l)) return false;
  else if(l->head == e) return true;
```

```
  else return member(e,l->tail);
}
```

ou encore la fonction `nth` qui renvoie le i^e élément d'une liste

```
char nth(int i, list l) {
  assert(!is_empty(l));
  assert(i>=0);
  if(i == 0) return l->head;
  else return nth(i - 1, l->tail);
}
```

Dans cet exemple, on voit une façon de traduire les **préconditions** en C, avec les **assertions**. La fonction `assert` en C (définie dès que l'on inclut `assert.h`) vérifie la condition qu'on lui donne en argument et stoppe le programme si celle-ci n'est pas vérifiée (indiquant au passage à quelle ligne du programme l'erreur s'est produite).

Exercice : Prouver (par récurrence sur la longueur de la liste donnée en entrée) que les assertions de l'implémentation de `nth` sont équivalentes aux préconditions données dans le type abstrait.

On parle d'**égalité structurale** lorsque l'on cherche à savoir si deux représentations en mémoire correspondent au même objet mathématique. Dans le cas des listes, il s'agit de savoir si elles contiennent les mêmes éléments dans le même ordre. On peut aussi s'intéresser à l'**égalité physique**, c'est-à-dire vérifier que deux listes correspondent au même objet en mémoire. L'égalité structurale peut aussi se déterminer récursivement, cette fois sur deux listes simultanément

```
int equals(list l1, list l2) {
  if(is_empty(l1)) return is_empty(l2);
  else if(is_empty(l2)) return false;
  else if(l1->head != l2->head) return false;
  else return equals(l1->tail, l2->tail);
}
```

Nous passons à présent à des opérations qui modifient les listes, c'est-à-dire des opérations qui ajoutent ou retirent des éléments. Dans le type abstrait, ces opérations reçoivent en entrée une liste et retournent une liste modifiée en sortie. Le type abstrait est un outil mathématique qui ne décrit pas comment les listes sont représentées en mémoire – ce qui est son principal avantage, puisqu'il permet de raisonner sur une structure de donnée en oubliant certains aspects. Néanmoins, au moment de l'implémentation, il faut réfléchir soigneusement à la question de la représentation mémoire. Plus spécifiquement, si une opération modifie une liste (par exemple, en ajoutant un élément), faut-il recréer une liste complète en mémoire, ou modifier la liste donnée en entrée ? En C, on optera pour la deuxième réponse, qui est à la fois la seule praticable et aux performances acceptables. Elle présente cependant une difficulté, à savoir que lorsque l'on fournit une liste en argument à une fonction, celle-ci peut fort bien ne plus être définie à l'issue de l'appel. Donnons tout de suite un exemple, avec la fonction `trim` qui supprime le premier élément d'une liste non vide

```
list trim(list l) {
  list r;
  assert(!is_empty(l));
  r = l->tail;
  free(l);
  return r;
}
```

L'assertion assure que la précondition est bien vérifiée, à savoir que la liste donnée en entrée doit être non vide. Ensuite pour supprimer le premier élément de la liste, il suffit de retourner la queue de la liste, en ayant pris soin de libérer la mémoire occupée par le premier élément. Plusieurs remarques :

1. l'argument `l` est de type `list` et est donc un pointeur indiquant l'adresse du premier élément, contenu dans une structure de type `struct list_t`. Pour libérer l'espace occupé par le premier élément, il suffit donc d'utiliser `l`, dans l'instruction `free(l)`.

2. l'ordre des trois dernières lignes est crucial, il ne faut surtout pas libérer la mémoire allouée au premier élément avant d'avoir récupéré l'adresse de la queue de liste. Notamment le code

```
list trim(list l) {
  assert(!is_empty(l));
  free(l);
  return l->tail;  // ERREUR : faute de segmentation
}
```

   provoque une faute de segmentation. Dès que le pointeur `l` a été passé a `free`, IL NE FAUT PLUS L'UTILISER.

3. comme annoncé plus haut, il faut bien penser que la liste donnée en entrée de `trim` n'est donc plus utilisable après l'appel.

Nous passons ensuite à l'opération `filter`, qui élimine les occurrences d'un élément donné dans une liste. La fonction C reprend la structure vue dans le type abstrait

```
list filter(char e, list l) {
  if(is_empty(l)) return l;
  else if(e == l->head) {
    list r = filter(e,l->tail);
    free(l);
    return r;
  }
  else {
    l->tail = filter(e,l->tail);
    return l;
  }
}
```

Dans le deuxième cas (celui où la tête de la liste correspond à l'élément recherché), on libère l'espace occupé par la tête, **en ayant au préalable sauvegardé l'adresse de la queue** à l'aide d'une variable intermédiaire. Là encore, en ne respectant pas cet ordre, on court à la faute de segmentation.

Nous voyons pour finir l'insertion d'un élément dans une liste.

4 Exécution d'un programme C

Voir ce document : PDF

5 Arbres

Voir ce document : PDF

Auteur: Philippe Veber et Tristan Bitard-Feildel <philippe.veber [AT] gmail.com>

Date: 2011-05-20 17:18:53 CEST

HTML generated by org-mode 6.33c in emacs 21