Listes, associations et interprétation

Sommaire

2. Implémenter une interface graphique
3. Explorer un mini-interprète
4. Tables de hachage
5. Conclusion

Implémentation d'une interface graphique

Pour préparer à la partie suivante, nous allons maintenant apprendre à programmer une interface simple en C. Comme nous sommes en 2017, il n'est pas question que l'interface de notre interprète soit une console (terminal) avec des puts et des gets : notre interface doit être graphique. Je vais ici vous guider pour construire un programme contenant une interface graphique minimale. Vous allez, à la fin de la partie, devoir appliquer cette interface à votre programme construisant un dictionnaire. Je vous conseille donc au fur et à mesure de tester votre code. Vous trouverez, à la fin, une référence pour télécharger un programme exemple.

Une remarque au passage : pour pouvoir utiliser un certain nombre de commodités de C++, bien que mes programmes soient fondamentalement écrits en C, ils portent l'extension .cpp et se compilent avec g++. Ces commodités sont en particulier : la définition de and et de or (plutôt que && et !!, décidément illisibles), les commentaires commençant par //, la possibilité de donner une valeur par défaut à un argument dans le prototype d'une fonction, exemple :

```
int fonction(int = 123) ;
```

Ce qui signifie que cette fonction, qui retourne un int, prend un int comme argument ; si elle est appelée sans cet argument, le paramètre sera initialisé à 123. Le prix à payer pour ces commodités c'est la plus grande sévérité dans les casts implicites (ex., le résultat de malloc doit absolument être casté explicitement).

1 - Les widgets

Les interfaces graphiques (graphical user interfaces, ou GUI) sont nés au début des années 1960 (voir en particulier On-Line-System), avec les fenêtres et la souris. Aujourd'hui, comme nous sommes sous Linux, notre environnement graphique de référence sera X11. Nous pourrions construire une interface graphique avec simplement des fonctions X11, disponibles naturellement en C. Cela nous prendrait beaucoup de temps ; nous allons plutôt utiliser des widgets, qui sont les composants génériques des fenêtres de nos interfaces graphiques.

Il existe beaucoup de bibliothèques de widgets. Mais si on se restreint aux bibliothèques open source, disponibles en C, faciles à trouver et sur lesquelles il y a beaucoup de documentation libre, le choix est plus limité. On a par exemple la bibliothèque des Athena widgets (Xaw), mais je lui ai préféré la bibliothèque Motif, devenue opensource sous le nom d'Openmotif (Xm), qui est plus riche et plus souple. Cependant les principes sont les mêmes partout.

Je n'ajouterais pas ici un manuel des widgets motif, il y en a déjà beaucoup d'excellents (par exemple, celui de Dave Marshall, et ceux mis en ligne ici). Je me contenterais du strict minimum nécessaire à comprendre l'interface proposée.

2 - Principes d'implémentation

La programmation d'une interface graphique est dirigée par les évènements : cliquer sur un bouton, bouger la souris au-dessus d'une fenêtre, taper des caractères... En fonction de chaque évènement, ou sélecteur, le widget où l'évènement a lieu (par exemple, le widget associé à un bouton) réagit par un comportement défini par une méthode propre. Sélecteur, méthode : oui, les widgets sont des

objets.

En fait, c'est un peu plus compliqué : chaque évènement (déplacement de souris, clic, touche enfoncée) est traduit en une action relativement à la classe du widget concerné ; chaque instance de cette classe peut redéfinir cette action. Il y a deux façons de redéfinir cette action :

- soit on utilise le mécanisme des callbacks, qui donne au programmeur la possibilité d'associer à des évènements plus ou moins abstrait (frappe de la touche <Enter> ou frappe de n'importe quelle touche du clavier) des procédures définies en C (ou dans un autre langage) ;

- soit, dans le cas où aucun callback n'est prévu pour un certain évènement, on modifie directement la table de traduction (translation table) en remplaçant l'action normalement associée à cet évènement par une autre action ; cette table de traduction est une table d'association, au fonctionnement très semblable aux listes d'associations vues en Programmation fonctionnelle.

Mais nous allons d'abord voir comment construire des widgets.

3 - Inclusion de Motif

La première chose à savoir commence par les bibliothèques à inclure. Comme vous le savez, tout programme C (ou C++) commence par inclure les bibliothèques standard, appelées des headers (.h), contenant en code source les prototypes (ou déclarations) de fonctions et de données dont il a besoin, par exemple :

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

Mais les bibliothèques openmotif ne figurent pas dans les bibliothèques standard. Il faut donc les installer ; il faut également installer les bibliothèques compilées correspondantes (l'équivalent sous Linux des .dll de MsWindows), qui contiennent les définitions des fonctions et des variables globales à utiliser. Vous devrez donc installer libopenmotif (chez moi libopenmotif4), l'ensemble des bibliothèques compilées, et libopenmotif-devel (chez moi libopenmotif4-devel), qui contient l'ensemble des headers. Les headers sont destinés à être inclus dans votre programme par le préprocesseur, les bibliothèques compilées sont destinées à être linkées avec votre code une fois compilé.

Voici ce que doit contenir le début du programme (après l'inclusion des bibliothèques standard) :

```
// pour openmotif
#include <Xm/Xm.h>              // la base
#include <Xm/Text.h>            // pour les widgets de type texte
#include <Xm/MainW.h>           // fenêtres principales
#include <Xm/CascadeB.h>        // cascade button
#include <Xm/RowColumn.h>       // menu bar
```

On pourra rajouter d'autres bibliothèques pour pouvoir utiliser d'autres types de widget. Pour compiler notre programme, il faut utiliser la ligne de commande suivante (éventuellement adaptée si chez vous les bibliothèques ne sont pas au même endroit ou si vous changez le nom des programmes), qui indique au compilateur où se trouvent les bibliothèques compilées :

```
g++ monprog.cpp -o monprog -Wall -L/usr/X11R6/lib -lX11 -lXm
```

Les headers de X11, utilisés par Xm, sont normalement déjà installés, mais si ce n'est pas le cas, vous devrez les installer.

J'en profite pour indiquer que vous devez créer votre propre header, contenant les prototypes de vos fonctions et de vos structures de données, pour alléger la lecture de votre fichier, et l'inclure après l'inclusion de tous les autres headers, sous la forme suivante :

```
#include "monheader.h"
```

Les guillemets indiquent au préprocesseur qu'il trouvera le header en partant du répertoire où se trouve le fichier à compiler.

4 - Implémentation de la fenêtre

Motif (comme d'ailleurs Athena) utilise X11 et également X Toolkit Intrinsics (Xt), ou simplement Intrinsics, une bibliothèque basique de widgets, déjà inclue. Les fonctions de X commencent toutes par X, les fonctions de Xt commencent par Xt, et les fonctions de Motif (Xm) commencent par Xm. Les classes de Motif commencent par xm.

Dans le main, grâce à Xt, on commence par déclarer notre interface comme une application X :

```
XtAppContext interface ;
```

Puis on crée un widget maître, grâce à la fonction suivante qui crée une connexion entre le serveur X et notre interface. On lui donne comme arguments l'adresse de notre interface, un nom pour sa classe (permet de trouver le fichier de ressources), NULL et 0 (arguments spécifiques à X qu'on pourrait trouver sur la ligne de commande et leur nombre), le nombre d'arguments de la ligne de commande restants et le tableau de ces arguments, NULL (un fichier de ressources de secours), et NULL pour indiquer qu'il n'y a pas d'arguments optionnels. Nous n'aurons besoin que des deux premiers arguments, mais il faut tous les remplir.

```
Widget top_widget = XtVaAppInitialize(&interface, "Interlisp", NULL, 0, &argc, argv, NULL,
NULL) ;
```

On va associer maintenant à ce widget Intrinsics un widget Motif qui va en hériter (Managed), c'est à dire que sa taille et sa place seront les mêmes. Ce qu'apporte ce widget, c'est un certain nombre de ressources pré-programmées plus sophistiquées et ayant le look and feel Motif ; il appartient à la classe des "fenêtres principales" (MainWindow), d'usage fortement conseillé (les autres widgets Motif lui seront accrochés). Le NULL final marque la fin des arguments de la fonction.

```
Widget main_window = XtVaCreateManagedWidget("main_window",
xmMainWindowWidgetClass, top_widget, NULL) ;
```

Cette fenêtre principale contiendra une barre de titre (avec le nom de l'application) et les habituels indicateurs de fermeture, d'agrandissement et de réduction. On va maintenant lui associer une barre de menu, puis on va s'arranger pour que cette barre s'aligne (taille et position) sur son parent (main_window) :

```
Widget menu_bar = XmCreateMenuBar(main_window, (String) "main_list", NULL, 0) ;
XtManageChild(menu_bar) ;
```

String étant défini par :

```
typedef char * String ;
```

Ce cast évite les warning de g++.

Sur cette barre de menu on va placer un bouton pour quitter l'application (le premier argument sera le texte affiché sur le bouton) :

```
Widget quit = XtVaCreateManagedWidget("Quit", xmCascadeButtonWidgetClass,
menu_bar, NULL) ;
```

Enfin, on va créer une zone de saisie de texte, intégrant le scroll (avec barres de scroll), qui sera alignée (largeur et position) sur son parent (main_window) :

```
Widget saisie = XmCreateScrolledText(main_window, (String) "saisie", NULL, 0) ;
XtManageChild(saisie) ;
```

Si on ne déclare par comme Managed les widgets accrochés, ils seront indépendants : déplacer la fenêtre principale ne déplacera pas la barre de menu ou la zone de saisie, l'agrandir ne changera pas leur taille.

Nous aurons besoin de référer, dans le main, à top_widget et à interface, qui doivent donc y être déclarés ; le reste peut être mis dans une fonction, dont voici une première esquisse :

```
void initMotifWidgets(Widget top, XtAppContext app)
{ Widget main_window = XtVaCreateManagedWidget("main_window",
xmMainWindowWidgetClass, top, NULL) ;

  Widget menu_bar = XmCreateMenuBar(main_window, (String) "main_list", NULL, 0) ;
  XtManageChild(menu_bar) ;

  Widget quit = XtVaCreateManagedWidget("Quit", xmCascadeButtonWidgetClass, menu_bar,
NULL);

  Widget saisie = XmCreateScrolledText(main_window, (String) "saisie", NULL, 0) ;
  XtManageChild(saisie) ;
}
```

Et le main (dans le module principal) contient :

```
int main(int argc, String argv[])
{ XtAppContext interface ;

  Widget top_widget = XtVaAppInitialize(&interface, "Interlisp", NULL, 0, &argc, argv, NULL,
NULL) ;
  initMotifWidgets(top_widget, interface) ;

  // affiche les widgets
```

```
  XtRealizeWidget(top_widget) ;
  // boucle principale
  XtAppMainLoop(interface) ;

  return 0 ;
}
```

Evidemment à ce stade le programme ne fait pas grand chose ; on pourra constater qu'il affiche une très petite zone de saisie (qu'on peut toutefois agrandir à la souris), et qu'on ne peut pas y faire grand chose. Mais l'essentiel de la structure est là.

5 - Ressources des widgets

Les deux derniers arguments des fonctions XmCreate..., que nous avons mis respectivement à NULL et à 0, vont nous permettre d'améliorer considérablement le fonctionnement de nos widgets, et en particulier du widget texte (appelé ici saisie). L'avant dernier argument indique le tableau des ressources dont dispose le widget, et le dernier leur nombre ; chaque type de widget peut avoir toutes sortes de ressources ; celles qui nous seront utiles ici sont regroupées dans cette fonction :

```
// initialiser le tableau des ressources pour le widget texte
int initSetArg(Arg args[])
{ int n = 0 ;
   XtSetArg(args[n], XmNrows, 30) ;                          // longeur initiale
   n++ ;
   XtSetArg(args[n], XmNcolumns, 80) ;                        // largeur initiale
  n++ ;
  XtSetArg(args[n], XmNeditable, True) ;                      // le texte est modifiable
  n++ ;
  XtSetArg(args[n], XmNcursorPositionVisible, True) ;         // le curseur est visible
  n++ ;
  XtSetArg(args[n], XmNeditMode, XmMULTI_LINE_EDIT) ;  // le widget comprend plusieurs
lignes
  return n + 1 ;
}
```

Donc, en modifiant comme suit notre fonction, la zone de saisie va déjà être plus adaptée :

```
void initMotifWidgets(Widget top, XtAppContext app)
{ Widget main_window = XtVaCreateManagedWidget("main_window",
xmMainWindowWidgetClass, top, NULL) ;

  Widget menu_bar = XmCreateMenuBar(main_window, (String) "main_list", NULL, 0) ;
  XtManageChild(menu_bar) ;

  Widget quit = XtVaCreateManagedWidget("Quit", xmCascadeButtonWidgetClass, menu_bar,
NULL);

  // créer la zone de saisie et ses ressources
  Arg args[10]
  int n = initSetArgs(args) ;
  Widget saisie = XmCreateScrolledText(main_window, (String) "saisie", args, n) ;
  XtManageChild(saisie) ;
}
```

Testez... et voyez. Le widget de classe ScrolledText contient un véritable petit éditeur de texte tout fait...

6 - Le mécanisme des callbacks

Exemple simple : nous aimerions bien que quand on clique sur le bouton "Quit", l'application se ferme. Il existe pour cela un callback prévu, tout simple, XmNactivateCallback, qui s'exécute chaque fois qu'on clique sur un widget ; il suffit d'écrire :

```
XtAddCallback(quit, XmNactivateCallback, quit_call, NULL) ;
```

Et quit_call est la fonction qui sera activée ; il reste à l'écrire :

```
void quit_call(Widget W, XtPointer Ev, XtPointer client)
{ printf("Quitting program\n"); exit(0); }
```

Notez que le message "Quitting program" s'affiche dans la console d'où ce programme a été appelé.

Le prototype de toutes les fonctions de callback doit toujours être celui-ci, même si, comme ici, nous n'utilisons pas ses arguments :

```
void fonction(Widget, XtPointer, XtPointer) :
```

7 - La table de traduction

Nous avons besoin que chaque fois que l'utilisateur tape un retour-charriot (entrée), le programme puisse répondre. Or actuellement dans notre fenêtre, taper un retour-charriot insère un retour-charriot. Aucun callback n'est prévu pour cela avec les widgets XmText multilignes comme le nôtre. Il en existe avec les widgets XmTextField, mais ceux-ci ne comprennent qu'une seule ligne, et ça ne conviendrait pas à l'interprète que nous allons développer dans la partie suivante. Une seule solution est possible : modifier la table de traduction de notre widget de saisie. Ce mécanisme est offert par Intrinsics ; la traduction se déroule en deux temps : quand un évènement s'est produit, Xt regarde dans la table de traduction le nom de l'action correspondante ; ensuite Xt regarde dans la table des actions à quelle fonction ce nom correspond, pour l'appliquer.

Par conséquent, il faut d'abord créer une table où chaque nom d'action (une chaîne) est associé à une fonction (ici la table ne contient qu'un élément), et ajouter cette table à la table des actions de l'application :

```
static XtActionsRec actions[] = {{(String) "Done", Done}} ;
XtAppAddActions(app, actions, XtNumber(actions)) ;
```

Dans un deuxième temps, on va créer une chaîne de caractères constante associant l'évènement retour-charriot à ce nom d'action, puis on va la compiler dans une table de traduction qu'on va intégrer à notre widget saisie avec priorité (override) sur l'association originelle (qui n'est pas détruite) :

```
static ConstantString traduction = "<Key>Return: Done()" ;
XtTranslations mytranslations = XtParseTranslationTable(traduction) ;
XtOverrideTranslations(saisie, mytranslations) ;
```

Tout ça est à ajouter dans la fonction initMotifWidgets. Et voilà, désormais taper Entrée ou Retour déclenchera la fonction Done, qu'il ne reste plus qu'à définir ; c'est elle qui appellera l'interprète Lisp (ou n'importe quel autre) pour qu'il analyse les données entrées.

8 - Exercice : intégrer l'interface graphique à votre programme

Reprenez votre programme sur le dictionnaire et, au lieu d'utiliser scanf et printf ou puts et gets, écrivez et lisez à l'aide de l'interface graphique. Vous pouvez vous inspirer du programme final de l'interprète Lisp, qui se trouve dans Documents et Liens, sous le nom interlisp.tar.gz, en modifiant la fonction Done. Vous êtes concernés par les fichiers widget.cpp, widget.h, commode.h et interlisp.cpp. Mettez dans un seul fichier le contenu d'interlisp.cpp et de widget.cpp, et incluez widget.h et commode.h. Enlevez tout ce qui commence par extern (explication à la partie suivante).

Envoyez votre code avec comme sujet "Exo Listes 7 Graphique".

Explorer un mini-interprète

Nous allons maintenant écrire en C un interprète lisp simple, ce qui ne sera pas si simple. Le programme sera assez complexe, il va donc falloir le découper en un certain nombre de modules. Je présente rapidement le principe de la construction modulaire, puis l'interprète lisp dans ses différentes parties. Le but de l'exercice final est que vous compreniez les structures de contrôle (la structure throw-catch) et les structures de données (tables d'association) employées ici, et non que vous sachiez les reproduire ; c'est dans le cours suivant (Structures de données et algorithmes 2) que vous aurez à les utiliser vous-même ; dans ce cours-ci je vous donne le programme tout fait, vous n'aurez que des modifications à y faire.

I - Construire un programme modulaire

Tout d'abord, comme je l'ai déjà indiqué, on découpe en général un programme non modulaire en un fichier source et un fichier header :

- le fichier source (.c ou .cpp) contient les définitions de fonctions et de variables globales, et l'inclusion de son header ci-dessous ;
- le fichier header (.h) contient les prototypes de fonctions, les prototypes de structures, les types de données, les inclusions de fichiers des bibliothèques standard, les variables et les macros pour le préprocesseur (#define).

Ce qui caractérise un programme modulaire, c'est qu'il contient plusieurs fichiers sources ; en général chacun dispose de son propre header. Chacun des fichiers sources contient les définitions propres à une sous-partie clairement délimitée (interface graphique, interprète lisp), sauf celui qui contient le main, qui, par définition, utilise toutes les sous-parties.

Quand plusieurs fichiers sources ont en commun la même variable globale (par exemple, nil), elle est déclarée et définie dans l'un des modules (par exemple, interlisp) et redéclarée comme extern dans les autres :

```
extern objet nil ;
```

Ce qui signifie que nil est un objet dont la définition est faite dans un des autres modules. Ceci est bien entendu inutile quand il n'y a qu'un seul fichier source.

Chaque fichier source doit être compilé à part, et l'ensemble des fichiers compilés doit ensuite être assemblé (linké) pour constituer l'exécutable final. Comme cette procédure est assez complexe, au lieu d'appeler directement le compilateur pour compiler le programme, ce que vous avez fait jusqu'à maintenant, on construit un fichier appelé Makefile qui contient tous les appels au compilateur. Il suffit ensuite d'utiliser la commande make dans le répertoire où figure le fichier Makefile, et l'exécutable est constitué. Il existe d'autres procédures plus complexes (install / make / make install), que vous avez sans doute déjà utilisées pour installer des programmes qui ne sont pas constitués en packages (.rpm ou .deb).

On ne peut en général pas définir dès le début tous les modules qui composeront le programme ; on commence donc par faire un programme unique, puis, dès qu'il devient difficile à gérer, on le décompose, et ainsi de suite. Dans le problème qui nous occupe, on peut prévoir qu'il nous faudra trois modules (interface, lisp, et main) ; mais en réalité, dans le programme présenté, j'ai fini par

créer les cinq modules suivants :

- le module main, appelé interlisp
- un header utilisé par tous les modules, appelé commode
- un module widget (avec son header), contenant la gestion de l'interface graphique
- un module lisp (avec son header), contenant l'interprète lisp
- un module garbage (avec son header), pour nettoyer les objets créés
- un module primf (avec son header), contenant toutes les fonctions utilisées par Lisp.

II - L'interprète

Un interprète est un programme qui réagit immédiatement à chaque commande de l'utilisateur par une réponse appropriée et redonne la main à l'utilisateur pour qu'il puisse formuler une nouvelle commande. Vous connaissez, comme interprètes, le shell, python, lisp, squeak (au moins) ; votre programme sur le dictionnaire était un interprète puisqu'à chaque mot entré il vous donne la traduction puis vous redonne la main. Chaque widget peut aussi être considéré comme un interprète ; d'une manière générale, tout objet d'un environnement objet (et un widget est un objet) est un interprète.

Les interprètes sont fondés sur trois fonctions fondamentales, exécutées en boucle :

- read (lis l'entrée, supposée être interprétable) ;
- eval (évalue l'entrée et donc exécute la commande si c'est possible) ;
- print (affiche la sortie).

Dans le programme que vous avez réalisé, sur le dictionnaire, read était un simple scanf (ou gets), eval allait chercher la traduction, et print était un simple printf (ou puts). Dans ce qui suit je montre comment écrire les fonctions read et print pour qu'elles fonctionnent comme dans un interprète Lisp. Mais tout d'abord on va s'intéresser à la manière de structurer les données Lisp, qui se trouve dans le header lisp.h.

3 - Lisp.h : l'objet, structure fondamentale

Nous avons déjà vu comment implémenter en C les listes :

```
typedef struct Doublet {void * car ; struct Doublet * cdr ;} doublet ;
typedef doublet * list ;
```

Les atomes numériques et les chaînes ne posent pas de problème particulier : on utilisera les types numériques et le type char *, par exemple :

```
typedef long int * entier ;
typedef char * String ;
```

Pour les symboles, ce n'est pas beaucoup plus compliqué, il suffit de construire une structure avec un nom et trois valeurs, une valeur comme donnée, une valeur comme liste de propriétés, une valeur en tant que fonction.

```
typedef struct {char * pname ; void * cval ; void * props ; void * fval ;} symbole ;
typedef symbole * symbol ;
```

Mais ici nous avons un double problème : d'une part nous ne saurons pas, depuis notre programme,

ce que l'utilisateur va entrer (liste, atome numérique, chaîne, symbole) ; d'autre part nos fonctions (par exemple eval ou print) doivent pouvoir traiter aussi bien des listes que des atomes de n'importe quel type. Il nous faut donc définir une structure qui puisse recouvrir tous les types d'objets que l'utilisateur pourra entrer. Appelons-la l'objet.

Il y a deux façons standard d'implémenter en C les objets Lisp. Tout d'abord, on peut donner la même structure aux différents types d'objet (listes, symboles, nombres, chaînes, etc) ; prenons par exemple les structures de données les plus complexes, le doublet et le symbole :

```
typedef struct Doublet {void * car ; struct Doublet * cdr ;} doublet ;
typedef struct {char * pname ; void * cval ; void * props ; void * fval ;} symbole ;
```

Si on les identifie, on peut avoir :

```
typedef struct Objet {char * pname ; struct Objet * car_or_cval ; struct Objet * cdr_or_props ;
struct Objet * fval} objet ;
```

La seule différence entre doublet et symbole sera dans l'utilisation des parties de la structure : on n'utilisera pas le pname (print-name) pour un doublet, car_or_cval désignera la valeur d'un symbole ou le car d'une liste, cdr_or_props désignera la liste de propriétés d'un symbole et le cdr d'une liste, et on n'utilisera pas fval (valeur fonctionnelle) pour un doublet. Mais il y a dans ce cas beaucoup de place perdue ; même un atome numérique devra avoir cette structure, dont il n'utilisera rien (et il faudra y ajouter sa valeur). Il y a des astuces pour mieux utiliser cette place, mais elles compliquent la représentation.

Une autre solution consiste à faire précéder chaque structure d'un en-tête qui indique comment traiter la structure qui suit. Du même coup, la place occupée par un objet est variable, et il faut regarder l'en-tête pour savoir comment traiter l'objet. C'est la solution choisie ici.

Nous avons deux façons d'implémenter l'en-tête : soit on prend une structure de type enum, qui occupe un entier :

```
typedef enum {CONS, SYMB, ...} Type ;
```

Soit on prend un caractère (un octet), ce qui est largement suffisant (il y a moins de 128 types d'objets) :

```
typedef unsigned char type ;
```

Et on utilise des #define pour les valeurs de type possibles :

```
#define CONS 0
#define SYMB 1
```

Etc. J'ai donc pris cette dernière manière.

L'allocation mémoire s'effectuera en ajoutant la taille du char, appelée ci-dessous JMP, à la taille de la structure à créer :

```
#define JMP sizeof(char)
```

D'où la fonction :

```
objet alloc_objet(size_t size) {return (objet) malloc(size + JMP) ;}
```

Par exemple, pour allouer la mémoire nécessaire à un symbole :

```
alloc_objet(sizeof(symbole)) ;
```

Un objet est un pointeur sur une telle structure ; comme cette structure commence par un caractère, un objet est donc un pointeur sur un caractère :

```
typedef type * objet ;
```

A ne pas confondre avec une chaîne de caractères (les deux sont définis comme char *).

Pour accéder au type, il suffit de prendre la valeur de l'objet ; pour accéder à l'objet lui-même il faut sauter par-dessus le type, et caster en fonction du type :

```
objet X ;
* X = CONS ;
(list) (X + JMP)->car = ... ;
```

En testant donc d'abord (* X) on saura en quoi il faut caster (X + JMP) pour pouvoir le traiter...

Evidemment, pour se simplifier la vie, on a tout intérêt à utiliser des macros :

```
#define List(objet) ((list) (objet + JMP))
#define Car(objet) (List(objet)->car)
```

J'ai utilisé par ailleurs cet en-tête pour d'autres usages, par exemple pour pouvoir lire aussi bien dans un fichier (FILE *) que dans la chaîne (char *) renvoyée par un callback...

4 - Lisp.cpp : print et l'usage de l'en-tête

La fonction prin1(), qui est la principale fonction d'affichage (print() se contente d'appeler prin1() en ajoutant un newline et un espace), affiche soit dans un widget soit dans un fichier. Elle est elle aussi fort simple, et illustre l'emploi de l'en-tête pour traiter l'objet :

```
objet prin1 (objet obj, objet output)
{ if (Consp(obj)) print_list(obj, output) ;
  else print_atom(obj, output) ;
  return obj ;
}
```

Consp() est une macro qui teste si l'objet est un CONS, comme toutes les macros se terminant par la lettre p (p comme predicate), Nilp(), Symbolp(), etc. Voici la fonction print_list(), qui imprime une liste :

```
void print_list(objet obj, objet output, char firsttime)
{ if (firsttime)
```

```
 { emitprint((String) "(", output) ;              // au début, imprime une parenthèse
   prin1(Car(obj), output) ;                      // imprime le car
   print_list(Cdr(obj), output, FALSE) ; }    // et récurse sachant qu'on n'est plus au début
 elseif (not Listp(obj))                          // terminé par atome autre que nil
 { emitprint((String) " . ", output) ;            // imprime "."
   prin1(obj, output) ;                           // l'atome
   emitprint((String) ")", output) ; }            // et la fermante
 elseif (Nilp(obj)) emitprint((String) ")", output) ; // nil : on ferme
 else
 { emitprint((String) " ", output) ;              // sinon espace
   prin1(Car(obj), output) ;                      // imprimer le car
   print_list(Cdr(obj), output, FALSE) ; }    // et récurse sachant qu'on n'est pas au début
}
```

Et la fonction print_atom() utilise la fonction string_atom() pour savoir, en fonction du type d'atome, quelle est la chaîne qu'elle doit imprimer (nombre, chaîne, symbole).

```
String string_atom(objet obj)
{ switch (objet_type(obj))
  { case NIL :
    case SYMB : return String(Pname(obj)) ;
    case NUM : return string_number(obj) ;
    case STRING : return String(obj) ;
    default : return (String) "" ; }          // on ne devrait pas passer par là...
}

String string_number(objet N)
{ static String S = (String) malloc(BUFSIZ) ;
  sprintf(S, "%ld", Number(N)) ;
  return S ;
}
```

5 - Lisp.cpp : read et la structure de contrôle throw-catch

La fonction de lecture read() doit pouvoir lire à partir de n'importe quel point d'entrée, que ce soit la chaîne prise dans le widget saisie, ou que ce soit un fichier ; pour cela, j'ai défini un objet, Current_Input, qui est le périphérique d'entrée par défaut, et qui peut être soit un fichier, soit une chaîne quand on est dans l'action Done() ; le prototype de la fonction est :

```
objet read(objet = Current_Input) ;
```

Donc read() lit un objet et renvoie un objet ; voici sa définition :

```
objet read(objet input)
{ char * peek_char = peek(input) ;
  if (* peek_char == RPARENT) throw TOOMUCHPARENT ;
  if (* peek_char == LPARENT)  {getchar(input) ; return read_list(input) ;}
  return read_atom(input) ;
}
```

On a défini :

```
#define RPARENT ')'
```

```
#define LPARENT '('
```

```
#define TOOMUCHPARENT 4
```

La fonction peek() regarde quel est le prochain caractère, mais sans avancer dans la chaîne d'entrée ; appeler deux fois de suite peek() donnera le même résultat ; en revanche, getchar() lit le prochain caractère et avance ; appeler deux fois de suite getchar() nous donnera les deux caractères suivants.

Ces fonctions renvoient non le caractère mais un pointeur sur ce caractère, pour pouvoir faire la différence entre NULL (plus de caractères) et '\0' (caractère nul). A ne pas confondre avec l'objet (un pointeur sur un caractère suivi de l'objet lui-même) et la chaîne (un pointeur sur un caractère suivi d'autres caractères), même si tous ont la même déclaration.

La seule subtilité, dans ce read(), réside dans l'emploi de throw ; comme return, throw est un mot-clé qui indique qu'il faut terminer la fonction ; mais alors qu'avec return, le contrôle est rendu à la fonction appelante, avec throw il est rendu, en remontant la pile des appels, à la première fonction qui contient un try - catch compatible avec la valeur du throw (ici TOOMUCHPARENT, qui est un entier compatible avec "reason").

La structure de contrôle throw-catch (jette et attrape) permet d'échapper à l'enchâssement ordinaire des fonctions. Ici, l'idée c'est que si read() rencontre une parenthèse fermante alors qu'il n'y a aucune ouvrante, il ne faut même pas essayer d'aller plus loin, et le catch présent dans la fonction Done() renverra le contrôle au widget saisie, après avoir affiché un message d'erreur dans stdout. Voici ce que contient cette fonction :

```
try {Widget_read_eval_print(line) ;}
  catch(int reason) {fprintf(stdout, "Attention : %s\n", Throw[reason]) ; return ;} ;
```

Done() appelle Widget_read_eval_print() qui à son tour appelle la fonction read() ; le contrôle, si un throw est rencontré, est rendu à Done() et non à Widget_read_eval_print(). Le tableau Throw[] contient l'ensemble des messages d'erreurs définis.

Dans read_list(), appelé par read(), il y a plusieurs autres throw, activés quand par exemple il manque une parenthèse fermante : le contrôle n'est alors pas rendu à read(), qui a pourtant appelé read_list(), mais au même catch de Done().

6 - Lisp.cpp : read_atom et la table d'association des symboles

La fonction read_atom(), appelée par read(), utilise une variable globale, appelée Oblist, déclarée sous cette forme :

```
extern objet * Oblist ;
```

Cette variable, comme toutes les autres globales, est initialisée dans le fichier source principal (interlisp.cpp), par appel à la fonction initLisp(), de la façon suivante :

```
Oblist = (objet *) calloc(MAXOB, sizeof(objet)) ;
```

Où MAXOB désigne le nombre maximum de symboles qu'on peut définir (défini dans lisp.h). Chaque fois qu'un symbole est défini, on l'insère dans Oblist grâce à la fonction put_symb(), et la variable globale Obindex contient le nombre de symboles définis :

```
void put_symb(objet obj) {Oblist[Obindex++] = obj ;}
```

La fonction read_atom(), quand elle lit un mot qui n'est pas une chaîne ou un nombre, et qui est donc un symbole, appelle la fonction find_symb() pour savoir s'il est déjà défini, en parcourant la table Oblist et en comparant le pname de chaque symbole au mot lu.

```
objet find_symb(String S)
{ for (unsigned long int i = 0 ; i < Obindex ; i++)
  { if (not Oblist[i]) continue ;
    if (not strcmp(String(Pname(Oblist[i])), S)) return Oblist[i] ;
  }
  return NULL ;
}
```

Cette fonction se comporte donc tout à fait comme la fonction assoc de Lisp, à ceci près qu'elle parcourt une table et non une liste. Pname(x) est une macro qui accède au champ pname d'un symbole. Si elle renvoie NULL, c'est que l'atome n'est pas défini, et read_atom() fera dans ce cas appel à put_symb() pour ajouter le nouvel atome (par l'intermédiaire de make_symb(), qui effectue d'autres opérations comme la réservation de la place pour le symbole).

7 - Lisp.cpp : eval et les atomes

Voici le code de la fonction eval()

```
objet eval(objet obj)
{ gc_garbage() ;
  switch(objet_type(obj))
  { case CONS : return eval_list(obj) ;
    case SYMB : return Cval(obj) ;
    default : return obj ; }
}
```

L'appel à la fonction gc_garbage() n'est pas au programme ; cette fonction a pour but de nettoyer tous les objets créés (listes, symboles, chaînes, nombres) qui ne sont pas utilisés. Elle utilise les variables globales GcTemp et GcAllobjects, et l'ensemble des fonctions de garbage collecting (qui commencent par gc_) est inclu dans le fichier garbage.cpp et son header garbage.h, que vous pouvez donc ignorer totalement. Ils sont simplement fournis pour que l'interprète n'explose pas en ayant utilisé toute la mémoire disponible.

Vous voyez que l'évaluation d'un symbole est toute simple : on renvoie la cval de l'objet ; pour les nombres et les chaînes, c'est encore plus simple : on les retourne directement. J'ai choisi ici l'autoquote comme valeur d'un symbole non défini (il est sa propre valeur), il n'y aura donc aucun message d'erreur, tout symbole est défini dans make_symb(), au moment de la lecture du symbole. Voici le code de make_symb(), expurgé de tout ce qui concerne le garbage.

```
objet make_symb(String S)
{ objet pname = alloc_objet(strlen(S) + 1) ;              // allocation de l'objet chaîne (S est le
mot lu)
  objet_type(pname) = STRING ;                            // on lui met son type
  strcpy(String(pname), S) ;                              // on copie le mot lu dans l'objet
  objet new_objet = alloc_objet(sizeof(Symbole)) ;        // allocation du symbole
  objet_type(new_objet) = SYMB ;                          //  on lui met son type
```

```
 Pname(new_objet) = pname ;                    // on lui met son nom
 Cval(new_objet) = new_objet ;                 // il est lui-même sa propre valeur
 Fval(new_objet) = NULL ;                      // pointeur de fonction initialisé à NULL
 put_symb(new_objet) ;                         // on le met dans Oblist
 return new_objet ;
}
```

Avant de traiter le cas des listes, je vais revenir sur les pointeurs.

8 - Les pointeurs de fonction

Tout d'abord, déclarons et définissons un entier :

```
int entier ;                    // prototypage
entier = 3 ;                    // définition
```

Pour déclarer un pointeur sur cet entier, on écrit :

```
int * Ptr_entier ;              // prototypage de "Objet"
```

Ce qui revient à remplacer entier par (* Ptr_entier) dans la déclaration de entier (les parenthèses sont inutiles). Ensuite nous pouvons faire pointer Ptr_entier sur entier, de la façon suivante :

```
Ptr_entier = &entier ;
```

Et on peut accéder à la valeur de entier par le pointeur en utilisant :

```
* Ptr_entier ;
```

Et bien, pour une fonction, les choses se passent de manière très semblable. Par exemple, déclarons et définissons une fonction :

```
objet fonction(objet) ;                 // prototypage
objet fonction(objet) {return objet ;}  // définition
```

Si on veut définir un pointeur sur cette fonction, il faut remplacer fonction par (* Ptr_fonction) ; ici les parenthèses sont obligatoires, sinon il y aurait une ambiguité :

```
objet (* Ptr_fonction)(objet) ;
```

En toute logique, on écrit ensuite :

```
Ptr_fonction = &fonction ;
```

Mais ici, comme pour les tableaux, l'opérateur & ne sert à rien (fonction est déjà une référence) ; on peut aussi écrire :

```
Ptr_fonction = fonction ;
```

L'expression (* Ptr_fonction) permet d'accéder à la valeur de la fonction ; il suffit de lui rajouter des arguments (du bon type) pour pouvoir l'exécuter :

```
(* Ptr_fonction)(obj) ;
```

Pour définir le type pointeur sur un entier, on écrit :

```
typedef int * Ptr_entier ;
```

De même, ici, j'ai défini le type primitive comme un pointeur sur une fonction recevant un objet et renvoyant un objet :

```
typedef objet (* primitive)(objet) ;
```

Pour déclarer, définir et exécuter une primitive, on écrit :

```
primitive fonction ;          // déclaration de fonction comme pointeur sur une primitive
fonction = lisp_quote ;       // définition de fonction comme pointant sur lisp_quote
(* fonction)(obj) ;           // exécution de la fonction = exécution de lisp_quote
fonction = lisp_car ;         // on change la valeur de fonction, qui pointe maintenant sur lisp_car
(* fonction)(obj) ;           // exécution de la fonction = exécution de lisp_car
```

Evidemment, cela suppose que toutes les fonctions primitives devront avoir comme seul argument un objet et renverront un objet.

9 - Lisp.cpp : eval et l'application de fonctions

Dans l'évaluation de listes, le premier élément est le nom d'une fonction ; voici le code d'eval_list().

```
objet eval_list(objet obj)
{ if (not Symbolp(Car(obj))) return nil ;
  if (not Fval(Car(obj))) return nil ;
  switch (objet_type(Fval(Car(obj))))
    { case NPRIMF :    return Primitive(Fval(Car(obj)))(Cdr(obj)) ;
      case PRIMF : return Primitive(Fval(Car(obj)))(eval_cdr(Cdr(obj))) ;
      default : return nil ; }
}
```

Si le premier élément n'est pas un symbole (chaîne, nombre ou liste), eval_list() renvoie nil ; si c'est bien un symbole, mais qu'il n'a pas de fval (valeur fonctionnelle), il renvoie nil - il n'y a donc aucun message d'erreur du type fonction non définie.

Pour le reste : on manipule en Lisp deux types de fonctions : celles qui sont prédéfinies, dites primitives, et celles qui sont définies en Lisp par la fonction defun ou un de ses équivalents. Dans mon Lisp primitif, seules les primitives sont possibles. Elles se subdivisent en deux types : celles qui n'évaluent pas leurs arguments, comme quote, et celles qui les évaluent, comme car et cdr.

Je vais détailler le premier cas du switch :

```
Primitive(Fval(Car(obj)))(Cdr(obj)) ;
```

Fval(Car(obj)) renvoie l'objet qui est la fonction associée au symbole Car(obj) (le premier élément de la liste). Comme tous les objets, il est produit par la concaténation d'un en-tête et d'un objet, défini comme une primitive (section précédente). Cet en-tête, pour une primitive, ne peut être que NPRIMF (n'évalue pas ses arguments) ou PRIMF (les évalue).

Primitive est une macro qui nous donne la valeur associée à l'objet, donc la fonction. Il suffit de lui ajouter des arguments pour exécuter la fonction qui a été stockée là. Par conséquent le code ci-dessus exécute la fonction dont l'adresse a été stockée dans le symbole. Ce stockage s'effectue par la fonction make_prim() :

```
objet make_prim(String S, primitive fun, type T)
{ objet symbole = find_symb(S) ;
  if (not symbole) symbole = make_symb(S) ;       // si le symbole n'existe pas, le créer
  objet function = alloc_objet(sizeof(primitive)) ;    // allouer l'objet de type fonction
  objet_type(function) = T ;                            // y mettre son type
  Primitive(function) = fun ;                           // y stocker la fonction C
  Fval(symbole) = function ;                            // mettre l'objet fonction dans le symbole
  return symbole ;
}
```

Comme vous pouvez le voir dans le code de eval_list(), la seule différence entre NPRIMF et PRIMF est qu'on utilise dans le dernier cas la fonction eval_cdr(), qui va évaluer récursivement les arguments de la fonction.

```
objet eval_cdr(objet obj)
{ if (not Consp(obj)) return eval(obj) ;
  return cons(eval(Car(obj)), eval_cdr(Cdr(obj))) ;
}
```

Voici, pris dans initLisp, l'internement de la fonction lisp_car :

```
make_prim((String) "car", lisp_car, PRIMF) ;
```

Dès lors qu'on tapera "car", la fonction lisp_car() sera exécutée. Cette fonction est définie dans primf.cpp (header primf.h), comme toutes les autres primitives Lisp, de la façon suivante :

```
objet lisp_car(objet O)
{ if (not Consp(O)) return nil ;
  if (not Consp(Car(O))) return nil ;
  return Car(Car(O)) ;
}
```

Les primitives reçoivent comme argument la liste de tous leurs arguments (après évaluation éventuellement). Pour définir une nouvelle fonction Lisp, il faut donc d'abord l'ajouter dans primf.cpp (et sa déclaration dans primf.h), et l'interner dans la fonction initLisp() en lui attribuant le bon type, avec make_prim().

10 - Makefile : assemblage d'un programme

Voici le makefile permettant de construire notre programme d'un seul coup. Le début contient quelques variables qui indiquent, dans l'ordre : le compilateur à utiliser, le nom de l'exécutable, les flags pour linker l'interface graphique, les flags généraux, la liste des fichiers sources (l'ordre

compte), et OBJ, qui contient les fichiers objets produits à partir des sources (par remplacement de .cpp par .o)

```
CPP=g++
EXEC=interlisp
LDFLAGS= -lX11 -lXm
CXXFLAGS= -W -Wall
SRC= primf.cpp garbage.cpp lisp.cpp widget.cpp interlisp.cpp
OBJ=$(SRC:.cpp=.o)
```

Vient ensuite la liste des "cibles" : all avec ses sous-cibles, clean et cleanall. Pour exécuter toute autre cible que la première, il faut la donner en argument à make : make clean, make cleanall. Voyons d'abord la première :

```
all: $(EXEC)

$(EXEC): $(OBJ)
    $(CPP) -o $@ $^ $(LDFLAGS)

primf.o: primf.cpp primf.h lisp.h commode.h
    $(CPP) -o $@ -c $< $(CXXFLAGS)

garbage.o: garbage.cpp garbage.h lisp.h commode.h
    $(CPP) -o $@ -c $< $(CXXFLAGS)

lisp.o: lisp.cpp lisp.h primf.h garbage.h commode.h
    $(CPP) -o $@ -c $< $(CXXFLAGS)

widget.o: widget.cpp widget.h commode.h
    $(CPP) -o $@ -c $< $(CXXFLAGS)

interlisp.o: interlisp.cpp widget.h lisp.h garbage.h commode.h
    $(CPP) -o $@ -c $< $(CXXFLAGS)
```

Chaque cible est suivie de la liste de ses dépendances ; si on modifie l'une des dépendances, make reconstruira la cible qui en dépend. Par exemple, si on modifie garbage.h, make reconstruira garbage.o et interlisp.o, et donc aussi $(EXEC) puisque $(OBJ) contient garbage.o et interlisp.o (obtenus à partir de $(SRC) en remplaçant l'extension .cpp par l'extension .o). La commande make ne reconstruit que ce qui est strictement nécessaire.

Sous la liste des dépendances, figure l'invocation du compilateur $(CPP) avec les bons paramètres (je ne détaille pas, vous pouvez consulter le web pour ça).

Les dernières cibles sont destinées à supprimer les .o après compilation (clean) et à supprimer l'exécutable.

```
# pour forcer clean/cleanall même s'il existe un fichier cible de même nom
.PHONY: clean cleanall

clean:
rm -rf *.o

cleanall: clean
rm -rf $(EXEC)
```

11 - Exercices à faire avec le programme

L'ensemble du programme est disponible dans Documents et Liens, sous le nom interlisp.zip. Vous devez construire des primitives Lisp supplémentaires (à vous de choisir), trois en PRIMF et trois en NPRIMF, en modifiant primf.cpp et primf.h, ainsi que initLisp() dans lisp.cpp, et donner un exemple d'utilisation (en Lisp) de chaque primitive que vous avez créée, en indiquant étape par étape comment le programme C a pu aboutir au bon résultat. Renvoyer primf.cpp, primf.h et lisp.cpp sous forme de sources et l'exemple tracé dans un fichier PDF.

Envoyez-moi le tout avec en sujet du mail « Exo Listes 8 Lisp ».

Tables de hachage

Dans l'application précédente, pour trouver un symbole dans Oblist, on doit parcourir toute la table puisqu'on ne connait pas son rang dans la table. Une table de hachage permet justement de connaître à l'avance le rang d'un symbole dans la table, exactement ou approximativement. L'accélération de la recherche est considérable, surtout quand la table est grande. L'intérêt des tables de hachage est donc considérable, et elles sont utilisées partout (aussi, pour d'autres raisons, en cryptographie). Nous allons voir comment transformer notre Oblist en une telle table.

I - Principe du hachage

Le principe en est le suivant : le nom du symbole est utilisé comme adresse pour stocker le symbole dans la table. Pour simplifier, supposons que ces noms ne peuvent contenir que les caractères de 0 à 9 ; du coup le nom "5854" correspondra tout simplement à l'adresse 5854 en base 10. La base hexadécimale contient aussi les caractères de A à F ; donc, en choisissant cette base, le nom "FADE" correspondra à l'adresse FADE, en base 16. Pour que les noms puissent contenir tout l'alphabet, il nous faut prendre une base assez grande.

Le premier élément du hachage est donc la base dans laquelle les adresses sont calculées. La méthode de calcul du nombre associé à une chaîne est la suivante, si on prend la base 10 :

$$5854 = 5 * 10^3 + 8 * 10^2 + 5 * 10^1 + 4 * 10^0$$

(autrement dit, 5 mille 8 cent 5 dix 4 un - pour parler comme en chinois).

Avec une base quelconque, appelons-la Base (chaque lettre est un chiffre de la base), la méthode se généralise comme suit :

$$primf = p * Base^4 + r * Base^3 + i * Base^2 + m * Base^1 + f * Base^0$$

Il y a un deuxième problème : pour pouvoir contenir toutes les adresses possibles (tous les noms possibles), la taille de la table doit être pratiquement infinie ; d'autre part, comme on n'utilise pas tous les noms possibles, cette table est presque vide.

Pour résoudre ce problème, on fixe une taille, et on calcule l'adresse modulo cette taille. Par exemple, avec une taille de 12, pour stocker le symbole de nom "25", on effectue la division entière de 25 par 12 (ce qui donne 2) et on prend le reste (25 - 2 * 12 = 1), donc on stocke ce symbole à la deuxième place (0 est à la première). Plus simplement encore, c'est comme si la table était un cercle et qu'au bout de 12 tours on se retrouve au début. Cette solution introduit un autre problème, celui des collisions : quand deux symboles ont le même rang ("1" et "25" dans notre exemple). Généralement, plus la taille est grande, plus le nombre de collisions théoriques diminue.

Il existe une relation entre la base et la taille de la table qui fait que si la base et la taille ont des diviseurs en commun, le nombre de collisions augmente considérablement. L'exemple le plus simple est de prendre une base de même taille que la table ; prenons notre exemple ci-dessus : on constate que seul le dernier chiffre est pertinent :

```
primf = p * Base^4 + r * Base^3 + i * Base^2 + m * Base^1 + f * Base^0  = f
```

Trop compliqué ? Prenons l'autre exemple : 5850 est un multiple de 10, donc il sera rangé, dans une table de 10, à la première place. 5854 sera donc rangé à la cinquième place, comme tous les nombres terminés par quatre. Seul le dernier caractère compte. En prenant une table de 11, le risque de collisions diminue, non pas seulement parce que 11 est plus grand que 10, mais surtout parce que 10 et 11 sont premiers entre eux. Tous les multiples de 10 ne seront pas rangés à la même place.

Par conséquent, on choisit souvent un nombre premier pour la taille de la table, ce qui évite de se poser la question de la relation avec la base. D'autre part, on ne choisit pas forcément une base qui soit assez grande pour contenir tout l'alphabet, puisqu'il y aura de toutes façons des collisions. Le nombre de possibilités donne le vertige, mais je n'en dirai pas plus : la question des tables de hachage, si on la creuse du point de vue théorique, devient vite extrêmement sophistiquée et absconse, et nous entraînerait dans des mathématiques fort complexes.

Pour résoudre le problème des collisions, il existe plusieurs solutions ; je n'en exposerai ici qu'une seule : celle des listes chaînées. La table n'est plus une table de symboles, c'est une table de listes de symboles ; quand on a trouvé l'adresse dans la table, on parcourt la liste des symboles qui se trouve à cette adresse. Du coup, le temps de recherche est augmenté (uniquement en cas de collision), mais en tout état de cause il est nettement plus court qu'avec une table ordinaire. D'autre part, la table peut contenir un nombre quelconque de symboles, alors que notre Oblist ne pouvait contenir que MAXOB objets.

II - Structure de la table de hachage

Notre table de hachage (qu'on continuera à appeler Oblist) aura donc la structure suivante :

```
objet * Oblist ;
```

Surprise : il n'y a aucun changement. En fait, toute la différence sera dans la façon d'exploiter cette table ; au lieu de contenir des symboles, elle contiendra des listes de symboles.

On pourrait utiliser, pour gagner en temps, la structure de liste plate définie dans lisp.h et utilisée surtout dans garbage.cpp (pas d'en-tête, et pas de nil à la fin). Du coup, la table deviendrait :

```
flatlist * Oblist ;
```

III - Fonction de hachage

Le calcul de la clé de hachage c'est le calcul de l'adresse d'un symbole donné, d'après son nom. Il existe de nombreux algorithmes pour ce calcul, je vous en ai choisi un, l'algorithme de Daniel Bernstein (trouvé sur Wikipédia !) :

```
unsigned long int hash(String str)
{ unsigned long int hash = 5381 ;          // ou autre chose
  while (* str)
  { int c = * str ;
    hash = ((hash << 5) + hash) + c ; // hash = hash * 33 + c
    str++ ; }
  return hash ;
}
```

Le décalage à gauche (opérateur <<) est une façon très efficace de multiplier par 2 (ici, 2^5, soit 32). La "base" ici est 33, avec un offset initial.

Il faut également ramener, moyennant par exemple un modulo, la clé à une position possible dans la table, en fonction de sa taille (dans lisp.h) qu'on changera pour qu'elle soit un nombre premier assez grand (aux environs de 65.000) ; pour l'instant :

```
#define MAXOB 65000
```

Pouvoir agrandir la table au cas où elle est trop remplie doit pouvoir aider. Par ailleurs, nettoyer régulièrement la table (gc_garbage()) permet normalement de ne pas la remplir de symboles ne servant à rien ; c'est ce qui est déjà fait dans eval().

IV - Maintenir le programme

Il faut bien entendu maintenir le programme pour qu'il continue à tourner ; les fonctions put_symb() et find_symb() doivent être réécrites. Par exemple, find_symb() pourrait ressembler à :

```
objet find_symb(String S)
{ flatlist liste = Oblist[hashmodulo(S)] ;                  // trouve l'adresse
while (liste)
{ if (not strcmp(S, String(Pname(liste->car))))     // fouille la liste
return liste->car ;
liste = liste->cdr ; }
return NULL ;
}
}
```

La fonction hashmodulo() est à écrire, et la version de Oblist choisie ici est flatlist *.

V - Exercice

Allez-y (il faut que ça tourne avec hachage) et envoyez votre code avec le sujet « Exo Listes 9 Hach ».

Conclusion

A la fin de ce chapitre, vous maîtrisez normalement les tables d'associations diverses et variées, que ce soit dans l'évaluation des symboles, l'application de fonctions, le hachage, ou les tables de traduction et les callbacks des widgets. Sous ces différentes formes, c'est toujours le même concept qui s'applique, celui que vous aviez déjà rencontré avec les listes d'associations ou de propriétés en Lisp.

Au passage, vous avez pris connaissance de la plus sophistiquée des structures de contrôle standards, le throw-catch, que vous retrouverez dans d'autres langages, ainsi que du principe de la compilation modulaire par Makefile.

Cette compilation modulaire par Make est largement améliorée dans d'autres outils plus récents de la même famille. Je vous recommande en particulier Cmake, qui permet de compiler plusieurs langages, ainsi que Qt, au sein d'un même projet.

Nous allons maintenant abandonner les problèmes qui s'expriment avec des listes, qui n'ont plus de secrets pour vous, et passer à un niveau de complexité supérieur, celui des arbres.