Cours Programmation
MIM 1, 2000–2001, Devoir 3
Un interprète Lisp dynamique en C

Luc Bougé, Alice Bonhomme
Conception initiale : Luc Bougé, Raymond Namyst, 1998

8 décembre 2000

1 Présentation du devoir

Ce devoir est à faire par groupes de 2. Il est à rendre pour le

vendredi 23 décembre 2000

Vous devez rendre pour chaque groupe :

- une description des choix de conception et de leurs conséquence ; (en particulier, commentez le découpage en modules, les fonctions exportées et celles confinées au module, etc.)
- les listings des programmes et des diagnostics produits par lint ;
- un suite de tests de validation de votre produit ;
- une copie d'écran des sorties.

Au cours du TP suivant, chaque groupe devra faire la démonstration de son programme.

L'ensemble est évalué en tout à 6 heures de travail et 2 heures de rédaction. Pour toute difficulté, contactez votre "TDman" par courrier électronique :

Luc.Bouge@ens-lyon.fr, Alice.Bonhomme@ens-lyon.fr

Remarque. *La qualité de la rédaction est très importante. La notation prend en compte le soin, la structure et l'orthographe. En particulier, l'aspect "esthétique" de vos programmes est déterminant : indentation, aération, lisibilité, choix des identificateurs, etc. Vos programmes doivent être aussi élégants qu'une démonstration de maths !*

Étape 1 : évaluateur LISP en C, version de base

Le but du devoir est de réaliser un évaluateur LISP en C en soignant tout particulièrement la technique de conception et de programmation.

Pour vous mettre en route rapidement, on vous fournit les fonctions de manipulation des listes, et les fonctions de lecture et d'écriture.

Le code est disponible sous

http://www.ens-lyon.fr/~bouge/Teaching/Programmation/Lisp_C/.

Le coeur est la boucle *read/eval/print* :

```
for (;;) {
        printf(">\t");
        fflush(stdout);
        print_list(eval_list(read_list()));
        print_newline();
}
```

Voici un exemple du comportement de départ. Pour le moment, la fonction eval_lisp est l'identité. Pour sortir, on tape ^D. À vous de faire mieux !

```
petit-gris% make lisp
gcc -g -Wall -c eval.c
gcc -g -Wall -c list.c
gcc -g -Wall -c main.c
gcc -g -Wall -c print.c
gcc -g -Wall -c read.c
gcc -g -Wall -c token.c
gcc -g -Wall eval.o list.o main.o print.o read.o token.o -o  lisp
petit-gris% lisp
>       1
1
>       (1 2)
(1 2)
>       (a b c)
(a b c)
>       (((nil 1) 2)))))))))       /* Note the extra clo-
sing pars */
((() 1) 2))
>       End of input stream?       /* The user has just ty-
ped ^D */
See you later, alligator!


User signal 1
petit-gris%
```

Concevoir un évaluateur list eval_list(list) du sous-ensemble classique du langage LISP qui propose les mêmes fonctionnalités que les interprètes *lelisp* et LISP en CAML vus en cours. Le squelette est dans le fichier eval.c. Préciser si vous considérez une liaison dynamique ou statique, et montrer la différence sur des exemples.

Remarque. *Votre code ne doit dépendre que de l'interface proposée. Il ne doit manipuler les listes qu'à travers les fonctions prototypées dans le fichier lisp.h, sans jamais faire d'hypothèse sur l'implémentation sous-jacente.*

Votre implémentation doit contenir au moins les fonctions suivantes :

- nil et (), cons, car, cdr, eq (égalité *physique*) ;

– quote, if ;
– opérations arithmétiques et booléennes usuelles ;
– read, print ;
– définitions globales de variables/fonctions par define.

Pour cela, vous serez amenés à ajouter des fonctions de manipulation du genre de celles de list.c, en particulier l'égalité des objets LISP : l'égalité *logique* sur les entier et les noms, mais l'égalité *physique* sur les listes. Ajoutez-les dans des fichier séparés : fonctions.c, et concevez le coeur de votre évaluateur de manière extensible (*extensible*). On doit pouvoir ajouter des fonctions arithmétiques sans modifier cette partie de code !

Commentez soigneusement votre implémentation. L'usage de make vous sera très utile pour maintenir votre projet : étendez le Makefile proposé.

Comme test, faites tourner *factorielle*, *fibonacci*, et un micro-évaluateur LISP simple écrit en LISP, par exemple celui réalisé en TD. Mesurer le nombre d'objets alloués à chaque fois en équipant le code de compteur. Votre avis sur les résultats ?

Étape 2 : gestion des erreurs

Modifier votre implémentation pour rattraper élégamment les anomalies et ainsi éviter de sortir brutalement de l'évaluateur en perdant l'environnement en cas d'erreur de frappe ou d'évaluation.

Erreurs à la lecture : détection de fin de fichier, etc.

Erreurs à l'évaluation : nombre erroné de paramètres, application d'un opérateurs en dehors de son domaine, variable non définie, etc.

Ajouter de plus une fonction de fin de session élégante (bye).

Remarque. *Un conseil : lever un signal et de le rattraper dans un* handler *qui fait un saut non local par la fonction longjmp.*

Bonus : Amélioration de la gestion mémoire

L'objectif est ici de raffiner la gestion mémoire, c'est-à-dire la fonction alloc_new_object(). Rien d'autre ne doit changer dans votre code !

Au lieu d'appeler systématiquement la fonction malloc du système, définissez un (grand) tableau d'objets et utiliser les techniques de gestion d'occupation vues en TD : ajout d'un champ additionnel ou gestion d'une liste libre.

Tester soigneusement l'interaction entre votre gestion mémoire et la récupération d'erreurs. Que faites-vous en cas de saturation mémoire ? L'espace alloué lors d'une évaluation avortée est-il récupéré ?

Remarque. *Il s'agit juste ici de cerner le problème du* garbage collection*. La mise en place d'un* garbage collector *est largement en dehors de la portée du devoir ! Mais que cela ne vous empêche pas d'y réfléchir, ô futurs chercheurs !*