Cours de Programmation MIM1, 2000-2001, Devoir 3
Un compilateur LISP en C

Laure DANTHONY, Clément MENIER

pour le 22 décembre 2000

Table des matières

Commentaires généraux

Dans ce devoir, nous présentons la conception d'un évaluateur LISP en C. Dans un premier temps, nous présentons notre code de l'évaluateur LISP. Dans un deuxième temps, nous présenterons ses capacités à l'aide d'un jeu de nombreux tests. Ensuite nous détaillerons notre gestion des erreurs. Enfin, nous montrerons comment améliorer l'analyse lexicale à l'aide des outils C.

1 La boucle d'évaluation LISP

1.1 La segmentation des modules

Nous avons choisi de partager les différentes fonctions de la façon suivante :

- le fichier *main.c* contient la fonction principale qui initialise les fonctions de base, et évalue une ligne de commandes LISP ;
- les fichiers "boîtes à outils" *list.c* (fichier initial) et *list_add.c* (fonctions ajoutées) offrent des outils de manipulation des listes ;
- les fichiers *read.c*, *print.c*, *token.lex* sont les outils d'analyse lexicale, de saisie clavier et d'impression à l'écran ;
- les fichiers *eval.c*, *keyword_fct.c*, *base_fct.c* réalisent le cœur de l'évaluateur, c'est-à-dire la boucle `eval/apply` et les fonctions de base. Notons que l'on a séparé les fonctions de type "keyword" (qui ne nécessitent pas d'évaluation préliminaire de tous leurs arguments) et les autres ;
- enfin le fichier *errors.c* permet de rattraper "élégamment" les erreurs.

1.2 La boucle eval/apply

Les fonctions `eval` et `apply` du module *eval.c* sont mutuellement récursives. Elles constituent le cœur de l'évaluateur LISP :

- La fonction `eval(list l, environment env)` évalue la liste `l` dans l'environnement local `env`, c'est-à-dire :
  - si la liste `l` est vide ou si c'est un entier, elle est renvoyée telle qu'elle ;
  - si c'est une variable, elle renvoie la valeur de la variable dans l'environnement ;

  Dans les autres cas, la liste `l` est de type `cell`, alors :
  - si le premier élément est un mot clef (`quote`, `if`, `define`, ...), elle n'évalue pas les arguments et appelle la fonction associée via la procédure `apply_keyword_fct` (par exemple, il ne faut surtout pas évaluer toute la queue d'une liste dont le premier élément est `if`) ;
  - sinon, le premier élément est une fonction `list f`, et il s'agit d'évaluer la liste de ses arguments, puis appliquer la fonction à ces arguments , d'où l'appel à la fonction `apply`.

- La fonction `apply(list f, list arg, environment env, int traced)` applique quant à elle la fonction `f` (qui peut être prédéfinie ou non) à la liste de ses arguments :
  – dans le cas où `f` est prédéfinie, on applique cette fonction de base à la liste des arguments dans ce même environnement ;
  – dans le cas où la liste `f` est en fait une définition de fonction, on crée un environnement local correspondant aux paramètres et on évalue le corps de la fonction dans ce nouvel environnement.

  Remarque 1 Le paramètre `traced` indique si la fonction à traiter doit être "tracée".

1.3 Gestion des environnements

Nous avons délibérément choisi de créer une structure `environment` plutôt que de récupérer l'outil liste préexistant: la notion d'environnement est interne au programme C et ne se situe pas au niveau LISP. Un environnement est en fait une liste chaînée de listes ("`value`") qui possèdent un nom ("`name`"), un booléen ("`traced`") et qui pointent sur la liste suivante "`next`". Les fonctions du module *environment.c* permettent de modifier un environnement, de vérifier si une variable est définie dans un environnement donné, et de récupérer la valeur de la variable dans cet environnement. Nous avons décidé de différencier l'environnement global, c'est à dire celui du toplevel, de l'environnement local qui est modifié au cours de l'évaluation. L'environnement global se trouve dans la variable `it`. Cette distinction permet de définir des variables globales au cours d'une évaluation.

Remarque 2 Voici quelques spécifications de notre code :

– la variable t ("true") est protégée et ne peut être redéfinie ;

– il y a une notion de "sur-définition" : une variable a redéfinie est en fait liée à deux endroits dans la liste, mais le parcours est ensuite fait de sorte que ce soit la dernière valeur de a qui soit prise en compte (heureusement pour la gestion des variables locales !) ;

– la distinction environnement global / local nous a contraint de choisir un ordre de priorité dans la recherche d'une variable dans l'environnement: nous avons choisi de privilégier les variables locales aux variables globales. Malheureusement ce système ne nous permet pas de modifier la valeur d'une variable locale par un `define`. Cela n'est pas vraiment gênant si l'on considère le `let` de la même manière que CAML (c'est-à-dire statique);

– la fonction `is_bounded` renvoie FALSE si la variable n'est pas définie, 1 si elle n'est pas tracée et 2 sinon pour éviter de refaire un parcours d'environnement.

1.4 Les fonctions de base

Nous avons implémenté un certain nombre de fonctions de base, telles que les fonctions arithmétiques usuelles ($+, *, \ldots$), les comparaisons ($<$, neq, $<=, \ldots$), les fonctions usuelles sur les suites (longueur, car, cdr) les fonctions booléennes (ou, et ...), ainsi que d'autres fonctions qui sont à découvrir dans la section "tests". Voyons maintenant les caractéristiques de cette implémentation :

– en premier lieu, elle est totalement *extensible*, c'est à dire que si l'on veut ajouter une fonction de base, on ne change rien au code, on ajoute la définition de la fonction à la suite des fonctions déjà existantes ;

- les noms des fonctions de base sont regroupés dans un type `enum` ;
- on dispose d'un tableau des fonctions de base, une fonction de base étant une structure contenant un nom ("`name`") et un pointeur sur une fonction qui prend en paramètre une liste d'arguments ;
- les fonctions de base sont chargées en même temps que le toplevel : la fonction `init_base_fct` initialise le tableau des fonctions de base avec ces fonctions.

Remarque 3 Au sujet des fonctions "de base"

- les fonctions `cadr`, `cdadar`, `cdadadadddddadr`, ... , sont regroupées en une seule fonction. Nous aurions aussi pu gérer les fonctions `car` et `cdr` par cette fonction, mais vu la fréquence d'utilisation de ces 2 fonctions nous avons privilégié la rapidité (la fonction générique étant beaucoup plus lourde) ;
- certaines de ces fonctions ne sont pas implémentées comme dans *lelisp*, mais sont des versions plus intuitives, comme par exemple la comparaison $<$.
- les paires pointées et `set` ne sont pas gérés, mais leur gestion est très aisée (pour la paire pointée la séparation `car` et `cdr` dans le type `object` nous permet de les créer mais la fonction d'affichage serait à modifier légèrement).

2 Tests

2.1 Tests des fonctions de base

Dans cette section nous montrons comment notre évaluateur répond à des commandes de base.

```
>       1
= 1
>       (+ 1 2)
= 3
>       (+ 1 2 3 4)
= 10
>       (- 1 2 3 4)
= -8
>       (1 2 3)
**eval : cannot apply a number
>       ()
= ()
>       nil
= ()
>       (+ 1 (- 2 3))
= 0
>       (define a 1515)
= 1515
>       a
= 1515
```

Quelques exemples de manipulation des listes :

```
>       (cons 1 nil)
= (1)
```

```
>       (cons 1 (cons 2 nil))
= (1 2)
>       (car (cons 1 nil))
= 1
>       (cdr (cons 1 (cons 2 nil)))
= (2)
>       (cdr '(1 2))
= (2)
```

En ce qui concerne les fonctionnalités de programmation du LISP :

```
>       (eval (quote (+ 1 2)))
= 3
(eval '(+ 1 2))
= 3
>       (quote (eval (+ 1 2))))
= (eval (+ 1 2))
>       (print 'toto)
toto
= toto
>       (read)
1
= 1
>       (null nil)
= t
>       (eq 1 (- 3 2))
= t
>       (eq 1 2)
= ()
>       (eq '1 '1)
= t
>       (eq '(1) '(1))
= ()
>       (atom 1)
= t
>       (atom '(1))
= ()
>       (null ())
= t
>       (setq a 12)
= 12
>       (if (< a 10) (print 'petit) (print 'grand))
grand
= grand
>       (progn (print 'a) (print 'b) (print 'c))
a
b
c
= c
>       (cond ((eq a 1) (print 'un)) ((eq a 2) (print 'deux)) (t (print 'grand)))
grand
= grand
>       (setq x 12)
= 12
>       (eval (cons '+ '(1 2)))
= 3
>       (or (eq 1 2) (eq 2 2))
= t
```

```
>       (or (eq 1 2) (eq 2 3))
= ()
>       (and (eq 1 2) (eq 2 2))
= ()
>       (and (eq 1 2) (eq 2 3))
= ()
```

Et maintenant des définitions de fonctions :

```
>       (de f (m n) (+ m n))
= f
>       (f 1 2)
= 3
>       (de f () (print 'hello))
= f
>       (f)
hello
= hello
>       (de fac (n) (if (eq n 0) 1 (* n (fac (- n 1)))))
= fac
>       (fac 4)
= 24
```

Et même des fonctions d'arité variable:

```
>       (define add (lambda l (if (null l) 0 (+ (car l) (apply add (cdr l))))))
= (lambda l (if (null l) 0 (+ (car l) (apply add (cdr l)))))
>       (add 1 2)
= 3
>       (add 1 2 3 4 5 6 7)
= 28
```

Plus fort encore :

```
>       (+ 1.5 3.5)
= 5
>       (sin (/ 3.14 2))
= 1
>       (sin 3.14)
= 0.00159265
>       (asin 1)
= 1.5708
```

Et oui, notre LISP gère les réels (voir la dernière section pour plus d'information) !

2.2 Tests de l'environnement

```
>       (de f (m n) (+ m n))
= f
>       (f (define c 1) 2)
= 3
>       (let ((a 1 ) (b 2)) (+ a b)))
= 3
>       (let ((a 1)) (progn (let ((b 3)) (define c b)) (+ c a)))
4
>       (let ((a 1)) (progn (define a 2) a))
= 1
>       a
= 2
```

Sur ce dernier exemple on remarque la priorité des variables locales (`let`) par rapport aux variables globales (`define`). Mais la variable globale reste bien effective pour le toplevel.

2.3 Quelques fonctions plus élaborées

Tout d'abord, la fameuse fonction des tours de hanoï :

```
>(de hanoi (n a b c)
    (if (eq n 1)
        (
         progn
          (print '"deplacement de ")
          (print a)
          (print '" vers ")
          (print c))
      (progn
        (hanoi (- n 1) a c b)
        (print '"déplacement de ")
        (print a)
        (print '" vers ")
        (print c)
        (hanoi (- n 1) b a c))))
= hanoi
>      (hanoi 2 1 2 3)
deplacement de
1
 vers
2
déplacement de
1
 vers
3
deplacement de
2
 vers
3
= 3
```

Ensuite, les nombres de Fibonnaci, version récursive et version plus calculatoire :

```
>       (de pow (m n) (if (<= n 0) 1 (* m (pow m (- n 1)))))
= pow
>       (pow 2 3)
= 8
>       (de fib (n) (if (eq n 0) 1 (if (eq n 1) 1 (+ (fib (- n 1)) (fib (- n
2))))))
= fib
>       (fib 6)
= 13
>       (define a (/ (+ 5 (sqrt 5)) 10))
        (define b (/ (- 5 (sqrt 5)) 10))
        (define c (/ (+ 1 (sqrt 5)) 2))
        (define d (/ (- 1 (sqrt 5)) 2))
= -0.618034
= 0.276393
= 1.61803
= -0.618034
```

```
>       (de fib2 (n) (+ (* a (pow c n)) (* b (pow d n))))
= fib2
>       (fib2 6)
= 13
```

Enfin un peu de calcul numérique :

```
>       (de tvi (f a b eps)
    (if (< (- b a) (* 2 eps))
        (/ (+ a b) 2)
      (if (<= (* (f a) (f (/ (+ a b) 2))) 0)
          (tvi f a (/ (+ a b) 2) eps)
          (tvi f (/ (+ a b) 2) b eps))))
= tvi
>       (de f (x) (sin x))
= f
>       (tvi f 1 4 0.0001)
= 3.14151
>       (de g(x) (- x (cos x)))
= g
>       (tvi g 0 4 0.0001)
= 0.739075
```

2.4 Un évaluateur LISP en LISP

```
>       (de toplevel2 ()
    (progn
      (print '"new toplevel")
      (progn (define running t))
        (while running
          (progn
            (print '"MD-LISP?")
            (let ((it (read)))
              (if (equal it '(end))
                  (setq running nil)
                (progn
                  (let ((result (eval it)))
                    (progn
                      (print '"result = ")
                      (print result)
                      ))))))))))
= toplevel2
>       (toplevel2)
new toplevel
MD-LISP?
(+ 1 2)
result =
3
MD-LISP?
(end)
= ()
```

Magnifique! Un méta-évaluateur LISP exécuté au-dessus d'un interpréteur LISP en C!

3 Gestions des erreurs

Malheureusement l'utilisateur est un être humain qui fait parfois des erreurs et il serait dommage pour une simple faute de frappe de perdre le travail d'une longue nuit. Ainsi il paraît souhaitable de pouvoir rattraper les erreurs de l'interpréteur en stoppant l'évaluation en cours et en retournant au toplevel pour permettre au programmeur de retaper sa commande sans perdre l'environnement.

3.1 La signalisation d'une erreur

Lorsqu'une fonction détecte une erreur, elle affiche un message d'erreur explicitant la raison de l'arrêt de l'évaluation en cours et provoque la levée d'un signal MYSIGERROR. Pour éviter la redondance de code, nous avons créé 3 fonctions de signalement d'erreurs:

- `argerror(char *message)`: qui affiche le message d'erreur et lève le signal MYSIGERROR ;
- `argnberror(char *name, int n)`: pour les erreurs dues à un mauvais nombre d'arguments où `name` est le nom de la fonction incriminée et `n` est le nombre d'arguments désiré ;
- `argtyperror(char *message, list l)`: pour les erreurs dues à un mauvais type de `l`.

3.2 Le rattrapage des erreurs

Le rattrapage d'une erreur se fait par le biais de la gestion des signaux par le système Unix. En effet il est possible de demander au système l'exécution d'une fonction particulière lors de la levée d'un signal. Ainsi nous avons écrit 2 fonctions pour la gestion des erreurs:

- `sigerrorhandler`: gestion des erreurs d'évaluation ;
- `sigeofhandler`: gestion de la fin de l'interpréteur LISP.

Le retour au toplevel lors de l'appel à `sigerrorhandler` est effectué par le jeu de commandes `setjmp` et `longjmp`. Un appel à `setjmp` est fait avant chaque lecture de commande ce qui permet lors de l'appel `longjmp` de revenir au toplevel dans l'état précédent la saisie de commande défectueuse. Ainsi l'interpréteur sera remis dans l'état d'attente de saisie de commandes.

Quelques problèmes sont cependant à éviter. Il faut penser à l'intérieur de la fonction `sigerrorhandler` à débloquer le signal MYSIGERROR sinon il restera bloqué dans la suite du programme et les autres erreurs ne seront alors plus traitées. De plus il y a un risque que la fonction `sigerrorhandler` ne soit pas appelée dès la levée du signal MYSIGERROR ce qui serait gênant car l'évaluation continuerait alors malgré l'erreur. En pratique ce risque n'existe pas car il est spécifié dans la plupart des Unix que `raise` ne retourne qu'après l'exécution du handler.

3.3 Quelques exemples de gestion des erreurs

```
>       (1)
** eval : cannot apply a number
>       (+ a b)
** eval : undefined variable : a
>       (define 1 2)
** define : not a variable name : 1
>       (define t 2)
** define : not a variable name : t
```

```
>       (f 1 2)
** eval : not a function : f
>       (car 1)
Cannot take the car of a number or a name!
>       (define a 1 2)
** define : wrong number of arguments : 2
>       (end)
That's all folks ...
```

Les erreurs sont accompagnées d'un message d'erreur et d'un retour au toplevel. Malheureusement quelques programmes sont vraiment très durs à gérer.

```
>       (de f () (f))
= f
>       (f)
Segmentation fault
```

Cette dernière erreur doit être due à une erreur d'overflow (nos tests montrent que la fonction `apply` est appelée environ 35000 fois avant de produire cette erreur).

4 Améliorations : un nouveau parseur, une gestion des réels, ...

4.1 La gestion de l'environnement

La méthode de séparation des deux types d'environnement est assez efficace comme le montrent les quelques tests si toutefois on garde en mémoire cette spécificité. Pour échapper à ce problème une bonne méthode serait de gérer l'environnement de la manière suivante :

- l'environnement est représenté par une liste d'éléments ;
- un élément comporte un nom de variable et une liste de définitions, c'est-à-dire de valeurs, comportant chacune un champ pour savoir si elles sont globales ou locales ;
- lors d'une déclaration locale, il suffit de rechercher l'occurence de cette variable dans l'environnement :
  - si elle avait déjà été définie, il suffit d'ajouter à la liste de ses définitions la nouvelle valeur en tête de liste avec la mention "locale" ;
  - sinon on crée une nouvelle entrée dans la liste d'éléments pour cette variable et on ajoute la définition locale ;
- lors d'une déclaration globale il faut remettre toute la liste des définitions à zéro et ne laisser que la nouvelle définition globale ;
- après chaque évaluation avec création de variables locales, il suffit de supprimer la tête de liste des définitions si celle-ci est de type locale.

Cette méthode nécessiterait une exploration plus approfondie suivant les différents cas possibles mais elle semble offrir des possibilités assez grandes.

4.2 Le problème de l'utilisation mémoire

Dans notre LISP il n'y a malheureusement aucune libération d'espace mémoire, ce qui est assez gênant. Pour éviter cela il suffirait de libérer les variables locales en supprimant après l'évaluation celles créées (les `n` dernières si l'évaluation en avait créé `n`). Une autre amélioration serait de libérer les listes crées lors de l'analyse. Pour cela il suffit d'appeler `free_list(l)` à la fin d'`eval_list`. Pour que cela fonctionne correctement il faudrait alors copier les listes lors des définitions et non leur adresse car sinon leur valeur disparaîtrait.

4.3 L'utilisation de lex

Nous avons réécrit l'analyseur syntaxique en utilisant l'outil *lex* qui permet de rechercher un motif dans un texte. Le nouveau parseur se trouve dans le fichier *token.lex*. La fonction `look_token` appelle `yy_lex` si il n'y a pas déjà eu un token lu mais non enregistré (variable `is_empty_token`). Le résultat du parseur est stocké dans le token `x`. Les règles de parsing sont assez simples :

- `[\n\t ]` : pour la gestion des espaces et des fins de ligne ;
- `[(]` et `[)]` : pour les parenthèses ;
- `[']` : pour le quote ;
- `[0-9]+` : pour les entiers ;
- `[^\"()\'\n\t ]+` : pour une chaîne de caractères normale ;
- `\".+\"` : pour une chaîne de caractères entre guillemets.

La compilation nécessite l'ajout d'une instruction *lex -t token.lex >token.c*. Malheureusement l'outil *lex* génère un code qui provoque quelques warnings lors de la compilation avec -Wall mais ceux-ci sont sans gravité. Un autre problème est la mauvaise gestion de la fermeture de l'entrée standard par *lex*. Il semblerait qu'il faille redéfinir une fonction `yy_wrap` avec un test de fermeture de l'entrée standard avec dans le cas affirmatif la levée du signal MYSIGEOF.

4.4 La gestion des réels

En fait la réécriture de l'analyseur syntaxique fut surtout motivée par notre volonté de gérer des réels à la place des entiers. Pour ce faire nous avons modifié le type `TOKEN_INT` en `TOKEN_NUM` et modifié les structures en conséquence. Un nombre est alors de type `double`. L'affichage des réels se fait par l'option `%g` de printf qui permet un affichage plus esthétique.

La définition des réels utilisée dans le format *lex* est

```
[-+]?([0-9]+|([0-9]*\.[0-9]+))([eE][-+]?[0-9]+)?
```

et nous avons ajouté une fonction `string_to_num` qui convertit une chaîne de ce format en un réel. Cela permet de gérer des calculs mathématiques assez sympathiques comme les fonctions trigonométriques usuelles.

Conclusion

Nous disposons à présent d'un interpréteur LISP d'une fonctionnalité satisfaisante pour de petites applications. Pour une utilisation avancée, une première amélioration notoire serait la gestion d'autres fonctions de base, comme `set`, `append`, `map`, ... D'autre part les améliorations citées telles que la gestion mémoire et l'implémentation d'un système d'environnement plus performant deviennent vite nécessaires lorsque l'on considère des programmes un peu plus gros. Enfin une dernière amélioration serait de permettre à l'utilisateur de pouvoir manipuler les fonctions de base comme des objets à part entière, et aussi d'avoir une plus grande compatibilité avec la notion de langage d'ordre supérieur.