Valisp séance 4 : Interpréteur et Parseurs

L2 Informatique – Université Côte d'Azur

Partie 1 — Interpréteur

Exercice 1 — Une première version de l'interpréteur

1. On travaille dans cette partie dans le fichier `interpreteur.c`. Les fonctions ci-dessous sont toutes mutuellement récursives. Pour éviter des problèmes d'œufs et de poules (je ne peux pas déclarer `eval` car `apply` n'est pas définie, mais je ne peux pas non plus déclarer non plus `apply` car `eval` n'est pas définie), commencez par déclarer `eval` et `apply` dans le fichier *header* (= `.h`) associé et incluez ce fichier en-tête (*header*) dans votre fichier source (= `.c`).
2. Écrivez alors les quatre fonctions ci-dessous. Dans un premier temps, ne gérez pas les `lambda` ni les `macro` mais uniquement les primitives et les formes spéciales.
   (a) `sexpr eval_list(sexpr liste, sexpr env)`
   (b) `sexpr bind(sexpr variables, sexpr liste, sexpr env)`
   (c) `sexpr eval(sexpr val, sexpr env)`
   (d) `sexpr apply(sexpr fonction, sexpr liste, sexpr env)`
3. Testez bien votre code dans l'interpréteur valisp avec la primitive `+` et les formes spéciales `defvar` et `setq`.

Exercice 2 — Primitives supplémentaires

1. Formes spéciales
   (a) `sexpr quote_valisp(sexpr liste,  sexpr env)`
   (b) `sexpr lambda_valisp(sexpr liste,  sexpr env)`
   (c) `sexpr macro_valisp(sexpr liste,  sexpr env)`
   (d) `sexpr if_valisp(sexpr liste,  sexpr env)`
2. Primitives
   (a) `sexpr eval_valisp(sexpr liste, sexpr env)`
   (b) `sexpr apply_valisp(sexpr liste, sexpr env)`

Exercice 3 — Configurer emacs pour valisp

1. Configurer GNU/Emacs. Si vous êtes là, merci d'appeler M. Baldellon
   — Si vous êtes sur les machines de la fac, il suffit de taper dans un terminal `/u/profs/upinfo/config/configurer-emacs.sh`.
   — Si vous êtes sur votre machine personnelle, allez sur la page https://upinfo.univ-cotedazur.fr/~obaldellon/tp
2. Ouvrez le fichier demo.val et testez votre code dans GNU/Emacs

Exercice 4 — Finir l'interpréteur

1. Ajoutez maintenant le support des `lambda`.
2. Testez avec le fichier `demo-fonction.val`
3. Si tout marche bien, ajoutez maintenant le support des `macro`.

4. Testez avec le fichier `lib.val`.

Partie ii — Le parseur

Pour prétendre avoir écrit un langage de A à Z, il vous manque le parseur : c'est-à-dire la fonction qui transforme une chaîne en `sexpr`.

Exercice 5 — Fonctions de prédicats

1. `int est_espace(char c)` : il y a quatre espaces en vaλisp, l'espace traditionnel, la tabulation le retour à la ligne et le changement de page (`'\f'`).
2. `int est_chiffre(char c)` Ce sont les symboles entre `'0'` et `'9'`.
3. `int est_symbole(char c)` un symbole est n'importe quel caractère sauf : les espaces (cf question 1), les parenthèses, le symbole quote « ' », le symbole commentaire (point-virgule) et évidemment le caractère nul.
4. `int fin_mot(char c)` si c correspond à un symbole valide à la fin d'un mot (entier ou chaîne). La fonction teste si `c` est un espace ou si `c` est une parenthèse fermante `')'` ou un début de commentaire `';'`.

Exercice 6 — Fonctions auxilliaire

La partie délicate du parseur sera d'extraire des sous-chaînes d'un texte et de les stocker dans notre mémoire.

1. Pour lire une chaîne, il va falloir être capable de lire les caractères échappés. Par exemple la chaîne `"123\"\n456"` ne contient que 8 caractères. Écrire une fonction `longueur_chaine(char * texte, int deb, int fin)` qui calcule la longueur de la sous-chaîne de `texte` entre les indices `deb` (inclus) et `fin` (exclus) et en prenant en compte les caractères échappés (le nombre sera forcèment inférieur ou égal à la quantité `fin-deb` ; l'égalité aura lieu dans le cas ou il n'y a aucuns caractères échappés.).
2. Écrivez une fonction `char * sous_chaine(char * texte, int deb, int fin)` qui copie la sous-chaîne comprise entre les indices `deb` (inclus) et `fin` (exclus) en remplaçant les caractères échappés par leur véritables valeurs.
3. Écrivez une fonction `int nettoyer_espaces(char * texte, int i)` qui parcours la chaîne à partir de `i` en ignorant les espaces et les commentaires jusqu'à tomber sur le début du `mot` suivant et renvoie alors l'indice du premier caractère de ce mot. Remarque : un commentaire commence par un point-virgule et se termine par une nouvelle ligne.

À partir de là, nous allons pouvoir commencer le parseur à proprement parler. Chaque fonction cherchera à lire à partir d'un indice `i` un objet particulier. L'objet lu sera stocké dans un pointeur de `sexpr` et la valeur de retour indiquera le succés de l'opération. On utilisera les codes d'erreurs suivants :

- -1 si vide
- -2 si la sexpr est incomplète
- -3 si erreur durant la lecture d'un entier
- -4 si on rencontre une parenthèse fermante à un endroit impossible.
- indice suivant sinon

Exercice 7 — Parser les atomes

1. `int parse_chaine(char * texte, int i, sexpr * res)`
2. `int parse_entier(char * texte, int i, sexpr * res)`
3. `int parse_symbole(char * texte, int i, sexpr * res)`

Exercice 8 — Le parseur

En appliquant l'algo du cours, implémentez les deux fonctions suivantes.

1. `int parse_liste(char* texte, int i, sexpr * res)`
2. `int parseur(char* texte, int i, sexpr* res)`