Valisp séance 3 : Environnement et ramasse-miettes

L2 Informatique – Université Côte d'Azur

Partie I — Environnement

On rappelle que l'environnement est une liste de liaisons (*bindings*), c'est-à-dire une liste de couple où chaque `car` correspond à un symbole et chaque `cdr` à une valeur. Par exemple, l'environnement dans lequel `x` vaut 3, `y` vaut 2 et `z` vaut 57 sera codé par la liste chaînée :

`(x . 3)` → `(y . 2)` → `(z . 57)` → `nil`

Exercice 1 — Initialiser l'environnement (fichier `environnement.c`)

1. Créez une variable globale `ENV` initialisée à `NULL`. Ce sera notre environnement global.
2. `.h` Pour permettre l'accès à cet environnement sans avoir à le rendre public, écrivez une fonction `sexpr environnement_global(void)` qui renvoie cette variable globale.
3. `.h` Écrivez enfin une fonction `void initialiser_memoire(void)` qui appelle la fonction du premier TP `initialiser_memoire_dynamique`, crée un nouveau symbole `t` (correspondant au booléan `true`) et affecte à `ENV` une liste chaînée d'un seul élément : le couple `(t . t)`.

Exercice 2 — Manipuler l'environnement

1. `.h` Écrivez une fonction `int trouver_variable(sexpr env, sexpr variable, sexpr *resultat)` qui parcourt l'environnement `env` et cherche si la variable (=le symbole) donnée en paramètre s'y trouve. S'il la trouve, on met la valeur associée à la variable dans `resultat` et on renvoie `0`; si la variable n'est pas présente dans l'environnement on renvoie `-1`.
2. `.h` Écrivez une fonction `int modifier_variable(sexpr env, sexpr variable, sexpr valeur)` qui parcourt l'environnement donné en paramètre, cherche la variable donnée en paramètre et modifie la liaison correspondante. La fonction renverra `0` en cas de succès et `-1` si la variable n'a pas été trouvée.
3. `.h` Écrivez enfin une fonction `void definir_variable_globale(sexpr variable, sexpr valeur)` qui travaille avec l'environnment global (`ENV`) et modifie comme dans la question précédente la valeur associée à la variable. Si, par contre, la variable n'est pas encore dans l'environnement, il faudra ajouter la nouvelle liaison **à la fin** de la liste chaînée.
4. `.h` En utilisant la fonction précédente écrire en deux lignes chacune des deux fonctions ci-dessous :
   (a) `void charger_une_primitive(char* nom, sexpr (*prim)(sexpr, sexpr))`
   (b) `void charger_une_speciale(char* nom, sexpr (*prim)(sexpr, sexpr))`

Partie II — Ramasse-miettes

Exercice 3 — Dans le fichier `allocateur.c`

1. `.h` Écrivez une fonction `int pointeur_vers_indice(void *ptr)` qui renvoie l'indice du bloc correspondant au pointeur passé en argument (attention : le pointeur pointe vers la première case de la zone. Le bloc est la case précédante). Ajoutez une erreur fatale si l'indice n'est pas un indice valide du tableau.
2. `.h` Écrivez une fonction `int ramasse_miettes_lire_marque(void *ptr)` qui renvoie le bit associé au ramasse-miettes. Attention, la fonction prend en paramètre un pointeur.

3. `.h` Écrivez une fonction `int ramasse_miettes_poser_marque(void *ptr)` qui met à 1 le bit associé au ramasse-miettes. On échouera avec une erreur fatale (*cf.* TP 2) si le bloc est déjà marqué.
4. Écrivez une fonction `int bloc_libre(i)` qui renvoie un booléen indiquant si le bloc d'indice `i` doit être supprimé. Attention, le dernier bloc (celui qui est son propre successeur) ne doit jamais être supprimé.
5. `.h` Écrivez enfin la fonction `void ramasse_miettes_liberer()` qui parcourt les blocs mémoires et libère tout les blocs supprimables. On fera attention de fusionner les blocs libres adjacents (cf. CM).

Exercice 4 — Dans le fichier `memoire.c`

1. Écrivez une fonction `void ramasse_miette_parcourir_et_marquer(sexpr s)` qui parcourt récursivement l'environnement en marquant toutes les sexpr rencontrées. On fera attention de ne pas re-parcourir des élements déjà marqués afin d'éviter des boucles infinies (techniquement, l'environnement peut contenir des cycles).
2. `.h` Écrivez alors en deux lignes la fonction `void valisp_ramasse_miettes(sexpr env)` qui appelle la fonction précédente puis la fonction `ramasse_miettes_liberer` de l'allocateur.

Partie iii — Le repl valisp

Exercice 5 — Affichage des informations sur l'environnement

1. Écrivez la fonction `int longueur_env(sexpr env)` qui renvoie le nombre de liaison dans l'environnement.
2. Écrivez la fonction `void valisp_stat_memoire(void)` qui appelle la fonction `afficher_stat_memoire()` de `allocateur.c` puis affiche le nombre de variable. Le résultat devra être de la forme :
   `541/32768 (1.65 %) 89/90 blocs alloués → 18 variables`
3. Écrivez une fonction `void afficher_env(sexpr env)` qui affiche toutes les liaisons de l'environnement (une par ligne), avec en bleu les variables et en blanc les valeurs associées

Exercice 6 — Les primitives

1. `.h` Vous devez avoir déjà écrit la primitive `add_valisp`
2. `.h` Récupérez dans le cours la forme spéciale `defvar_valisp`
3. `.h` Inspirez-vous de cette dernière pour écrire `setq_valisp`

Exercice 7 — Dans le fichier `erreur.c`

1. Definissez une variable global `buf` de type `jmp_buf`
2. `.h` Écrivez une fonction `jmp_buf *jump_buffer()` qui renvoie un pointeur vers cette variable globale.
3. Modifiez alors la fonction `erreur` pour qu'elle fasse un *long jump* vers `buf` plutôt qu'un `exit(1)`. On enleverra aussi l'affichage de l'erreur qui sera fait dans le REPL.

Exercice 8 — On branche le tout

1. Faîtes une sauvegarde de votre travail.
2. Copiez les fichiers suivants dans votre répertoire de travail : `Makefile`, `main.c`, `valisp.c`, `valisp.h`, `parseur.o`, `parseur.h`, `interpreteur.o` et `interpreteur.h`
3. Compilez le tout et lancez le programme `./valisp`
4. Faîtes quelques calculs en affichant régulièrement la mémoire avec `@mem`
5. Lancez le ramasse-miettes avec `@rm`. Vérifiez que la mémoire a bien été nettoyée.
6. Modifiez le fichier `main.c` pour automatiquement appeler le ramasse-miettes entre chaque calculs.

Félicitations !

Vous avez atteint le premier objectif… maintenant, écrivez le parseur et l'interpréteur pour terminer le langage.