Valisp séance 2 : Types et encodages

L2 Informatique – Université Côte d'Azur

Partie I — Préambule

Exercice 1 — Finitions de l'allocateur

1. Téléchargez l'archive `valisp-2.zip`. Dans le répertoire ainsi créé, veuillez copier les fichiers `allocateur.h` et `allocateur.c` de la séance précédente.
2. Compilez le tout. Vous devez absolument avoir fini l'exercice 4 (`allocateur_malloc`) pour commencer ce TP. Pour tester les fonctions de la séance précédente, décommentez le code correspondant dans le fichier `mes_tests.h` et dans la fonction `main` et corrigez éventuellement votre code de la semaine dernière.

Exercice 2 — Gestion des erreurs

1. Dans le fichier `erreur.c` modifiez la fonction `erreur_fatale` pour qu'elle termine le programme avec le code d'erreur 1.
2. En utilisant le pré-processeur, dans le fichier `erreur.h`, créez une macro `#define ERREUR_FATALE(CAUSE)` qui appelle la fonction précédente en renseignant automatiquement le nom du fichier et la ligne à laquelle la macro a été appelée.

Exercice 3 — Gestion de la mémoire

Par la suite, nous n'utiliserons jamais directement `allocateur_malloc` qui est une fonction de bas niveau. Créez un fichier `memoire.c` contenant une fonction `valisp_malloc`. Cette fonction fera appel à `allocateur_malloc` et renverra le pointeur trouvé par cette dernière. Par contre on mettra fin au programme avec une erreur fatale si le pointeur renvoyé est `NULL`.

Lors des autres séances, on modifiera cette fonction pour avoir un comportement plus subtil.

Partie II — Encodage des données

Exercice 4 — Création des types

1. Dans le fichier `types.h`, commencez par définir un alias du type `int` sous le nom de `bool`.
2. Déclarez ensuite dans ce même fichier le type `struct valisp_object`. Attention, nous parlons ici de déclarer et non de définir. Ensuite créez un alias `sexpr` du type `struct valisp_object *`. Ce type est celui de tous les objets valisp que nous créerons à partir de maintenant.
3. Dans le fichier `types.c`, créer un type `valisp_type` construit à partir d'un `enum` correspondant aux six différentes valeurs : `entier`, `chaine`, `symbole`, `couple`, `prim` et `spec`.
4. Définissez dans ce même fichier le type `valisp_cons` défini à partir d'une `struct` contenant deux champs, `car` et `cdr` de type `sexpr`.
5. Créez un type `valisp_data` de type `union` correspondant aux différents types :
   (a) `int`
   (b) `char *`
   (c) `valisp_cons`
   (d) Des fonctions prenant en paramètre deux `sexpr` et renvoyant une `sexpr`.
6. Finalement, créez le type `struct valisp_object`, contenant un champ `type` de type `valisp_type` et un second champ `data` de type `valisp_data`.

Pour chaque type il faudra créer : un constructeur, une fonction pour tester le type (le suffixe `_p` signifiant prédicat) et un accesseur. Il faudra aussi créer une procédure `afficher` que l'on modifiera à chaque nouveau type qui affiche la donnée selon son type ; cette fonction sera commune à tous les types.

Chaque fonction écrite devra être ajoutée au fichier `types.h`.

Exercice 5 — Entiers

1. `sexpr new_integer(int i)`
2. `bool integer_p(sexpr val)`
3. `int get_integer(sexpr val)`
4. `void afficher(sexpr val)`
5. Testez vos fonctions en décommentant les tests de `mes_tests.c` correspondant aux entiers.

Exercice 6 — Chaînes et symboles

1. Afin de pouvoir supprimer les chaînes de caractères avec notre ramasse-miettes il faudra faire attention de placer toutes les nouvelles chaînes dans notre mémoire dynamique (à une adresse renvoyée par `valisp_malloc`). Écrivez ainsi la fonction de signature `char *chaine_vers_memoire(const char *c)`.
2. Créez maintenant les fonctions classiques :
   (a) `sexpr new_string(char *c)` et `sexpr new_symbol(char *c)`
   (b) `bool string_p(sexpr val)` et `bool symbol_p(sexpr val)`
   (c) `char *get_string(sexpr val)` et `char *get_symbol(sexpr val)`
   (d) `void afficher(sexpr val)` (à modifier)
3. On aura souvent besoin de tester si un symbole correspond à une chaîne de caractères donnée. Écrivez la fonction `bool symbol_match_p(sexpr val, const char *chaine)`. On suppose que `val` est une `sexpr` correspondant à un symbole que l'on souhaite comparer avec la chaîne donnée en paramètre. Indice : n'hésitez pas à demander au ChatGPT des hackers en faisant dans le terminal « `man strcmp` ».
4. Testez vos fonctions en décommentant les tests de `mes_tests.c` correspondant aux chaînes et aux symboles.

Exercice 7 — Listes

1. `sexpr cons(sexpr e1, sexpr e1)`
2. `bool cons_p (sexpr e)`
3. `bool list_p(sexpr e)` : si `e` correspond à `nil` ou à un `cons` dont le `cdr` est lui-même un `cons` ou `nil`.
4. `sexpr car(sexpr e)` et `sexpr cdr(sexpr e)`. On générera une erreur fatale si `e` est `NULL`.
5. `void set_car(sexpr e, sexpr nouvelle)` et `void set_cdr(sexpr e, sexpr nouvelle)`
6. `void afficher_liste(sexpr e)` Remarque : se reporter au cours
7. `void afficher(sexpr e)`
8. Testez vos fonctions en décommentant les tests de `test_types.c` correspondant aux listes.

Exercice 8 — Primitives

1. `sexpr new_primitive(sexpr (*p)(sexpr, sexpr))` et `sexpr new_speciale(sexpr (*p)(sexpr, sexpr))`
2. `bool prim_p (sexpr val)` et `bool spec_p (sexpr val)`
3. `sexpr run_prim(sexpr p, sexpr liste, sexpr env)` Qui exécute la primitive `p` avec comme paramètres `liste` et `env`.
4. `void afficher(sexpr V)`

Exercice 9 — L'égalité

1. `bool sexpr_equal(sexpr e1, sexpr e2)` (à faire après l'exercice 10)

Partie iii — Les primitives

Exercice 10 — Un nouveau type d'erreur

Dans cet exercice, nous travaillerons dans le fichier `erreur.c` et chaque primitive écrite devra voir sa signature ajoutée au fichier `erreur.h`.

1. En vous basant sur le cours, créez un type `enum erreurs` contenant les huit erreurs que l'on pourra rencontrer.
2. Chaque erreur contiendra quatre informations : La `sexpr` fautive, le nom de la primitives fautive, un message explicatif et un type d'erreur. Déclarez les quatre variables globales correspondantes :
   (a) `sexpr SEXPR_ERREUR;`
   (b) `char *FONCTION_ERREUR;`
   (c) `char *MESSAGE_ERREUR;`
   (d) `enum erreurs TYPE_ERREUR;`
3. Écrivez maintenant la fonction `void afficher_erreur(void)` qui affiche en rouge un message d'erreur avec toutes les informations des quatre variables globales précédentes. On s'inspirera du cours pour l'affichage.
4. Écrivez alors la fonction `void erreur(enum erreurs type, char *fonction, char *explication, sexpr s)` qui modifie les quatre variables globales et appelle la fonction `afficher_erreur` avant de quitter avec un code d'erreur 1. (Cela peut semblez fastidieux de passer par des variables globales mais cela sera nécessaire lorsqu'on passera au GOTO longue distance pour gérer les erreurs).

Exercice 11 — Écriture des premières primitives

Dans cet exercice, nous travaillerons dans le fichier `primitives.c` et chaque primitive écrite devra voir sa signature ajoutée au fichier `primitives.h`. Dans un premier temps, n'écrivez que la primitives `add_valisp` puis passez au TP3. Vous pourrez terminer l'exercice chez vous.

1. `void test_nb_parametres(sexpr liste, char* fonction, int taille)`
2. `sexpr car_valisp(sexpr liste, sexpr env)`
3. `sexpr cdr_valisp(sexpr liste, sexpr env)`
4. `sexpr cons_valisp(sexpr liste, sexpr env)`
5. `sexpr add_valisp(sexpr liste, sexpr env)`
6. `sexpr less_than_valisp(sexpr liste, sexpr env)`
7. `sexpr sub_valisp(sexpr liste, sexpr env)`
8. `sexpr produit_valisp(sexpr liste,  sexpr env)`
9. `sexpr div_valisp(sexpr liste,  sexpr env)`
10. `sexpr mod_valisp(sexpr liste,  sexpr env)`
11. `sexpr equal_valisp(sexpr liste,  sexpr env)`
12. `sexpr print_valisp(sexpr liste,  sexpr env)`
13. `sexpr type_of_valisp(sexpr liste, sexpr env)`