Programmation impérative en C

Valisp 2. Langage et encodage

Olivier Baldellon

Courriel : prénom.nom@univ-cotedazur.fr

Page professionnelle : https://upinfo.univ-cotedazur.fr/~obaldellon/

Licence 2 — Faculté des sciences et ingénierie de Nice — Université Côte d'Azur

Sommaire

Deux exemples de code

Valisp

```
(defun fact (n)                  ;; def fact(n)
    (if (= n 0)                  ;;     if n == 0:
        1                        ;;         1
      (* n (fact (- n 1)))))     ;;     else: n * fact(n-1)


(dolist (x (liste 11 22 33))     ;; for x in [11, 22, 33]
    (println "x=" x))            ;;     print("x=", x))
```

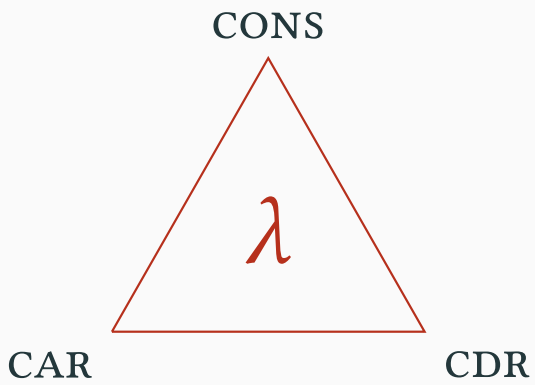
- Il n'y a qu'une syntaxe : `(procédure x1 x2 … xn)`
  - `(defun fonction (x y z) ligne1 ligne2 ligne3)`
  - `(+ 100 20 3)`
  - `(if test then else)`
  - On parle de S-expression
- Remarques :
  - Les retours à la ligne et les indentations sont cosmétiques
  - Très élégant conceptuellement (mais peu plébiscité)
  - Permet l'écrire rapide d'un parseur.
  - Permet l'homoïconicité (≈ le langage peut se programmer lui même)

Valisp

```
(defun fact (n)                  ;; def fact(n)
    (if (= n 0)                  ;;     if n == 0:
        1                        ;;         1
      (* n (fact (- n 1)))))     ;;     else: n * fact(n-1)


(dolist (x (liste 11 22 33))     ;; for x in [11, 22, 33]
    (println "x=" x))            ;;     print("x=", x))
```

- Les différents types rencontrés en Vaλisp :
  - Les entiers. Quoi d'autre ?
  - Les chaînes
  - Les symboles : fonctions, opérations, mots-clé, etc.
  - Les listes
  - Les primitives (non représentées syntaxiquement mais associée à des symboles)



Vaλisp

```
(defvar C (cons 11 22))
          ;; => (11 . 22)

(car C)
          ;; => 11

(cdr C)
          ;; => 12
```

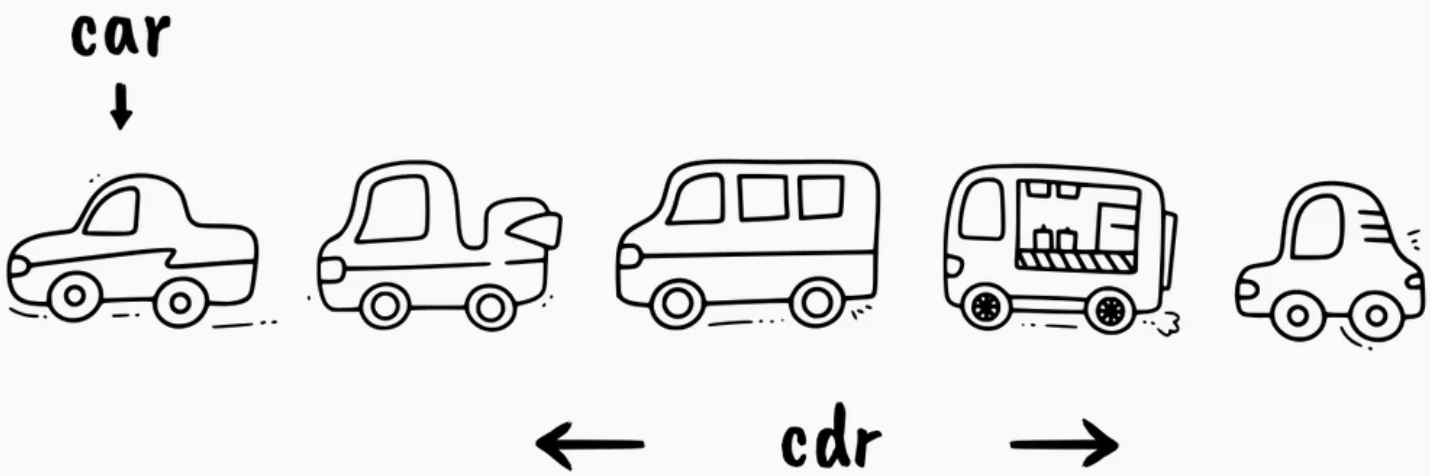
- C'est un fleuve d'afrique, mais pas que.

- C'est une valeur spéciale codée par un simple `NULL`

- C'est l'équivalent sémantique de :
  - False : la valeur fausse (toutes les autres valeurs sont vraies)
  - None : pour coder les valeurs par défaut.
  - `[]` : la liste vide

- Une liste en LISP est définie récursivement par :
  - `nil` est la liste vide
  - `(cons x liste)` est une liste de premier élément `x`.
    - avec `x` une `sexpr` quelconque
    - et `liste` une autre liste
- Mais c'est la même chose qu'en TP avec les listes chaînées ?
- Oui mais...
  - En TP, on parlait de `ajout_lc` de `head` et de `tail`
  - Maintenant on parle de `cons` de `car` et de `cdr` ; c'est beaucoup plus classe.
- Oui mais...
  - En C le premier élément était un entier et le second un pointeur vers une nouvelle liste chaînée.
  - En Vaλisp ce sont deux pointeurs vers des `sexpr` : leur rôle est symétrique
  - Remarque : pas de couple en C.

- Prononciation : `cons` : KONSSE, `car` : KAR et `cdr` : KOUDEUR



- En anglais ***car*** signifie *« voiture »* et ***cdr*** *« le reste des voitures »* (à vérifier)
- Autre explication :
  - CAR : *Contents of the Address Register*
  - CDR : *Contents of the Decrement Register*
- Ce sont des choix historiques gardés par tradition.

Sommaire

- En LISP tout est sexy et tout est expression : on parle de *sexpr*
  - sexpr est (aussi) le diminutif de S-expressions `(+ 1 2)`
  - par opposition au M-expressions `1+2` (désuètes et inutilisées)
  - S : symboliques et M : mathématiques
- Il n'y aura que 6 types dans notre implémentation :
  - les entiers
  - les chaînes
  - les symboles
  - les cons (rappel : à prononcer KONSSE)
  - les primitives
  - les formes spéciales (c'est comme des primitives mais c'est pas pareil)

- Chaque objet LISP est associé un type et une donnée.
- Le type de base pour stocker les objets sera `struct valisp_object`
  - cette structure aura deux champs
  - un champ `type` contenant un `valisp_types` (enumération)
  - un champ `data` contenant un `valisp_data` (union)

type	data	champs data
entier	int	INTEGER
chaine	char *	STRING
symbole	char *	STRING
cons	valisp_cons	CONS
prim	sexpr (*f) (sexpr, sexpr)	PRIMITIVE
spec	sexpr (*f) (sexpr, sexpr)	PRIMITIVE

- Une `sexpr` sera un pointeur vers un `struct valisp_object`
  - Le pointeur `NULL` correspondant à la valeur `nil`

- Coment créer un entier ?
  - J'alloue la mémoire
    - avec `valisp_malloc`
    - de quoi stocker un `struct valisp_object`
  - J'initialise le type
  - J'initialise la donnée

```c
sexpr e = new_string("toto");
printf("Hello world");
```
*.c

- Questions ?
  - où sont stockées les chaînes `"toto"` et `"Hello world"` ?
  - Que se passe-t'il si on la libére avec `free` ?
  - Comment notre `valisp_free` libère la mémoire ?

- Attention, lors de la création d'une nouvelle chaîne, il faudra copier la chaîne dans la mémoire dynamique.

- Sinon, le ramasse-miette ne pourra pas la gérer.

- Même problèmatique que pour les chaînes :

  nécessité de copier les données

- Quelle est la différence entre l'affichage d'un symbole et d'une chaîne ?
- Que doit-on renvoyer lorsque l'on crée le symbole `nil` ?

- Il faudra créer le type `valisp_cons` à partir d'une structure.
- Cette structure aura deux champs :
  - un champs `car` de type `sexpr`
  - un champs `cdr` de type `sexpr`
- Question : quelle taille fait cette structure ?

Afficher une liste

- Soit `e` une `sexpr` correspondant à un `(cons x y)` ?
- Comment afficher cette liste (non vide) proprement ?
  - en trois étapes :
    - J'affiche `"("`
    - J'appelle `affiche_liste(e)`
    - puis enfin j'affiche `")"`
- Certes, mais que fait `affiche_liste(e)` ?
  - on affiche `x`
  - puis on regarde `y`
    - si `y` est `NULL` : on quitte la fonction
    - si `y` est un cons : on affiche `" "` et on appelle `affiche_liste(y)`
    - sinon on affiche `" . "` puis `y`.

- Quel est l'affichage des expressions suivantes :
  - `(cons 1 2)` ⇒ `(1 . 2)`
  - `(cons 1 (cons 2 nil))` ⇒ `(1 2)`
  - `(1 . (2 . (3 . nil)))` ⇒ `(1 2 3)`
  - `(1 . (2 . (3 . 4)))` ⇒ `(1 2 3 . 4)`

- Aucunes difficultés
- On affichera `#<primitive>` et `#<speciale>`

Sommaire

La liste des fichiers

- Vous aurez de nombreux fichiers
  - Que vous avez précédemment écrit (vous-même)
    - `allocateur.c` et `allocateur.h`
  - Que vous allez devoir écrire/compléter
    - `erreur.c` et `erreur.h`
    - `memoire.c` et `memoire.h`
    - `types.c` et `types.h`
    - `primitives.c` et `primitives.h`
    - `mes_tests.c`
  - Que je vous fournis gracieusement (à ne pas toucher)
    - `Makefile`
    - `tests.c` et `tests.h`
    - `test_allocateurs.c`
    - `test_types.c`
    - `test_primitives.c`

- J'ai simplifié l'écriture des tests
- Il vous suffit de modifier deux fichiers pour lancer les tests.
  - `main.c`
  - `mes_test.h`
- Interdit de continuer le TP tant que votre malloc ne passe pas les tests

*.c
```
int run_test(int boolean, char *description, char* fichier, int line);
#define RUN_TEST(BOOL) run_test(BOOL, #BOOL, __FILE__, __LINE__)
```

- Que représentent les éléments suivants ?
  - `BOOL`
  - `#BOOL`
  - `__FILE__`
  - `__LINE__`
- Pourquoi la fonction qui lance les tests est une macro ?

- On va gérer deux types d'erreur
  - Les erreurs fatales : vous avez fait une erreur dans le code C
    - On crashe le programme en insultant le développeur
    - Je ne veux pas voir de *segmentation fault* !
    - Typiquement si on appelle `car` sur autre chose qu'un `cons`
  - Les erreurs Vaλisp : il y a une erreur dans le code Vaλisp
    - On affiche un joli message pour l'utilisateur
    - Le programme reste vivant
    - Utile lorsqu'on écrira les primitives

- On s'est embêter à écrire `allocateur_malloc`
- maintenant nous utiliserons `valisp_malloc`
  - appelle le premier et renvoie son résultat (*wrapper*)
  - lance une erreur Valisp en cas d'échec de l'allocation
  - plus tard, il lancera aussi le ramasse-miette dynamique
- C'est dans ce fichier qu'il y aura le code du ramasse-miette

*.c
```
enum erreurs {
    TYPAGE,             /* Paramètre du mauvais type                    */
    ARITE,              /* Mauvais nombres de paramètres                */
    NOM,                /* Variables non définie                        */
    MEMOIRE,            /* Plus de mémoire                              */
    DIVISION_PAR_ZERO,  /* Tentative d'invocation de l'infinie          */
    SYNTAXE,            /* Parseur : le code écrit n'est pas du Lisp    */
    MEMOIRE_PARSEUR     /* Parseur : la sexpr est trop grosse           */
    RUNTIME             /* Vos propres erreurs lancée depuis valisp */
};
```

- Exemple : l'exécution de `(+ 1 "saucisse")`
  - on affiche la description de l'erreur
  - dans ce TP, on quitte le programme avec un code d'erreur
  - Plus tard, on fera un goto longue distance (pour revenir dans le shell)

*.c
```
Erreur d'exécution [TYPAGE] : nécessite un entier
Fonction fautive : « + »
Valeur fautive : «"saucisse"»
```

- Pour l'instant on ignore l'environnement
- Les paramètres de la fonction sont dans le premier argument

*.c
```c
sexpr add_valisp(sexpr liste,  sexpr env) {
  sexpr a;
  sexpr b;
  /* On vérifie l'arité */
  test_nb_parametres(liste, "+", 2);
  /* On récupère les deux paramètres */
  a = car(liste);
  b = car(cdr(liste));
  /* On vérifie de typage */
  if (!integer_p(a)) erreur(TYPAGE, "+", "nécessite un entier", a);
  if (!integer_p(b)) erreur(TYPAGE, "+", "nécessite un entier", b);
  /* On fait le calcul */
  return new_integer(get_integer(a) + get_integer(b));
}
```

- Bonus : gérer un nombre quelconque d'argument :
  - `(+ 5000 100 20 3)` ⇒ 5123
  - `(+ 100)` ⇒ 100
  - `(+)` ⇒ 0

Merci pour votre attention
Questions

?

Cours 2 — Langage et encodage