



UNIVERSITÉ
CÔTE D'AZUR

Programmation impérative en C

Valisp 4. Interpréteur et parseur

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

- 🍃 Partie I. Rappels
- 🍃 Partie II. Le parseur
- 🍃 Partie III. Le débogueur
- 🍃 Partie IV. L'interpréteur
- 🍃 Partie V. La puissance des macros
- 🍃 Partie VI. Table des matières

- ▶ *Previously on Valisp*
 - Création d'une mémoire avec allocateur maison
 - Représentation des différents objets Valisp
 - ▶ Les objets sont stockés dans notre mémoire
 - ▶ Création d'une API pour y accéder
 - ▶ première abstraction de la mémoire (ensemble de données)
 - Organisation de la mémoire via l'environnement.
 - ▶ permet de définir des variables (globales)
 - ▶ permet d'y accéder et de la modifier
 - ▶ nouvelle abstraction de la mémoire (ensemble de variables)
 - ▶ permet d'inclure un ramasse-miette.
- ▶ Il reste maintenant à insuffler la vie à notre environnement
 - ▶ C'est le rôle de l'interpréteur
- ▶ Il faut aussi le nourrir,
 - ▶ C'est le rôle du parseur

- 🍃 Partie I. Rappels
- 🍃 Partie II. La parseur
- 🍃 Partie III. Le débogueur
- 🍃 Partie IV. L'interpreteur
- 🍃 Partie V. La puissance des macros
- 🍃 Partie VI. Table des matières

- ▶ Problème : Dans une chaîne lire le premier objet décrit après un indice.
- ▶ Chaque fonction aura deux résultats :
 - un code d'erreur qui est renvoyé
 - ▶ positif si tout s'est bien passé (nouvel indice situé après l'objet)
 - ▶ négatif sinon (avec un nombre différent pour chaque type d'erreur)
 - un `sexpr` stockée dans un pointeur donnée en paramètre
- ▶ `int parse_machin(char *texte, int i, sexpr *res)`
 - ▶ On commence à lire dans la chaîne `texte` à partir de l'indice `i` (inclus)
 - ▶ On stocke la `sexpr` lue dans la variable `res`.
 - ▶ On renvoie un code d'erreur associé si nécessaire
 - ▶ Sinon, on renvoie l'indice auquel on s'est arrêté

- ▶ Lors de l'appel de `int parseur(char *texte, int i, sexpr *res)`
- ▶ on commence à avancer pour ignorer les espaces et les commentaires
 - ▶ `int nettoyer_espace(char *texte, int i)`
- ▶ On regarde `texte[i]`
 - Si c'est le caractère `'\0'` : **Erreur : sexpr vide**
 - Si c'est une parenthèse fermante **Erreur : parenthèse fermante**
 - Si c'est un autre caractère, cf page suivante

- Si c'est une parenthèse ouvrante :
 - ▶ on appelle `parseur_liste` sur `res` à l'indice `i+1`
 - ▶ on renvoie le code d'erreur obtenu
- Si c'est un symbole *quote* « ' »
 - ▶ on appelle `parseur` sur une `sexpr temp` à l'indice `i+1`
 - ▶ si le résultat est négatif, on le renvoie
 - ▶ sinon on affecte à `res` la `sexpr` « (quote temp) »
 - ▶ et on renvoie le code d'erreur qu'à renvoyé la lecture de `temp`
- Si c'est un guillemet double :
 - ▶ on appelle `parseur_chaine` sur `res` à l'indice `i`
 - ▶ on renvoie le code d'erreur obtenu
- Si c'est un chiffre ou un symbole « - » suivie d'un chiffre :
 - ▶ on appelle `parseur_entier` sur `res` à l'indice `i`
 - ▶ on renvoie le code d'erreur obtenu
- Sinon :
 - ▶ on appelle `parseur_symbole` sur `res` à l'indice `i`
 - ▶ on renvoie le code d'erreur obtenu

- ▶ `int parse_entier(char *texte, int i, sexpr *res)`
- ▶ un entier est :
 - ▶ une succession de chiffre
 - ▶ commençant éventuellement par un unique signe « - »
 - ▶ le caractère suivant est un caractère parmi les espaces, ')' ou ';'
- ▶ on parcourt `texte` à partir de l'indice `i`
 - ▶ Très proche de l'exercice `lire_entier` du TP 1
 - ▶ On sauvegarde dans un `int` l'entier que nous sommes en train de lire
 - ▶ on continue tant que `texte[i]` est un chiffre
- ▶ Lorsqu'on tombe sur un indice `k` tel que `texte[k]` n'est pas un chiffre
 - ▶ si `texte[k]` n'est pas valide : **Erreur : lecture entier**
 - ▶ sinon : on convertit notre `int` en entier Valisp
 - ▶ on le stocke dans `res`
 - ▶ on renvoie `k`
- ▶ Remarque : il faudra gérer proprement le cas négatif

- ▶ `int parse_chaine(char *texte, int i, sexpr *res)`
- ▶ une chaîne :
 - ▶ commence par `"` et se termine par `"`
 - ▶ peut contenir des espaces (dont retour à la ligne) ou des `' ; '`
 - ▶ le caractère suivant est un caractère parmi les espaces, `' ) '` ou `' ; '`
- ▶ une fois les bornes de la chaînes trouvées
 - ▶ il faudra la copier dans notre mémoire dynamique
 - ▶ avant d'appeler `new_string`.
- ▶ Remarque : il faudra gérer proprement les échappements
 - ▶ `"12\"4\n6"` contient 6 caractères
 - ▶ `'1', '2', ' ', '4', '\n'` et `'6'`
- ▶ Concrètement : après avoir lu « `\` »
 - ▶ on passe au caractère suivant
 - ▶ si c'est « `n` » le caractère est `'\n'` (idem pour `'\t'` et `'\f'`)
 - ▶ sinon le caractère est celui directement lu
 - ▶ exemple : si c'est « `"` » le caractère est `'\"'`

- ▶ `int parse_symbole(char *texte, int i, sexpr *res)`
- ▶ un symbole :
 - ▶ peut contenir tous les symboles
 - ▶ sauf des espaces (dont retour à la ligne), des ';' ou des parenthèses.
- ▶ Pas d'échappements à gérer ou de difficulté particulières.

- ▶ `int parse_liste(char *texte, int i, sexpr *res)`
- ▶ on a déjà lu "("
- ▶ on commence par consommer les espaces/commentaires
- ▶ Puis :
 - si on tombe sur '\0' Erreur : `sexpr incomplète`
 - si on tombe le symbole ')', la liste est la liste NULL
 - sinon on appelle `parseur` avec un `sexpr` temp
 - ▶ si le code d'erreur k1 est négatif on le renvoie
 - ▶ sinon on lance `parse_liste` à partir du nouvelle indice k1 sur res
 - ▶ si le code d'erreur k2 est négatif on le renvoie
 - ▶ sinon on affecte « (cons temp res) » à res
 - ▶ et on renvoie k2

- 🍃 Partie I. Rappels
- 🍃 Partie II. La parseur
- 🍃 Partie III. Le débogueur
- 🍃 Partie IV. L'interpreteur
- 🍃 Partie V. La puissance des macros
- 🍃 Partie VI. Table des matières

- ▶ Comment comprendre une segmentation fault ?
 - L'idéal : ne pas faire d'erreur dans un premier temps.
 - Mettre des `printf` partout ! (C-c d sous emacs à la fac)
 - Lancer le débogueur (M-x gdb sous emacs)

- ▶ GDB : GNU débogueur
 - ▶ Pour les Mac (bouh ! c'est pas GNU) : `lldb`
 - ▶ Pour qu'il soit plus utile : ajouter l'option `-g` à `gcc`
- ▶ `gdb ./monprogramme`
 - Les commandes à connaître
 - ▶ `break` <function> (ou `b` <function>)
 - ▶ `run` (ou `r`)
 - ▶ `backtrace` (ou `bt`)
 - Permet aussi d'exécuter le code pas à pas
 - ▶ `next`
 - ▶ `step`
 - ▶ `continue` (s'arrête au prochain breakpoint)
 - ▶ `print` <variable>

- 🍃 Partie I. Rappels
- 🍃 Partie II. La parseur
- 🍃 Partie III. Le débogueur
- 🍃 **Partie IV. L'interpreteur**
- 🍃 Partie V. La puissance des macros
- 🍃 Partie VI. Table des matières

► Que vaut ?

- l'entier 3? ⇒ 3
- la chaîne "trois"? ⇒ "trois"
- le symbole trois? ⇒ dépends de l'environnement
- nil ⇒ nil
- une primitive? ⇒ elle-même
- une forme spéciale? ⇒ elle-même

► Heureusement qu'il y a les *cons*, sinon, on s'ennuierai.

- C'est dans ce cas seulement que l'on évalue une fonction

- ▶ `sexpr eval(sexpr val, sexpr env)`
- ▶ Que vaut `val` dans l'environnement `env` ?
- ▶ Trois cas :
 - Si `val` est `NULL`, en renvoie `NULL`
 - Si `val` est un symbole, on recherche sa valeur dans l'environnement
 - Si `val` est une liste (`list_p(val)`)
 - ▶ On applique la fonction `car(val)` aux paramètres `cdr(val)`
 - ▶ C'est la fonction `apply`
 - Et sinon ? `val` vaut `val`.

- ▶ On souhaite appliquer une fonction à une liste de paramètres
 - ▶ `(f e1 e2 e3)`
 - ▶ Fonction : `f`
 - ▶ Paramètre : `liste = (e1 e2 e3)`
- ▶ On commence par évaluer `f` : 4 cas possibles
 - `f` est une forme spéciale :
 - ▶ On applique `f` avec les paramètres `liste` et `env`
 - ▶ Exemples : `defvar`, `setq` et `if`
 - `f` est une primitive :
 - ▶ on applique `eval` à chaque élément de la liste
 - ▶ c'est la fonction `eval_list`
 - ▶ `((+ 1 2) x (* 10 x)) → (3 5 50)`
 - ▶ Puis on applique `f` avec comme paramètres la nouvelle liste et `env`
 - Et les deux autres cas ?

- ▶ Fondamentalement, un lambda est un cons
- ▶ Plus précisément c'est une liste d'au moins trois éléments :
 - ▶ Le mot clé `lambda`
 - ▶ la liste des paramètres
 - ▶ le reste de la liste correspond au code de la fonction.

```
(defvar somme (lambda (a b c)
  (print "Je calcule la somme de ")
  (println a ", " b " et " c)
  (+ (+ a b) c)))
```

Valisp

- ▶ Que vaut `(lambda (x) (+ x 1))`? $\Rightarrow$ `(lambda (x) (+ x 1))`

- ▶ Si le second paramètre est un symbole
 - ▶ il correspond à la liste de tous les paramètres.
 - ▶ permet d'écrire des fonctions d'arité variable

```
(defvar somme-liste (lambda (liste) ;; 1 paramètre
  (if liste ;; si liste est non nil
    (+ (car liste) (somme-liste (cdr liste)))
    0))

(defvar somme (lambda args ;; Une liste de paramètre
  (somme-liste args)))

(somme-liste (cons 1 (cons 2 (cons 3 nil))))
(somme 100 20 3) ;; ici args vaut (100 20 3)
```

Valisp

- ▶ Pourquoi je ne peux pas écrire : (somme-liste (1 2 3))?

► Il existe une forme spéciale fondamentale `quote`

- permet de ne pas évaluer une expression.
- Dans le parseur `'machin` est remplacés par `(quote machin)`.

- Exemples :

- `(1 2 3)` ⇒ Erreur : 1 n'est pas une fonction
- `(quote (1 2 3))` ⇒ (1 2 3)
- `'(1 2 3)` ⇒ (1 2 3)
- `'((+ 1 0) (+ 1 1) (+ 1 2))` ⇒ ((+ 1 0) (+ 1 1) (+ 1 2))
- `(somme-liste '(100 20 3))` ⇒ 123

► C'est la forme spéciale la plus simple à écrire :

- renvoie toujours son premier (et unique) paramètre.

- ▶ Il existe une autre méthode plus propre

```
(defvar list (lambda arg arg))  
(list 100 20 3) ;; ici arg vaut (100 20 3)
```

Valisp

- ▶ Exercices :

▶ (list 1 2 3)	⇒ (1 2 3)
▶ (list (* 1 1) (* 2 10) (* 3 100))	⇒ (1 20 300)
▶ (list 'x)	⇒ x
▶ (list)	⇒ nil
▶ (somme-liste (list 700 3 20))	⇒ 723

- ▶ Indispensable pour créer des listes!
- ▶ `quote` est à utiliser surtout pour renvoyer des symboles

- ▶ On souhaite appliquer `f` à une liste de paramètre `liste = (e1 e2 e3)`
- ▶ On commence par évaluer `f` : 4 cas possibles
 - `f` est une forme spéciale :
 - ▶ On applique `f` avec les paramètres `liste` et `env`
 - ▶ `defvar`, `setq` et `if`
 - `f` est une primitive :
 - ▶ on applique `eval` à chaque élément de la liste
 - ▶ c'est la fonction `eval_list`
 - ▶ $((+ 1 2) x (* 10 x)) \rightarrow (3 5 50)$
 - ▶ On applique `f` avec comme paramètres la nouvelle liste et `env`
 - Et les deux autres cas ? Comment évaluer un `lambda` ?

- ▶ On a **f** la fonction (**lambda** (x y z) ligne1 ... ligne10)
- ▶ Pour calculer (f 7 3 5) dans l'environnement env
 - Il suffit d'évaluer :
 - ▶ ligne1
 - ▶ ...
 - ▶ ligne10
 - dans l'environnement $(x \rightarrow 7) \rightarrow (y \rightarrow 3) \rightarrow (z \rightarrow 5) \rightarrow env$
 - et de renvoyer la dernière valeur calculée.

- ▶ On souhaite une fonction `bind` qui renvoie le nouvel environnement
- ▶ `sexpr bind(sexpr variables, sexpr liste, sexpr env)`
- ▶ Si `variables` est un `cons` :
 - ▶ `variables` et `liste` doivent avoir la même longueur
 - ▶ si `variables` vaut `(x y z)` et `liste` vaut `(1 2 3)`
 - ▶ Je veux renvoyer : `(z . 3) → (y . 2) → (x . 1) → env`
 - ▶ La construction avec une boucle `while` inverse naturellement les listes
- ▶ Si `variables` est un symbole :
 - ▶ si `variables` vaut `toto` et `liste` vaut `(1 2 3)`
 - ▶ Je veux renvoyer : `(toto . (1 2 3)) → env`

- ▶ Je veux faire apply où f est un `lambda` sur les arguments listes
- ▶ f vaut (`lambda` args ligne1 ...)
 - ▶ `car(f)` vaut `lambda` (et on s'en fout)
 - ▶ `car(cdr(f))` vaut args : ce sont les variables.
 - ▶ `cdr(cdr(f))` vaut (ligne1 ...) : c'est le corps de la fonction.

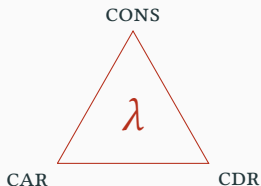
```
liste      = eval_list(liste, env);  
variables  = car(cdr(fonc));  
body       = cdr(cdr(fonc));  
env        = bind(variables, liste, env);
```

*.c

- ▶ Il me suffit alors :
 - d'évaluer chaque élément de body dans le nouvel environnement.
 - de renvoyer le dernier résultat

- ▶ Il existe un cousin aux `lambda` : ce sont les `macro`
- ▶ Pour les `lambda` :
 - ▶ J'évalue la liste des paramètres
 - ▶ J'obtiens un nouvelle environnement avec `bind`
 - ▶ J'évalue mon code dans cet environnement
 - ▶ Je renvoie la dernière valeur obtenue
- ▶ Pour les `macro` :
 - ▶ Je n'évalue pas la liste des paramètres
 - ▶ J'obtiens un nouvelle environnement avec `bind`
 - ▶ J'évalue mon code dans cet environnement
 - ▶ J'évalue la dernière valeur obtenue
- ▶ Il faut rajouter la forme spéciale `macro`
 - ▶ Que vaut `(lambda (x) (+ x 1))`
 $\Rightarrow$ `(lambda (x) (+ x 1))`
 - ▶ Que vaut `(macro (x) (list '+ x 1))`
 $\Rightarrow$ `(macro (x) (list '+ x 1))`

- ▶ Si c'est une formes spéciales :
 - ▶ J'évalue la fonction associée dans l'environnement courant
- ▶ Pour les primitives :
 - ▶ J'évalue la liste des paramètres avec `eval_list`
 - ▶ J'évalue la fonction associée dans l'environnement courant
- ▶ Si ce n'est pas une liste : erreur !
- ▶ Pour les `lambda` :
 - ▶ J'évalue la liste des paramètres
 - ▶ J'obtiens un nouvelle environnement avec `bind`
 - ▶ J'évalue mon code dans cet environnement avec `eval`
- ▶ Pour les `macro` :
 - ▶ J'obtiens un nouvelle environnement avec `bind`
 - ▶ J'évalue mon code dans cet environnement
 - ▶ **J'évalue** la dernière valeur obtenue dans l'environnement initial
- ▶ Et pour le reste : erreur !



Trois formes spéciales
triviales

lambda macro
quote

Quatre formes spéciales
avancées

if
defvar setq
while

► Il vous faudra ajouter les trois formes spéciales suivantes

- Que vaut (`quote` *machin*) ⇒ *machin*
- Que vaut (`lambda` *machin* ...) ⇒ (*lambda machin* ...)
- Que vaut (`macro` *bidule* ...) ⇒ (*macro bidule* ...)

- 🍃 Partie I. Rappels
- 🍃 Partie II. La parseur
- 🍃 Partie III. Le débogueur
- 🍃 Partie IV. L'interpreteur
- 🍃 Partie V. La puissance des macros
- 🍃 Partie VI. Table des matières

- ▶ qu'elle est l'équivalent de `i++` en `Valisp`?

```
(setq x (+ x 1))
```

Valisp

- ▶ On aimerait une syntaxe plus élégante comme :

```
(incr x)
```

Valisp

- ▶ Il est facile d'écrire une fonction qui fait une telle transformation

```
(defvar réécriture  
  (lambda (symbole)  
    (list 'setq symbole (list '+ symbole 1))))
```

Valisp

- ▶ Que vaut `(réécriture 'x)` ? ⇒ `(setq x (+ x 1))`
- ▶ On ne veut pas créer une liste, on veut exécuter du code !
 - ▶ c'est la sémantique des macro.
 - ▶ d'abord on exécute une fonction qui transforme notre code
 - ▶ puis on exécute l'expression obtenue.

- ▶ Considérons la macro ci-dessous

```
(defvar incr  
  (macro (symbole)  
    (list 'setq symbole (list '+ symbole 1))))
```

Valisp

- ▶ Le code suivant s'exécute en deux étapes.

```
(incr var)
```

Valisp

- ▶ d'abords on calcule la nouvelle `sexpr` sans évaluer les paramètres.

```
(setq var (+ var 1))
```

Valisp

- ▶ puis on exécute le code obtenu, ce qui incrémente `var`
- ▶ Cela fonctionne car en Lisp, il n'y a pas de différence en code et données.
 - ▶ un programme est une liste.
 - ▶ on parle d'homoiconicité.

```
(defun fonction (a b c)
  ligne1
  ...
  ligne10)
```

Valisp

- Les fonctions sont des variables comme les autres

```
(defvar fonction (lambda (a b c)
  ligne1
  ...
  ligne10))
```

Valisp

- On peut ainsi se permettre d'oublier les `lambda` qui deviennent des détails d'implémentation.

```
(defmacro réécriture (a b c)
  ligne1
  ...
  ligne10)
```

Valisp

- Les fonctions sont des variables comme les autres

```
(defvar réécriture (macro (a b c)
  ligne1
  ...
  ligne10))
```

Valisp

- Pour définir de nouvelles variables, on utilise la macro `let`

```
(let ((x 1)
      (y 20)
      (z 300))
  ligne1
  ...
  ligne10)
```

Valisp

- On encapsule le corps dans une fonction (`lambda`)
 - Les paramètres sont les valeurs que l'on veut créer
 - on appelle notre fonction sur les valeurs que l'on veut leur associer
 - La fonction `apply` va créer le nouvel environnement

```
((lambda (x y z) (ligne1 ... ligne10))
 1 20 300)
```

Valisp

- ▶ La forme spéciale (`if` teste alors sinon) prends trois paramètres
 - ▶ une expression booléenne
 - ▶ l'expression à exécuter en cas de succès du test
 - ▶ l'expression à exécuter en cas d'échec du test
- ▶ On a besoin de pouvoir regrouper plusieurs expressions en une.
- ▶ C'est le rôle de la macro `progn`

```
(defun abs (x)
  (if (< x 0)
      (progn
        (println x " est négatif")
        (- x))
      (progn
        (println x " est positif")
        x)))
```

Valisp

```
(progn  
  ligne1  
  ...  
  ligne 10)
```

Valisp

- ▶ On va utiliser que les `lambda` permettent d'exécuter plusieurs instructions
 - On va stocker le code dans une fonction
 - Et on va appeler cette fonction sans paramètres
 - La notation `(f)` en Valisp représente le `f()` du C ou python

```
((lambda ()  
  ligne1  
  ...  
  ligne10))
```

Valisp

```
(when (< x 10)
  ligne1
  ...
  ligne10)
```

Valisp

- Il est assez courant d'avoir un if sans else.
- Si le test est faux, l'expression renvoie nil

```
(if (< x 10)
  (progn
    ligne1
    ...
    ligne10)
  nil)
```

Valisp

```
(unless (< x 10)
  ligne1
  ...
  ligne10)
```

Valisp

- ▶ Même concept mais pour un `if` qui a seulement un `else`
- ▶ Si le test est vrai, l'expression renvoie `nil`

```
(if (< x 10)
  nil
  (progn
    ligne1
    ...
    ligne10))
```

Valisp

- ▶ Ce n'est pas un cours de Lisp.
- ▶ Le but est de vous montrer que votre langage est plus qu'un simple jouet
 - Nous avons non seulement créé un langage de programmation
 - Mais ce langage est lui même auto-programmable !
- ▶ C'est une spécificité unique qu'aucun autre langage ne partage
 - même s'il y a des tentatives (R, Ruby, Elixir, Rust, etc)
 - aucun n'est aussi simple ni aussi puissant
- ▶ La syntaxe particulière de lisp est le prix à payer pour ce pouvoir

Merci pour votre attention

Questions



Hello world !



Cours 4 — Interpréteur et parseur

Partie i. Rappels

Valisp

Partie ii. La parseur

L'API du parseur

Le code 1/2

Le code 2/2

Les entiers

Les chaînes

Les symboles

Les listes

Partie iii. Le débogueur

Segmentation fault

Le débogueur

Partie iv. L'interpreteur

Interprétons

La fonction eval

La fonction apply 1/2

Les lambdas

lambda et paramètres

la forme spéciale quote

la fonction list

La fonction apply 1/2

Appliquer un lambda

L'ajout des liaisons

Concrètement

Et les macros

La fonction apply 2/2

La photo de famille

Les trois nouvelles formes spéciales

Partie v. La puissance des macros

Le concept de macro

La macro incr

La macro defun

La macro defmacro

La macro let

Le problème du if

La macro progn

La macro when

La macro unless

Conclusion

Partie vi. Table des matières