# Programmation impérative en C

## Valisp 5. Finitions et conclusions

UNIVERSITÉ
CÔTE D'AZUR

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

- *Previously on Valisp*
  - Gestion de la mémoire
    - `allocateur.c` : construction de la mémoire
    - `memoires.c` : surcouche de haut niveau + ramasse-miettes
    - `environnement.c` : associations variables ↔ valeurs (≈ pile)
  - Gestion des calculs
    - `types.c` : création des expressions (`sexpr`) ; fait le lien avec la mémoire
    - `primitives.c` : fonctions fournis avec le langage
    - `interpreteurs.c` : expression (`sexpr`) ⟶ résultat (`sexpr`)
  - Interface utilisateur
    - `parseurs.c` : chaînes ⟶ expression (`sexpr`)
    - `erreurs.c` : gestions des erreurs
    - `valisp.c` : le REPL

# Sommaire

- Lorsque `valisp_malloc` échoue :
  - on interrompt le calcul avec une erreur (retour au REPL)
  - puis le REPL lance le ramasse-miette en attendant le prochain calcul
  - dès qu'un calcul est long (boucle), il échoue pour cause de mémoire.
- Nouvelle stratégie :
  - lorsque `allocatateur_malloc` échoue
  - on lance le ramasse-miette
  - et on relance `allocatateur_malloc`
  - (en cas de nouvel échec : erreur)
- Problème : sur quel environnement lancer le ramasse-miettes ?
  - l'environnement global ? …ne marche pas.
  - l'environnement local ? …ne marche pas non plus.
  - tous les objets ne sont pas accessibles depuis l'environnement
  - exemple : variables locales des primitives C.

- On crée un pile pour indiquer les pointeurs à sauvegarder
  - À chaque malloc réussi, on empile le pointeur
- Lors de l'appel du ramasse-miettes :
  - On marque (pour conserver) chacun des pointeurs
  - ainsi que les objets vers lesquelles ils pointent
- Problème on ne libère plus rien !

# Création d'une pile

- Une pile est constituée de deux variables :
  - un tableau statique (c'est à dire global)
  - l'indice du haut de pile (statique lui aussi)
- L'API de la pile
  - `void init_pile(sexpr env)`
    - Par défaut on empile l'environnement global pour le sauvegarder
  - `int pile_nouveau_cadre(void)`
    - On renvoie l'indice du haut de pile
  - `void pile_fin_cadre(int i)`
    - On restaure l'indice du haut de pile (la pile diminue)
  - `void pile_ajout(sexpr s)`
    - `PILE[HAUT++] = s`

- Comment et quand utiliser la pile ?

- On empile lors de l'allocation (`valisp_malloc`)

- Lors de l'appel d'une fonction (`apply`) :
  - on commence par sauvegarder l'indice de haut de la pile
    - c'est le rôle de la fonction `pile_nouveau_cadre`
  - À la fin de l'appel
    - on restaure la pile (`pile_fin_cadre`)
    - puis on empile la valeur à retourner
    - (qui se retrouve sur le cadre de pile de la fonction appellante)

*.c
```
sexpr apply(sexpr fonction, sexpr variables, sexpr env) {
    if (blabla) {
        ...
        return res;
    }
    if (bloublou) {
        ...
        return res;
    }
}
```

- Le goto est pratique pour *nettoyer avant de quitter*

*.c
```
sexpr apply(sexpr fonction, sexpr variables, sexpr env) {
    if (blabla) {
        ...
        goto fin;
    }
    if (bloublou) {
        ...
        goto fin;
    }
fin:
     ... /* On s'occupe de la pile */
     return rés
}
```

- On réserve un pointeur avec `allocateur_malloc` (bas-niveau)
- En cas d'échec :
  - On lance la ramasse-miettes dynamique
  - On re-réserve un pointeur avec `allocatateur_malloc`
  - en cas de nouvel échec on lance une erreur !
- on sauvegarde le pointeur dans notre pile
- on renvoie le pointeur

- Le ramasse-miettes statique parcourt l'environnement globale (`sexpr`)
- Maintenant on va parcourir chaque pointeur (`sexpr`) de notre pile
  - on utilise la même fonction que pour le ramasse-miettes statique
  - `ramasse_miette_parcourir_et_marquer`
- Puis on appelle `ramasse_miette_liberer`
- Il reste un petit problème…
  - il ne faut pas empiler les pointeurs des *data* des chaînes
  - `valisp_malloc` pour les `sexpr` (entier, chaîne, symbole, etc)
  - `valisp_malloc_data` pour le champs `DATA` des chaînes et des symboles

# Sommaire

Valisp

```
(defun plus-n (n)
    (lambda (x) (+ x n)))

(defvar incr (plus-n 1)) ;; (lambda (x) (+ x 1)

(defun suivant (n)
    (incr n))
```

- Que vaut `(suivant 10)`
  - 11 ?
  - 20 ?

- En fait `incr` vaut `(lambda (x) (+ x n))`
  - Et que vaut `n` ?

- On rajoute une forme spéciale
  - et un nouveau type `closure` (en plus des entiers, symboles, etc.)
  - codé avec un COUPLE (comme les CONS)
  - `(env-de-definition . fonction-lambda)`

Valisp

```
(defun plus-n (n)
    (closure (x) (+ x n)))

(defvar incr (plus-n 1));(lambda (x) (+ x n))
                        ; dans l'environnement (n . 1) → env
(defun suivant (n)
    (incr n))
```

- `sexpr new_closure(sexpr env, sexpr fonction)`
  - attention à l'affichage (boucle infinie)
  - l'environnement local pointant vers l'environnement globale
- `sexpr get_closure_env(sexpr closure)`
- `sexpr get_closure_lambda(sexpr closure)`
- On modifie `apply` pour évaluer le lambda dans l'env. de définition.

# Sommaire

- On conserve un tableau de tout les symboles crées
- La fonction `new_symbol`
  - regarde si le symbole existe déjà
  - Si oui, on récupère le pointeur.
  - sinon, on copie la chaîne du symbole en mémoire
- Permet d'éviter de nombreuses allocations
- Attention au ramasse-miettes !
  - Il faut mettre à jour le tableau après le ramasse-miettes

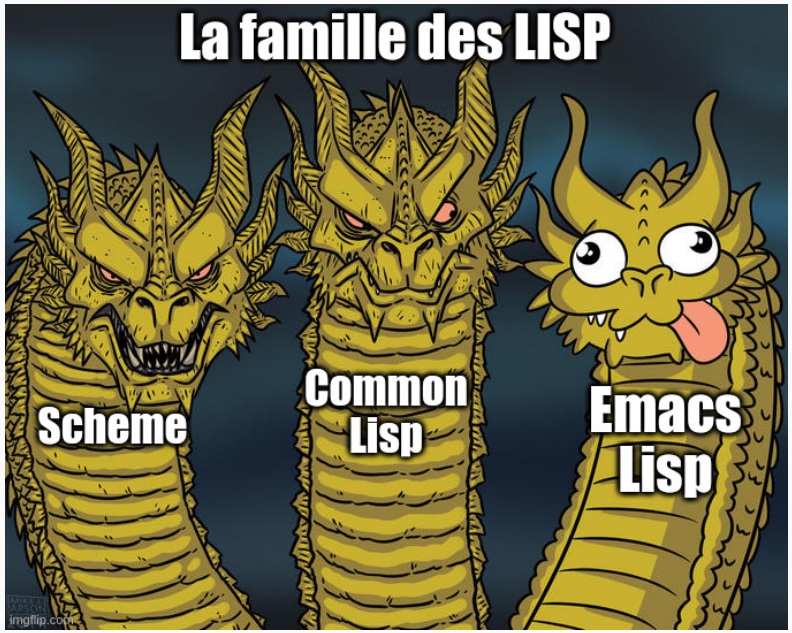
- L'encodage de la mémoire
  - on n'utilise jamais le `free`
  - un liste simplement chaînée suffit
  - permet de doubler la mémoire (et moins d'appelde ramasse-miettes)
- Précalculer les macro
  - On développe les macro à chaque fois qu'on les rencontre
  - très coûteux
  - il faudrait le faire uniquement lors de `defun`
  - on peut le vaire en Vaλisp !

# Sommaire

:)


- *Strucure and interpretation of computer programm*
- Introduction de la liaison statique (closures)

:)




- Emacs Lisp
  - C'est le plus récent (1984)
  - mais aussi le plus archaïque
  - et un des plus minimaliste
- Proche de Common Lisp
  - La syntaxe est quasi identiquqe

- Emacs n'est pas un éditeur de texte
- C'est un intepréteur Lisp
  - Chaque touche appelle une fonction
  - On peut combiner ces fonctions pour en écrire de nouvelle
  - Et les associer à des touches
- C'est le dernier héritage des hackeurs du MIT
  - un logiciel avec un contrôle complet
  - un bac à sable
- Aujourd'hui la mode est de tout faire en javascript dans le navigateur
  - Même concept
  - La liberté en moins

# Sommaire

# Conclusion

- L'informatique est riche en langage divers
  - Lisp, Erlang, Prolog, Smalltalk
  - Ces langages donnent une autre définition au mot *programmer*
- Le C permet de comprendre comment fonctionne les machines
  - Je vous encourage à jeter un coup d'œil à Rust
- Vous devez maîtriser vos outils !
  - Privilégier les outils libres et hackables

# Merci pour votre attention
# Questions

?

# Cours 5 — Finitions et conclusions