Programmation impérative en C

Valisp 3. Environnement et ramasse-miettes

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

- Qu'est ce qu'un environnement ?
- C'est une association entre un nom (de variable) et une valeur.

variable	valeurs
`x`	`3`
`y`	`2`
`z`	`"toto"`
`+`	`#<primitive>` (`add_valisp`)

- Chaque association va être codé par un couple (cons)
  - `(cons x 3)` qui s'affiche `(x . 3)`.
- Notre environnement sera une liste chaînée de couples.
  - ENV vaut `(x . 3)` → `(y . 2)` → ... → `nil`
  - Questions :
    - Que vaut `car(ENV)` ? et `cdr(ENV)` ?
    - Comment récupérer le nom de la première variable ? de la seconde ?

- Si je définis une nouvelle variable, deux possibilités :
  - C'est une redéfinition.
    - Dans ce cas, il suffit de mettre à jour le `cdr`
  - La variable n'existe pas encore.
    - Je crée le `cons (nom . valeur)`
    - Je l'ajoute **à la fin** de la liste.
- L'API de notre environnment :
  - Que vaut `x` ? `trouver_variable`
  - `(setq x 3)` `modifier_variable`
  - `(defvar x 3)` `definir_variable_globale`

- Parcours d'un tableau de taille `N`

```c
for (i=0; i<N ; i++) { … }
```
*.c

- Parcours de notre liste doublement chaînée

```c
for (i=0; i!=bloc_suivant(i) ; i=bloc_suivant(i)) { … }
```
*.c

- Parcours d'un environnement (listes chaînées)

```c
for (liste=env; liste!=NULL ; liste=cdr(liste)) {
    a = car(liste); /* a est de la forme (nom . valeur) */
    nom = car(a);
    val = cdr(a);
    …
 }
```
*.c

- Lorsque c'est possible, préférez le `for` au `while`

Sommaire

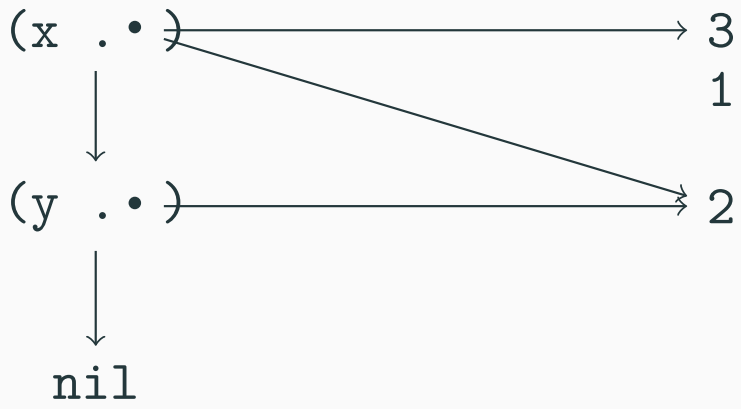
Principe

- Comment savoir si une ressource peut-être libérée ?
- On regarde chaque objet de l'environnement
  - ainsi que tous les objets vers lesquelles il pointe
  - et ainsi de suite récusivement.

  - On marque les blocs de tous les objets rencontrées
  - Les objets restant sont inatteignables (ils ne servent donc plus à rien)

Exemple

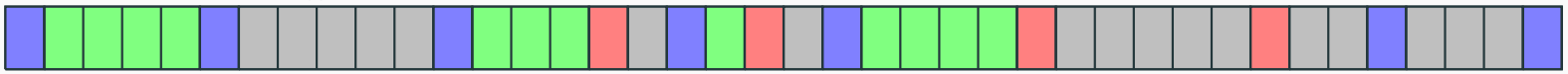
- Au début `x` vaut 2 et y vaut 2
- On exécute `(setq x (+ x 1))`
  - on crée l'entier 1
  - on crée l'entier 3
  - on modifie la liaison



- Comme aucun objet ne pointent vers 1, on peut le supprimer
  - techniquement : l'objet 1 n'est atteignable depuis l'environnement
  - c'est un problème de parcours de graphes

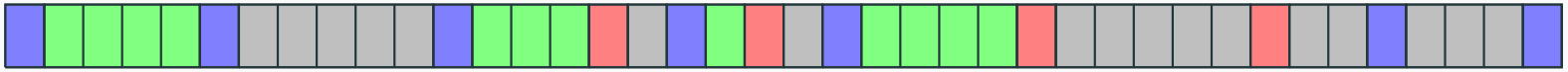
- Je commence avec l'objet *environnement* (c'est une `sexpr` de type `cons`)
- Si l'objet que je parcours est déjà marqué, je m'arrête
  - Sinon je le marque
- Ensuite, selon son type
  - Si l'objet est une chaîne est un symbole,
    - S'il n'est pas déjà marqué, je le marque
  - Si l'objet est un cons
    - Je parcours récursivement le car
    - Je parcours récursivement le cdr
  - Si l'objet est un entier ou une primitives, il n'y a rien à faire

- On a marqué tous les blocs des objets parcourus
  - On ne marque pas les objets, mais les blocs les précédant.
  - Ici les blocs marqués seront représentés en rouge
  - Les autre sont en bleu
- On cherche à maintenant nettoyer.
  - On veut passer de :

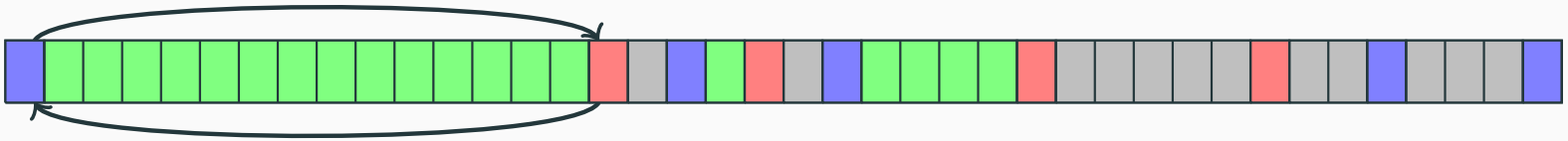


  - à :

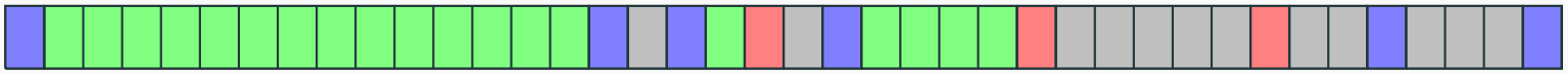
  - concrètement
    - on libère tous les blocs non marqués
    - deux blocs consécutifs ne peuvent pas être libre (sinon, on fusionne)
    - on enlève toutes les marques



- On parcours les blocs.
  - Si on tombe sur un bloc libre (non marqué = ici en bleu)
    - on cherche le prochain bloc à sauvegarder.
    - on relie les deux blocs ensembles.
    - on reprend le parcours à partir du bloc suivant



  - Si on tombe sur un bloc à conserver : on enlève sa marque



Sommaire

- `jmp_buf` est un type permettant une sauvegarde de l'état du programme.
- `setjmp(*buffer)` permet de sauvegarder l'état et renvoie 0.

*.c
```c
#include <setjmp.h>

jmp_buf *buf  = jump_buffer();

if (!setjmp(*buf)) {
    /* setjmp a renvoyé 0*/
    /* Je viens de faire une sauvegarde : je suis content */
    du code et des appels de fonctions
    du code et des appels de fonctions
    du code et des appels de fonctions
} else {
    /* setjmp a renvoyé val != 0. Oh mon dieu !!!  */
    /* Si je suis ici c'est que longjmp m'a tué    */
    /* Je viens de ressusciter au niveau du setjmp */
    printf(couleur_rouge);
    afficher_erreur();
    printf(couleur_defaut);
}
```

*.c
```c
longjmp(buf,1); /* 1 = val (retour de  setjmp) */
```

`lngjmp` = danger

- Le Long Jump est une pratique dangereuse
- Risque de fuite mémoire
  - La pile d'appel est détruite (jusqu'au `setjmp`)
  - Toutes les variables de piles qui pointaient vers le tas ont disparues
  - Impossible de les libérer à présent…
- Le problème ne se pose pas ici
  - Toutes nos variables sont allouées dans notre mémoire dynamique
  - Donc pas de tas.
  - Il suffit de lancer le ramasse-miette pour tous libérer

- Pour l'instant en cas d'erreur, on faisait `exit(1)`
- Dorénavant, en cas d'erreur :
  - on stocke les données de l'erreur dans des variables globales
  - on fait un long jmp vers le REPL (le shell lisp)
  - On affiche alors l'erreur à l'utilisateur
- Il suffit :
  - de définir dans erreur.c un `jump_buf buf` (sans pointeur)
  - d'écrire une fonction `jump_buffer` qui renvoit un pointeur vers `buf`
  - (Ce buffer sera utilisé par le `setjmp` dans le main)
  - puis d'appeller `longjmp(buf,1)` au lieu de `exit(1)` dans `erreur`

Sommaire

Valλisp

```
;; Environnement ((x . 3) (y . 7) …)

(+ (* 4 1)
   (* x 10)
   (* y 100))
```

- Pour évaluer la somme, je dois d'abord évaluer les arguments
- `(+ … … …)` ⇒ 734
  - `(* 4 1)` ⇒ 4
    - `4` ⇒ 4
    - `1` ⇒ 1
  - `(* x 10)` ⇒ 30
    - `x` ⇒ 3
    - `10` ⇒ 10
  - `(* y 100)` ⇒ 700
    - `y` ⇒ 7
    - `100` ⇒ 100

- Pour évaluer `(procédure arg1 arg2 … argn)`
  - J'évalue chacun des arguments :
    - arg1 ⇒ `val1`
    - arg2 ⇒ `val1`
    - …
    - argn ⇒ `valn`
  - J'appelle la primitive procédure avec comme paramètres (en C) :
    - la liste des paramètres (en Valisp) : `(val1 val2 … valn)`
    - l'environnement
- Questions ?
  - Pourquoi les primitives ont-elles besoin de l'environnment ?
- Comment s'exécutent `(defvar x 3)`

```
;; ((y . 7) … )
(defvar x 3)
```
Valisp

- `(defvar x 3)`
  - `x` ⇒ `ERROR : x non défini`
  - 3
- La règles précédentes ne fonctionnent pas.
- pour certaines procèdures, **il ne faut pas** évaluer les arguments.
- On parle de formes spéciales

- Exemple d'utilisation `(defvar toto (+ 2 1))`

*.c

```
sexpr defvar_valisp(sexpr liste, sexpr env) {
  sexpr nom;
  sexpr exp;
  sexpr res;
  test_nb_parametres(liste,"defvar",2);
  nom = car(liste);
  exp = car(cdr(liste));

  if (!symbol_p(nom)) {
      erreur(TYPAGE,"defvar",
                    "Le 1er paramètre doit être un symbole",
                    nom);
  }
  res = eval(exp, env); /* Il faut évaluer à la main le  */
  definir_variable_globale(nom,res); /* second paramètre */
  return res;
}
```

- `eval` (l'interpréteur) sera au programme de la semaine prochaine.
  - Ce sont les formes spéciales qui ont besoin de l'environnement.
  - car `eval` en a besoin
- `definir_variable_globale` est dans le TP 3.

Sommaire

- Première étape : la mémoire
  - On crée le tableau dynamique
  - On crée l'environnement qui ne contient qu'une valeur : `t`
  - Que vaut le symbole `t` ? Ça vaut `t` ! (booléen `true`)
  - Et comment est codé `false` ?
- Deuxième étape : les primitives et le formes spéciales
  - on ajoute à l'environnement des liaisons symboles/primitives
  - exemple : `+` avec `add_valisp` et `<` avec `less_than_valisp`
  - exemple : `defvar` avec `defvar_valisp` (spéciale)

*.c
```
void charger_primitives() {
charger_une_primitive("+", add_valisp);
charger_une_primitive("=", equal_valisp);
charger_une_primitive("<", less_than_valisp);
...
charger_une_speciale("defvar",  defvar_valisp);
charger_une_speciale("setq",    setq_valisp);
}
```

- Troisième étape : on lance le REPL

- Le REPL est le shell du langage avec lequel on interagit
- Read-Eval-Print LOOP
  - Read : C'est le parseur
  - Eval : C'est l'interpréteur
  - Print : C'est l'affichage du résultat
- Au début de chaque itération
  - on place un setjmp
  - et on affiche un message d'erreur s'il se *déclenche*

Merci pour votre attention
Questions

?

Cours 3 — Environnement et ramasse-miettes