Comment améliorer son efficacité

Les Japonais ont inventé une méthode appelée “Kaïzen”.

La méthode est très simple : il faut améliorer une petite chose tous les jours.

Exemples de choses à améliorer (Emacs)

- ne plus utiliser la souris pour changer la fenêtre Emacs (utiliser C-x o) ;
- utiliser M-m (ou ESC m) pour aller au début de la ligne :
- utiliser C-[à la place de ESC et de M- (plus rapide) ;
- utiliser les arguments numériques (C-u) ;
- ne plus utiliser les touches flèches, mais C-n, C-p, C-b, C-f à la place ;
- ne plus utiliser les touches “page suivante” et “page précédente”, mais C-v, M-v ;

Exemples de choses à améliorer (Emacs)

- commencer à utiliser la recherche incrémentale (C-s) ;
- commencer à utiliser la complétion dynamique (M-/) ;
- commencer à utiliser les abréviations ;

Exemples de choses à améliorer (Bash)

- utiliser la recherche dans l'historique (C-r) ;
- utiliser les commandes Emacs (C-a, C-e, C-k) ;
- utiliser la boucles `for` ;

Exemples de choses à améliorer (Lecture)

- lire un chapitre par semaine dans la documentation Emacs ;
- lire un livre par an sur la programmation ;
- lire le code de GNU Emacs ;

Exemples de choses à améliorer (Autres)

- apprendre à utiliser un clavier (dactylographie) ;
- éviter la souris ;
- mettre la touche Ctrl sur la touche actuelle Caps Lock ;
- utiliser une touche autre que Del pour effacer un caractère ;

http://dept-info.labri.fr/˜strandh/Teaching/USI/Common/Dactylographie/cours.html

General Problem Solver (GPS)

Un programme classique de l'intelligence artificielle.

Le but du programme est de chercher des solutions à des problèmes de planification.

Code initial (en un langage qui n'est plus utilisé) très long et très complexe.

Nous allons étudier la version de Peter Norvig de son livre "Paradigms of Artificial Intelligence Programming".

GPS

```
;;; -*- Mode: Lisp; Syntax: Common-Lisp; -*-
;;; Code from Paradigms of Artificial Intelligence Programming
;;; Copyright (c) 1991 Peter Norvig

(defstruct op "An operation"
  (action nil) (preconds nil) (add-list nil) (del-list nil))
```

GPS

```
(defparameter *school-ops*
  (list
    (make-op :action 'drive-son-to-school
             :preconds '(son-at-home car-works)
             :add-list '(son-at-school)
             :del-list '(son-at-home))
    (make-op :action 'shop-installs-battery
             :preconds '(car-needs-battery shop-knows-problem shop-has-money)
             :add-list '(car-works))
    (make-op :action 'tell-shop-problem
             :preconds '(in-communication-with-shop)
             :add-list '(shop-knows-problem))
    (make-op :action 'telephone-shop
             :preconds '(know-phone-number)
             :add-list '(in-communication-with-shop))
    (make-op :action 'look-up-number
             :preconds '(have-phone-book)
             :add-list '(know-phone-number))
    (make-op :action 'give-shop-money
             :preconds '(have-money)
             :add-list '(shop-has-money)
             :del-list '(have-money))))
```

GPS

```
(defvar *state* nil "The current state: a list of conditions.")

(defvar *ops* nil "A list of available operators.")
```

GPS

```
(defun GPS (*state* goals *ops*)
  "General Problem Solver: achieve all goals using *ops*."
  (if (every #'achieve goals) 'solved))

(defun achieve (goal)
  "A goal is achieved if it already holds,
  or if there is an appropriate op for it that is applicable."
  (or (member goal *state*)
      (some #'apply-op
            (find-all goal *ops* :test #'appropriate-p))))

(defun appropriate-p (goal op)
  "An op is appropriate to a goal if it is in its add list."
  (member goal (op-add-list op)))

(defun apply-op (op)
  "Print a message and update *state* if op is applicable."
  (when (every #'achieve (op-preconds op))
    (print (list 'executing (op-action op)))
    (setf *state* (set-difference *state* (op-del-list op)))
    (setf *state* (union *state* (op-add-list op)))
    t))
```

GPS

```
* (gps '(son-at-home car-needs-battery have-money have-phone-book)
       '(son-at-school)
       *school-ops*)
(EXECUTING LOOK-UP-NUMBER)
(EXECUTING TELEPHONE-SHOP)
(EXECUTING TELL-SHOP-PROBLEM)
(EXECUTING GIVE-SHOP-MONEY)
(EXECUTING SHOP-INSTALLS-BATTERY)
(EXECUTING DRIVE-SON-TO-SCHOOL)
SOLVED
```

GPS

```
* (gps '(son-at-home car-needs-battery)
       '(son-at-school)
       *school-ops*)
NIL

* (gps '(son-at-home car-works)
       '(son-at-school)
       *school-ops*)
(EXECUTING DRIVE-SON-TO-SCHOOL)
SOLVED
```

Présentation du Projet

Implémentation en Lisp de requêtes dans une base de données contenant des informations sur des livres.

L'origine de la base est un fichier XML, traduit en expressions Lisp.

Cette base doit être lu avec **`read`**.

La version Lisp

Alors que la version XML est un arbre, la version Lisp est un DAG (graphe sans cycles).

Chaque livre, auteur, maison d'édition, etc figure dans la base *une seule fois*.

Par conséquent, il est possible de comparer ces objets avec la fonction `eq`.

Cela améliore considérablement le temps de calcul.

Description de la base

La base entière est indiquée avec **`db`** :

```
(db (authors (author ...) (author ...) ...)
    (publishers (publisher ...) (publisher ...) ...)
    (books (book ...) (book ...) ...)
    (reveiws (review ...) (review ...) ...)
    (sources (source ...) (source ...) ...)
    (prices (price ...) (price ...) ...))
```

Description de la base

Structure d'un auteur :

```
(author (last "Graham") (first "Paul"))
```

Les champs ne sont pas toujours dans cet ordre, et d'autres champs peuvent exister.

Description de la base

Structure d'une maison d'édition :

```
(publisher (name "Prentice Hall"))
```

Les champs ne sont pas toujours dans cet ordre, et d'autres champs peuvent exister.

Description de la base

Structure d'un livre :

```
(book (year 1995)
      (title "On Lisp'')
      (author (last "Graham") (first "Paul"))
      (publisher (name "Prentice Hall"))
      (price 39.95))
```

Les champs ne sont pas toujours dans cet ordre, et d'autres champs peuvent exister.

Les champs `author` et `publisher` sont `eq` aux champs correspondants dans la partie `authors` et `publishers` respectivement.

Description de la base

Structure d'un compte rendu :

```
(reviews (book ...)
         (text "A great successor to `ANSI Common Lisp'"))
```

Les champs ne sont pas toujours dans cet ordre, et d'autres champs peuvent exister.

Le champ `book` est `eq` au champ correspondant dans la partie `books`.

Description de la base

Structure d'une source :

```
(source (name "www.amazon.com"))
```

Description de la base

Structure d'un prix :

```
(price (book ...)
       (source ...)
       (euros 35.95))
```

Les champs ne sont pas toujours dans cet ordre, et d'autres champs peuvent exister.

Les champs `book` et `source` sont `eq` aux champs correspondants dans la partie `books` et `sources` respectivement.