Cheat sheet Lisp (orienté common Lisp)

- Qu'est ce qui peut être appelé Lisp:
  langage:
  - minimaliste (basé sur un Σ de règle très petit)
  - homoiconique
  - extensible

Syntaxe

- Uniq[t] parenthèses et des parenthèse
- aucun mot clé réservé
- appel de fct° en ordre préfixe
  ex: (+ 1 2)
- nomage: • n'importe quoi
  • |...| si on veut mettre des parenthèse ds le nom
- Expression: • exp. littérale
  - s'evalue à elle-même
  • exp. combinée (appliqu° fct° à des args)
  - pas de distinct° entre var / fct°
  - première arg entre ()
    → (func arg1 ...)
  - ij préfixe
  - opérateur variadiq
  - imbricat° naturelle
  • exp. abstraite:
  abstract° fonctionnelle
  (ll est fct°)

#typage

- typage dynamique :
  - valeur et non les variables qui sont typée
  - vérificat° du type au runtime
- On a accès au type au runtime avec :

```
(type-of var)
(typep var 'integer)
```

→ convert° en common lisp
→ fonction prédicat

⇒ typep : est ce que arg1 est de type arg2

- Typage dynamique implique lenteur car rejoute vérificat° au runtime

⇒ Compromis perf. / expressivité à faire

- On peut aussi faire du typage statique en CL

```
(declare (type fixnum foobar))
```

declare est une indicat° pour le compilo ≠ expression

→ foobar est de type fixnum

et si on veut opti le runtime on peut supprimer vérif :

```
(declare (optimize (speed 3) (safety 0) (debug 0)))
```

Expressions conditionnelles

- booléen :
  - pas un type de base en CL
  - tout ce qui n'est ni nil ou () → true
    (Faux)
  - and / or / not : opérateur vérifiés
    → (and arg1 arg2 arg3 ...)

- if / then / else → macro en CL
  - if seul ≠ else
    (if condition si-vrai si-faux)
    - else if soit avec combinaison de if soit macro cond :

```
(cond (cond1  si-vrai1)
      (cond2  si-vrai2)
      ⋮
      pour un cond° default (t default))
```

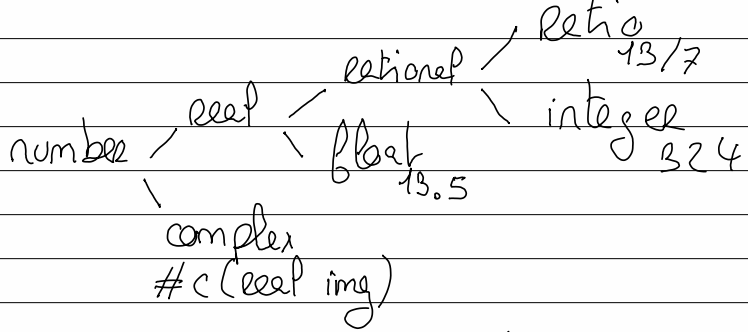
- pattern matching possible avec case

```
(case month
  ((janv march dec) si-vrai)
  ((feb august)     si-vrai)
  (otherwise        default))
```

- pour pattern matcher type → typecase

Arithmétique

- type number → type général qui rassemble les ces sous type



- transtypage implicite si possible

  ratio vers entier : 12/3 → 4
  complex vers reels : #c(13 0) → 13

- transtypage explicite : (float 13) → 13.0

- transtypage automatique vers float/complexe si passé en args d'une fct qui attend un des 2

  (+ 3.2 4) = (+ 3.2 4.0)

- prédicat pr connaître type d'un number : integerp, floatp, ...

- opérat° :
  - extract° :
    truncate, floor, ceiling, round

- arithmétique:
  + - * div mod abs signum
- relationnel:
  < > <= >= /= (≠ différent)
  = eql equal (et d'autre)
- Exponentiation
  exp, log, expt, ^, sqrt
- Trig:
  pi sin cos tan acos ...

Data type

- liste: type natif des langages fonct°
  (e1 e2 e3 ...)
- type hétérogène: (1 "string" 'c' 3)
- Prédicat listp envoie true si vce liste
- liste vide: () stricte° équivalent à nil
- représenté ss forme: 
- Constructeur (≠ générateur car constructeur → élémentaire):
  (cons head tail)

- générateur :
  - fonction list
    (list 1 2 "string")
  - make-list
    (make-list size initial-elt)

- accesseur :

→ de par structure [HTT]→

↳ car pr accéder à la tête
cdr pr accéder à la queue
+ combinaison c[ad]+r
avec un max de 4
→ (list 1 2 3 4 5)
(caddar ↑) → 3
(cddddr ↑) → '(3 4 5)

- last, (nthcdr n list) → renvoie un cons
- first ... tenth + (nth n list)

- Séquences : - vision d'une liste comme une série ordonnée d'éléments
  - ex : chaîne de char
  - hétérogène

- Chaîne de caractère :
  char : #\c
  ↳ prédicat : characterp

char ↔ int : char-code / code-char

comparaison caractères
char=, char>=, char/=, etc

Chaîne : implémentée sous forme de vecteur
Le prédicat : stringp
⇒ liste de char ≠ string

- Manipulat° de séquence / liste :

recherche (member elt lst) (pour les listes)
(find elt seq) (pour les types de séquence)

Indexat° (position elt seq)
Filtrage (remove elt seq)
Concaténat° : append / concatenate
(↳ ne copie pas les args, juste plie)
Sous-séquence : (subseq seq first &optional last)
Autre : reverse / length / ...

Variable

Variable globale :

(defvar var value)
(defparam var value) → pas réévaluée une 2ème fois

Variable locale :

(let ((a 2)
      (b "coucou"))

↳ ne permet pas de faire d'interférence

(let* ((a 2)
       (b (+ a 1)))

set : (setf var new-value)

fonction

définit° : (defun name-func (param)
)

fonction anonyme : (lambda (param))