Survol de la syntaxe Common Lisp

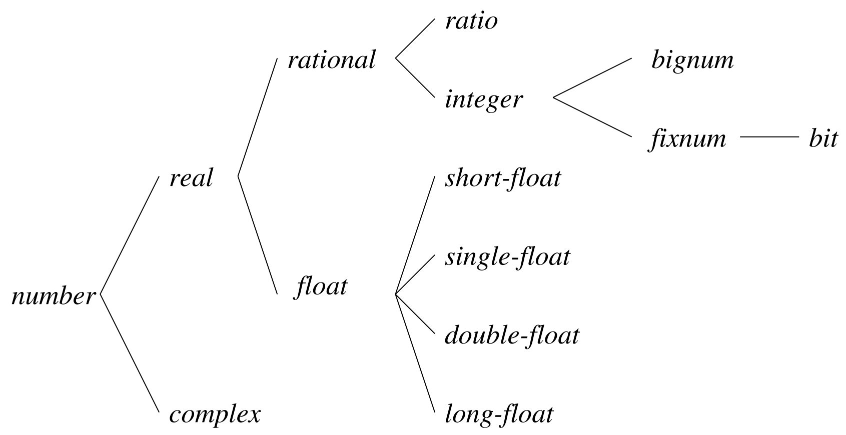
Support PG104

M.Desainte-Catherine

ENSEIRB, département d'informatique

January 17, 2007

Arborescence des types numériques



(R. Strandh)

Les entiers

- Types
  - bignum : taille non limitée (seulement par la mémoire).
  - fixnum : représentés dans un mot (au moins 16 bits).
- Constantes
  - most-positive-fixnum
  - most-negative-fixnum

Example

```
* most-positive-fixnum
536870911
* most-negative-fixnum
-536870912
```


Les ratios

- Accesseurs : **numerator**, **denominator**
- Prédicats : **ratiop**

Example

```
* (/ (* 536870911 10) 9)
5368709110/9
* (numerator 5368709110/9)
5368709110
* (denominator 5368709110/9)
9
* (ratiop 5368709110/9)
T
* (rationalp 5368709110/9)
T
```

Les flottants

Ex: 23.2e10
Le type d'un résultat d'un calcul est choisi parmi les sous-types

- **short-float**,
- **single-float**,
- **double-float**,
- **long-float**

et correspond au type le plus précis des arguments.

Les complexes

Ils sont sous forme cartesienne avec parties réelle et imaginaire étant des réels.

Example

```
* (sqrt -1)
#C(0.0 1.0)
* #C(1.0 0.0)
#C(1.0 0.0)
```


Les booléens

- Constantes : **t** et **nil** (qui est équivalent à **()**)
- Tous les objets à part le symbole **nil** (ou **()**) valent **vrai**.
- L'objet **t** est utilisé comme valeur vrai, quand aucune autre valeur ne paraît plus pertinente.
- Règles d'écriture :
  - utiliser nil dans un contexte booléen
  - utiliser '() pour signifier fin de liste dans une structure de données.
  - utiliser () dans les macros.

Prédicats

- typage : **numberp**, **realp**, **complexp**, **rationalp**, **ratiop**, **integerp**, **floatp**
- Nombres : **zerop**, **plusp**, **minusp**
- Entiers : **evenp**, **oddp**
- Comparaisons : $= \quad /= \quad < \quad <= \quad > \quad >=$ sur les réels.
- Égalités et inégalités sur les complexes.
- Nombre d'opérandes supérieur strictement à 0.
- **null** : égalité à nil (ou '())

Opérations numé riques

- Arithmétiques :+, -, *, / (nb quelconque d'arguments)
- Unaires : **1+**, **1-** (incrémentation, décrémentation)
- **max** et **min** : arguments réels.
- trigonométrie : **sin**, **cos**, **tan**, **asin**, **acos**, **atan**, **sinh**, **cosh**, **tanh**, **asinh**, **acosh**, **atanh**
- **log**, **exp** (naturelle), **expt** (d'une base arbitraire).
- opérations bit à bit (sur les entiers) : **logand**, **logandc1**, etc.
- conversions, troncatures, arrondis : **floor**, **ceiling**, **truncate**, **round**.

Opérations booléennes

- **and** : stop au premier argument évalué à faux
- **or** : stop au premier argument évalué à vrai
- **not**

Ce sont des opérateurs spéciaux. Les opérateurs *and* et *or* admettent n arguments, $n \geq 0$.

Example

```
* (and)
T
* (or)
NIL
```


Les symboles

Suite de caractères quelconques ne pouvant pas être interprétée comme un nombre.
Identificateurs de variables et de fonctions, données symboliques.

- Prédicat de type : **symbolp**
- Prédicat d'égalité : **eq**
- Variables spéciales et lexicales : globales, locales, paramètres de fonctions. **defvar**, **defparameter**

Example

```
* (defvar |Je suis un symbole| 1)
|Je suis un symbole|
* |Je suis un symbole|
1
```


Les chaînes de caractères

- Caractère : #\a
- Chaîne : “de caractères”
- Prédicats : **characterp**, **stringp**
- Comparaisons : *char* =, *char*/ =, *char* <, *char* >, *char* <=, *char* >=, *string* =, *string*/ =, *string* <, *string* >, *string* <=, *string* >=.

Les expressions conditionnelles (if)

La forme if

```
(if <condition> <alors> <sinon>)
(if <condition> <alors>)
```

- <condition>, <alors> et <sinon> sont des expressions
- si <condition> vaut vrai alors le résultat est la valeur de l'expression <alors>
- Sinon, le résultat est la valeur de l'expression <sinon>

Example

```
* (if 1 2 3)
2
* (if (= 1 2) 3 4)
4
* (if (= 1 2) 3)
NIL
```


Les expressions conditionnelles (when et unless)

La forme when

```
(when <condition> <e1> ... <en>)
```

Cette forme évalue les expressions <ei> et renvoie le résultat de la dernière quand l'expression <condition> vaut vrai.

La forme unless

```
(unless <condition> <e1> ... <en>)
```

Même chose mais quand l'expression <condition> vaut faux.

Les expressions conditionnelles (cond)

La forme cond

```
(cond <clause> <clause> ... <clause>)
```

- <clause> est une clause : (<condition> <e1> ... <en>)
- <condition> <e1> ... <en> sont des expressions
- Evaluation des conditions des clauses dans l'ordre
- soit ci=(cond e1 ... en) la première clause dont la condition vaut vrai, les ei sont évaluées dans l'ordre et le résultat est celui de en.

Example

```
(cond ((numberp x) ‘‘X est un nombre’’)
((stringp x) ‘‘X est une chaine’’)
((symbolp x) ‘‘X est un symbole’’)
(t (...)))
```


Expressions avec déclarations locales
Déclaration de variables (1)

La forme let

```
(let (l1
      l2
      ...
      ln)
  e)
```

- `li` est une **liaison** : (s_i o_i)
- s_i est un symbole (id. de variable)
- o_i un objet (valeur d'initialisation)
- e est une expression

L'évaluation des valeurs d'initialisation est effectuée en premier (dans l'ordre), puis les variables locales sont créées. Ce qui implique que les valeurs des variales locales définies dans un let ne sont pas utilisées dans l'évaluation des expressions d'initialisation.

Expressions avec déclarations locales
Déclaration de variables (2)

La forme let*

```
(let (l1
      l2
      ...
      ln)
  e)
```

- `li` est une **liaison** : (s_i o_i)
- s_i est un symbole (id. de variable)
- o_i un objet (valeur d'initialisation)
- e est une expression

L'évaluation des expressions d'initialisation est effectuée après la création des variables locales.

Expressions avec déclarations locales
Déclaration de fonctions

La forme flet

```
(flet (fn-1
       fn-2
       ...
       fn-k)
  expr1
  expr2
  ...
  exprn)
```

- `fn-i` est de la forme suivante (nom (p1 p2 ... pn) e1 e2 ... en)
- `nom` : id. de la fonction
- `pi` : paramètre de la fonction
- Les *ei* ne peuvent pas référencer les fonctions locales.

La forme labels *permet la récursivité entre les fonctions et a la même forme syntaxique que flet.*

Les paires pointées

- L'opération **cons** : non associative, non commutative.

Example

```
* (cons (cons (cons 1 2) 3) 4)
(((1 . 2) . 3) . 4)
* (cons 1 (cons 2 (cons 3 4)))
(1 2 3 . 4)
```

- Les sélecteurs **car** et **cdr**

Example

```
* (car (cons 1 2))
1
* (cdr (cons 3 (cons 1 4))
(1 . 4)
```

Attention (car '()) donne (), de même cdr.

- Prédicats **consp** et **atom**
- Abréviations : **cddr**, **caadr**, ..., **nthcdr**

Affichage d'une structure complexe

- (a . (pp)) ⟶ (a pp) si pp est une paire pointée.
- (a . ()) ⟶ (a)

Example

```
* (1 . (2 3))
(1 2 3)
* '(1 2 3)
* (deparameter f '(defun f (x) x))
(defun f (x) x))
* (cons '(1 2) 3)
((1 2) . 3)
```

Les listes

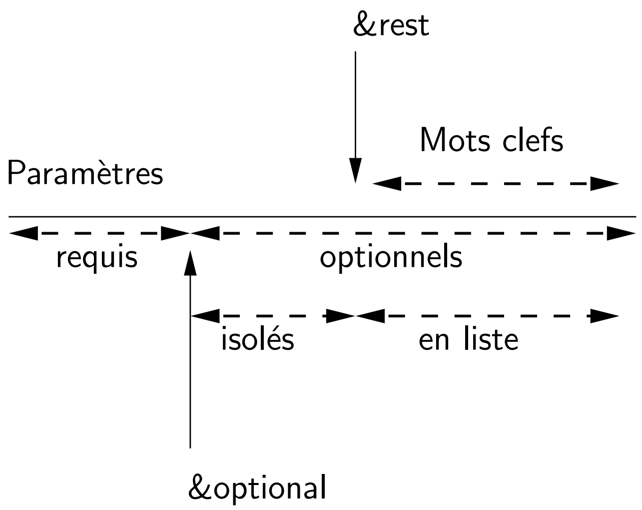
- Prédicats **listp** et **endp** (listes propres)
- Prédicats d'égalité : **eq**, **equal**
- Fonction de Construction : **list** (voir aussi list*)
- Fonctions prédéfinies : **list-length**, **copy-list**, **append**, **reverse**, **subseq**, **first**, ... **tenth**, **nth**, **last**, **but-last**, **member**

Example

```
* (append '(1 . (2 . 3)) '(1))
(1 2 1)
* (append '() '())
```

- Fonctions de a-listes : **assoc** et **acons**

Fonctions (paramètres)



Fonctions (optional)

Example

```
* (defun f(a b &optional c d)
    (list a b c d))
F
* (f 1 2)
(1 2 NIL NIL)
* (f 1 2 3 4)
(1 2 3 4)
* (defun g(a b &optional (c 0) d)
    (list a b c d))
G
* (g 1 2)
(1 2 0 NIL)
```


Fonctions (rest)

Example

```
* (defun g(a b &optional (c 0 c-supplied-p) d)
    (list a b c c-supplied-p d)
G
* (g 1 2)
(1 2 0 NIL NIL)
* (g 1 2 3 4)
(1 2 3 T 4)
```

Example

```
* (defun f(a b &optional (c 0) &rest d)
   (list a b c d))
F
* (f 1 2 3 4 5 6 7)
(1 2 3 (4 5 6 7))
* (f 1 2)
(1 2 0 NIL)
```

Fonctions (mots clefs)

Après le mot `&rest`, des mots-clefs peuvent apparaître dans la définition de la fonction. Ils sont précédés du mot `&key`. Ils peuvent être initialisés comme les paramètres facultatifs. Les mots-clefs non attendus dans une fonction sont ignorés seulement si le mot `&allow-other-keys` apparaît à la fin.

Example

```
* (defun f(a &rest r &key (b 0 b-supplied-p)
&allow-other-keys)
     (list a b b-supplied-p r))
F
* (f 1 :y 2 :c 3)
(1 0 NIL (:Y 2 :C 3))
* (f 1 :c 2 :y 3 :b 0)
(1 0 T (:C 2 :Y 3 :B 0))
```


Fonctions (résultat multiple)

La forme `(values v1 v2 ... vn)` permet d'associer plusieurs valeurs afin de les retourner en résultat d'une fonction.

Example

```
(defun sumdiff(x y)
    (values (+ x y) (- x y)))
```

Par défaut, seule la première valeur est récupérée. Pour accéder aux deux valeurs on utilise la forme `multiple-value-bind`

Example

```
(multiple-value-bind (a b)
                     (sumdiff 3 5)
                     (* a b))
```


Fonctionnelles

- **Function** : valeur fonctionnelle d'une variable. Abréviation **#'**
- **Funcall** : application de la valeur fonctionnelle d'une variable
- Fonctionnelles de listes : **apply**, **mapcar**

Example

```
* (mapcar #'1- '(1 2 3 4 5))
* (mapcar #'cons '(1 2 3 4) '(a b c d))
((1 . A) (2 . B) (3 . C) (4 . D))
(0 1 2 3 4)
```


Structures impératives

- **setf** : affectation
- **nconc** : concaténation
- Fonctionnelles : **mapcan** (mapc + nconc), **mapl**, **maplist** et **mapcon** (mapl + nconc)
- **prog1**, **progn** : blocs
- **case** : conditionnel
- **do** (do*, dotimes, dolist), **loop** : boucles
- Tableaux : **make-array**, **aref**
- Structures : **defstruct**, **make-nom**, **nom-champ1**
- Tables de hachage : **make-hash-table**, **gethash**, **rem-hash**