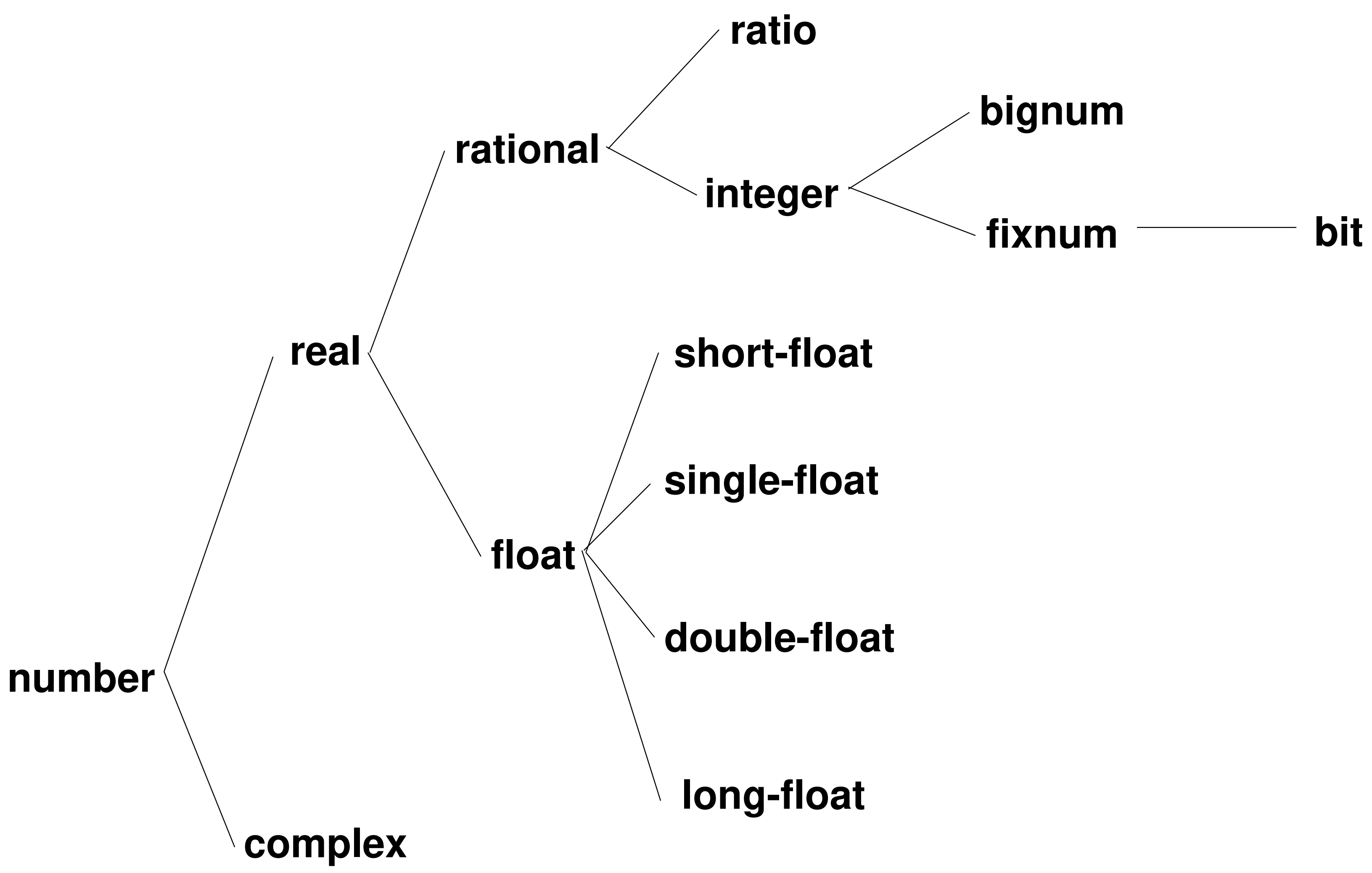
Nombres


ratio
bignum
rational
integer
fixnum
bit
real
short-float
single-float
float
double-float
number
long-float
complex

Nombres: Entiers

Précision arbitraire (bignums)

```
* (defun fact (n)
    (if (= n 0)
        1
        (* n (fact (1- n)))))

FACT
* (fact 5)

120
* (fact 40)

815915283247897734345611269596115894272000000000
*
```

Nombres: Entiers

Opérations logiques sur les entiers: vecteurs infinis de bits ayant soit un nombre fini de "1", soit un nombre fini de "0".

logior, *logxor*, *logand*, *logeqv*, *lognand*, *lognor*, *logandc1* *logandc2*, *logorc1*, *logorc2*, *boole*, *lognot*, *logtest*, *logbitp* *ash*, *logcount*

```
* (logior 14 3)

15
* (ash 20000000000000000000 2)

80000000000000000000
*
```

Nombres: Entiers

Les fixnums sont possible à représenter dans le pointeur

La manipulation de "bytes" d'un nombre de bits arbitraire

Avec les declarations, possible de représenter de façon compacte (par exemple dans les tableaux)

En particulier des tableaux de bits

```
* (fixnump (ash 1 28))

T
* (fixnump (ash 1 29))

NIL
* (make-array '(3 5) :element-type 'bit)

#2A((0 0 0 0 0) (0 0 0 0 0) (0 0 0 0 0))
*
```

Nombres: ratio

Permets de représenter le résultat d'une grande classe de calculs de façon exacte. Exemples:

- La programmation linéaire avec la méthode simplex
- Élasticité de widgets

Constantes : *1/3*, *2/10*, ...

```
* (+ 1/3 4/5)

17/15
* (/ 4/3 2/3)

2
*
```

Nombres: complexes

Constantes : *#c(4 5)*, *#c(1/2 3/4)*, *#c(4/6 2.0)*

Les opérations habituelles marchent *exp*, *expt*, *log*, *sqrt*, *isqrt*, *abs*, *phase*, *signum*, *sin*, *cos* *tan*, *cis*, *asin*, *acos*, *atan*, *sinh*, *cosh*, *tanh*, *asinh*, *acosh*, *atanh*

```
* (sin (/ pi 6))

0.49999999999999994d0
* (sin #c(3 4))

#C(3.8537378 -27.016813)
*
```

Exemples d'utilité : transformée de Fourier

Booléens

Constantes : *nil*, *t*

nil = ()

n'importe quel objet sauf *nil* est considéré équivalent à *t* pour les tests

nil et *t* sont des symboles

la valeur de la variable *nil* est le symbole *nil* la valeur de la variable *t* est le symbole *t*

```
* (if nil 2 3)

3
*
```

Booléens

```
* (if 0 2 3)

2
* (< 2 4 7 123)

T
*
```

Symboles

Constantes : *abc*, *234hello*, |*ceci n'est pas une pipe*|

Sauf avec la syntaxe |...|, le nom est en majuscules

```
* (defvar abc 22)

ABC
* (defvar 234abc 11)

234ABC
* (defvar |ceci n'est pas une pipe| 8)

|ceci n'est pas une pipe|
* (+ |ceci n'est pas une pipe| 3)

11
*
```

Quote

Comment est-ce que la valeur d'un symbole peut-être un autre symbole? (la valeur de *nil* est *nil*, et la valeur de *t* est *t*)

La solution est un opérateur spécial appelé *quote*

Quote

```
* (defvar *a* (quote hello))

*A*
* *a*

HELLO
* (defvar hello (quote |ceci n'est pas une pipe|))

HELLO
* hello

|ceci n'est pas une pipe|
*
```

Au lieu de taper (`quote` *machin*) on peut taper '*machin*.

Symboles

Tester l'égalité entre deux symboles est une opération très rapide.

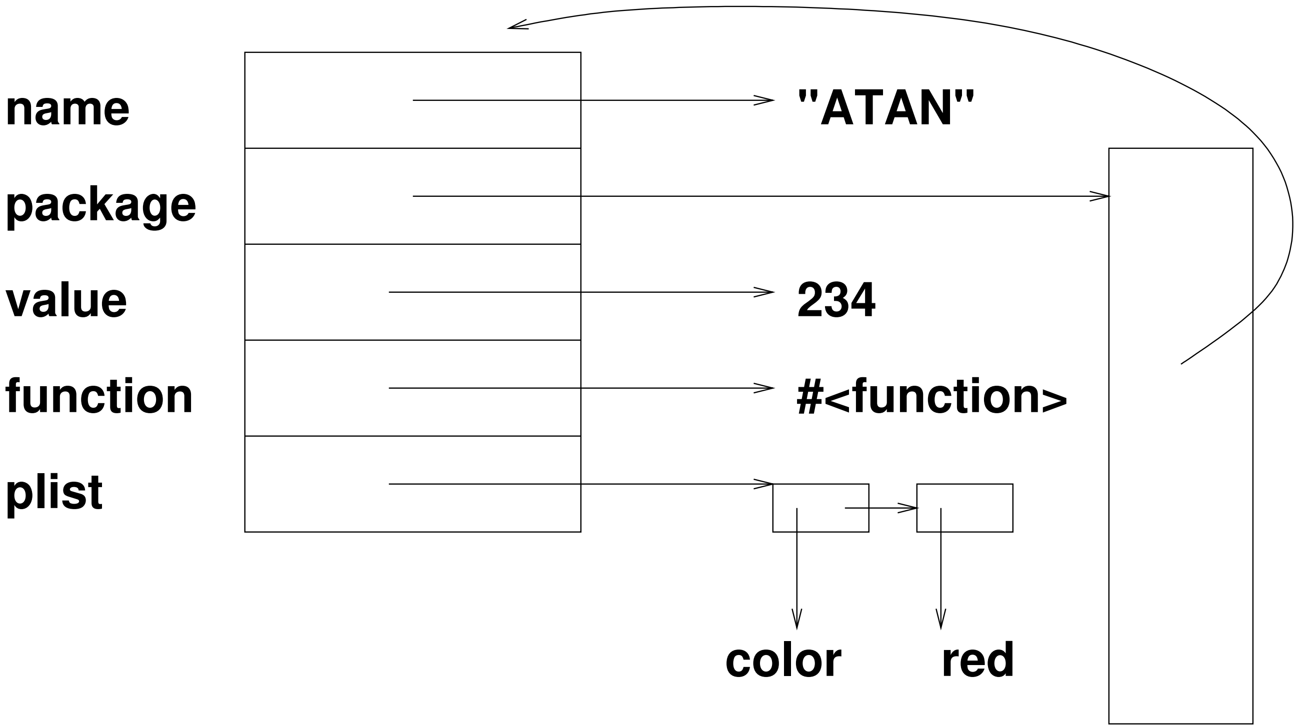
Tableau d'hachage dans le paquetage (package) courant.

C'est la fonction *read* qui cherche le tableau d'hachage et crée le symbole.

```
* (defvar *c* '|ceci n'est pas une pipe|)

*C*
* (eq *c* '|ceci n'est pas une pipe|)

T
* (eq *c* 'hello)

NIL
*
```

Symboles

Représentation:



Symboles: property list (plist)

Une base de données simple Exemple:

```
* (setf (get 'jean 'pere) 'paul)

PAUL
* (setf (get 'paul 'profession) 'plombier)

PLOMBIER
* (get (get 'jean 'pere) 'profession)

PLOMBIER
*
```

Caractères

Constantes : $\#\backslash a$, $\#\backslash b$, $\#\backslash Space$, $\#\backslash Newline$, ...

Fonctions :

alpha-char-p, *upper-case-p*, *lower-case-p*, *both-case-p*, *digit-char-p*, *alphanumericp*, *char=*, *char/=*, *char<*, *char>*, *char<=*, *char>=*, ...

Chaînes de caractères

Constantes: *"abc"*, *"ab\"c"*, ...

Une chaîne de caractères est un vecteur, et un vecteur est un tableaux mais aussi une séquence.

```
* (aref "abc def" 3)

#\Space
* (lower-case-p (aref "abc def" 4))

T
* (char< (aref "abc def" 1) #\a)

NIL
*
```