Spécifications de types

Utilité:

- Vérification de type
- Optimisation de la compilation
- Représentation de données (tableaux de bits, flots d'octets, ...)

Les types simples sont indiqués par des symboles.

Les types simples forment une arborescence (ou plutôt un DAG)

Le type simple le plus général est t (un objet et toujours de type t)

Le type simple le mois général est *nil* (aucun objet n'est de type *nil*)

Types non simples

On peut construire une spécification de type pour n'importe quel ensemble d'objets

Exemple:

```
(or vector (and list (not (satisfies circular?))))
(integer 1 100)
```

Possibilité d'indiquer le type contenu dans un tableau:

```
(simple-array fixnum)
(make-array (list height width) :element-type 'bit)
```

Certains éléments du type peuvent être omis:

```
(simple-array fixnum (* *))
(simple-array fixnum *)
```

Définition de nouveaux noms de types

Utilisation de la macro *deftype* (similaire à *defmacro*)

```
* (deftype proseq ()
    '(or vector (and list (not (satisfies circular?)))))

PROSEQ
* (typep #(1 2) 'proseq)

T
*
```

Définition de nouveaux noms de types

Paramétrage avec des arguments:

```
* (deftype multiple-of (n)
    `(and integer (satisfies (lambda (x)
                                (zerop (mod x ,n))))))

MULTIPLE-OF
* (typep 12 '(multiple-of 4))

T
* (typep 12 '(multiple-of 5))

NIL
*
```

Flots binaires

```
(with-open-file
  (input filename :direction input
                  :element-type 'unsigned-byte)
  ...
  (read-byte input)
  ...
  ...)
```

Read macros

Possibilité de programmer la fonction de lecture de code

Association entre un caractère est une fonction arbitraire

Les caractères ' (quote) et ` (anti quote) sont implémentés en utilisant ce mécanisme

```
(set-macro-character #\'
  #'(lambda (stream char)
      (list (quote quote) (read stream t nil t))))
```

Read macros (suite)

Possibilité de définir des caractères similaires à # (mais pas souvent nécessaire)

Possibilité de rajouter un deuxième caractère à # (ou similaire) avec *set-dispatch-macro-character*

Read macros (suite)

```
* (set-dispatch-macro-character
   #\# #\?
   #'(lambda (stream char1 char2)
       (list 'quote
             (let ((lst nil))
               (dotimes (i (+ (read stream t nil t) 1))
                 (push i lst))
               (nreverse lst)))))

...
* #?7

(0 1 2 3 4 5 6 7)
*
```

caractères réservés au programmeur: #!, #?, #[, #], #{, #}

Read macros (suite)

Création de listes:

```
* (set-macro-character #\} (get-macro-character #\)))

T
*
```

Read macros (suite)

```
* (set-dispatch-macro-character
   #\# #\{
   #'(lambda (stream char1 char2)
       (let ((accum nil)
             (pair (read-delimited-list #\} stream t)))
         (do ((i (car pair) (+ i 1)))
             ((> i (cadr pair))
              (list 'quote (nreverse accum)))
           (push i accum)))))

...
* #2 7

(2 3 4 5 6 7)
*
```

Packages

Pour traduire un nom à un symbole.

Package courant dans **package**

Le package courant est utilisé par *read*

Initialement, le package courant est *common-lisp-user*

Fonctions *package-name* et *find-package*:

```
* (package-name *package*)

"COMMON-LISP-USER"
* (find-package "COMMON-LISP-USER")

#<The COMMON-LISP-USER package, 132/192 internal, 0/9 external>
```

Packages

Fonction symbol-package pour obtenir le package d'un symbole

```
* (symbol-package 'sym)

#<The COMMON-LISP-USER package, 133/192 internal, 0/9 external>
*
```

Packages (suite)

Création et changement de package:

```
* (defparameter sym 99)

SYM
* (setf *package* (make-package 'mine :use '(common-lisp)))

#<The MINE package, 0/9 internal, 0/9 external>
* sym

Error ...

* common-lisp-user::sym

99
*
```

Éviter l'utilisation de ::

Packages (suite)

Utiliser plutôt le concept d'exportation

```
* (in-package common-lisp-user)

#<The COMMON-LISP-USER package, 135/192 internal, 0/9 external>
* (export 'bar)

T
* (defparameter bar 5)

BAR
*
```

Packages (suite)

```
* (in-package mine)

#<The MINE package, 3/9 internal, 0/9 external>
* common-lisp-user:bar

5
* (import 'common-lisp-user:bar)

T
* bar
5
*
```

Packages (suite)

Importation de l'ensemble des symboles exportés:

```
* (use-package 'common-lisp-user)

T
*
```

Loop

```
(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (let* ((wins (car lst))
             (max (funcal fn wins)))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (when (> score max)
              (setf wins obj)
                    max score))))
        (values wins max))))
```

Loop

```
(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (loop with wins = (car lst)
            with max = (funcall fn wins)
            for obj in (cdr lst)
            for score = (funcall fn obj)
            when (> score max)
              do (setf wins obj
                       max score)
            finally (return (values wins max)))))
```