7 Arrays et structures

7.1 Arrays

Des arrays sont des structures de donnée qui permettent de mettre des valeurs en des places spécifiées par des indices.

Un array peut avoir $n \in \mathbb{N}$ dimensions et chaque dimension a une taille m_i, $1 \leq i \leq n$.

Exemple: un array de dimension 1 et de taille $m_1 = 5$:

0	1	2	3	4
un	deux	trois	quatre	cinq

Pour créer un array, on se sert de `make-array`. Cette fonction prend comme argument une liste. La longueur de cette liste est le nombre de dimensions. Le i-ième élément de cette liste est la taille de la dimension i. Les éléments d'un nouvel array sont initialisé avec `nil`.

```
> (make-array '(2))
#(NIL NIL)
> (make-array '(3 4))
#2A((NIL NIL NIL NIL) (NIL NIL NIL NIL) (NIL NIL NIL NIL))
```

`#` signale qu'il s'agit d'un array de dimension 1, `#iA` signale qu'il s'agit d'un array de dimension i.

Si on n'a qu'une dimension, on peut remplacer la liste avec le nombre par le nombre lui-même: `(make-array 2)` est équivalent à `(make-array '(2))`.

En créant un array, on peut donner une valeur initiale pour toutes les positions en utilisant `:initial-element` suivi de la valeur initiale.

```
> (make-array 2 :initial-element 'e)
#(E E)
```

Si la valeur initiale n'est pas la même pour toutes les positions, on peut utiliser `:initial-contents` pour spécifier le contenu initial de l'array:

```
> (make-array 5 :initial-contents '(un deux trois quatre cinq)
#(UN DEUX TROIS QUATRE CINQ)
```

Pour accéder à un élément d'un array, on utilise `aref` avec l'array comme premier argument, le premier indice comme deuxième, le deuxième indice comme troisième etc. Les indices comptent à partir de 0.

```
> (setq array6 (make-array '(2 3) :initial-contents '((a b c) (d e f))))
#2A((A B C) (D E F))
> (aref array6 1 2)
F
```

Dans un appel de `aref` il faut donner des indices pour toutes les dimensions.

Pour mettre une nouvelle valeur à une position dans un array, on utilise `setf` avec la position donnée par un appel de `aref` comme premier argument et la nouvelle valeur comme deuxième argument.

```
> (setf (aref array6 0 1) 'x)
X
> array6
#2A((A X C) (D E F))
```

7.2 Structures

On peut définir des structures de données supplémentaires dans LISP. Ceci peut être utile pour éviter d'utiliser des listes d'une structure complexe pour certains objets.

Une telle structure de donnée consiste en certains champs auxquels on peut attribuer des valeurs default. Une structure est définie avec `defstruct`:

```
> (defstruct automate
   (start 1)
   (final ())
   (transitions ()))
AUTOMATE
```

Syntaxe générale:

```
(defstruct <structure>
  (<champ 1> <default 1>)
  ...
  (<champ n> <default n>))
```

Les valeurs default sont optionnelles, un champ pour lequel on ne spécifie pas de valeur default, reçoit `nil` comme valeur default.

`defstruct` définit une nouvelle structure de donnée (un nouvel type de donnée) mais la fonction ne crée pas encore une instance de cette structure.

`defstruct`

- crée une procedure `make-<nom>` pour créer des structures du nouvel type,
- crée des fonctions pour extraire des valeurs des champs d'une telle structure
- cause une généralisation de `setf` tel que `setf`, combiné avec une procédure d'extraction, peut être utilisé pour écrire dans une structure du nouvel type
- crée un prédicat qui teste si un objet est du type en question.

Quand la fonction `make-<nom>` est appelée, elle crée une nouvelle structure du type `<nom>` avec les valeurs default.

```
> (make-automate)
#S(AUTOMATE :START 1 :FINAL NIL :TRANSITIONS NIL)
```

On peut aussi donner d'autres valeurs aux champs d'une structure au moment de la créer:

```
> (setq french-np
    (make-automate  :final '(3)
                    :transitions '((1 det 2) (2 adj 2) (2 noun 3))))
#S(AUTOMATE :START 1 :FINAL (3) :TRANSITIONS ((1 DET 2) (2 ADJ 2) (2 NOUN 3)))
```

Pour chaque nom de champ dans une structure de donnée, il y a une fonction pour extraire la valeur de ce champ, la fonction `<structure>-<champ>`:

```
> (automate-final french-np)
(3)
```

C'est-à-dire on a accès aux différents champs par des procédures différentes, tandis que dans le cas d'un array on a accès par des indices différents.

Pour modifier la valeur d'un champ on utilise `setf` avec la procédure d'accès correspondante:

```
> (setf (automate-final french-np) '(2 3))
(2 3)
> french-np
#S(AUTOMATE :START 1 :FINAL (2 3) :TRANSITIONS ((1 DET 2) (2 ADJ 2) (2 NOUN 3))
  )
```

Le nom de `setf` (set field) vient de cet usage de la fonction.

Avec la nouvelle structure, on a aussi un prédicat `<nom>-p` qui teste si un objet est une telle structure.

```
> (automate-p french-np)
T
> (automate-p '(1 (3) (1 det 2)))
NIL
```

Il faut faire attention de bien définir tous les champs dont on a besoin. Une structure de donnée une fois définie ne peut plus être modifiée. Seul les valeurs des champs d'une instance de cette structure peuvent être modifiées avec `setf`.