3 Liaisons temporaires et liaisons globales

3.1 Liaisons globales

La fonction `setq` peut être utilisé pour définir une liaison globale: Même si `setq` n'est pas appellé au plus haut niveau mais par exemple à l'intérieur d'une autre fonction, cet appel cause une liaison globale, si le premier argument de `setq` n'est pas une variable locale.

```
> (defun newpair () (setq ab-pair (cons 'a 'b)))
NEWPAIR
> ab-pair
ERROR (ab-pair undefined)
> (newpair)
(A . B)
> ab-pair
(A . B)
```

3.2 Liaisons temporaires

L'appel d'une fonction établit une liaison temporaire: Les paramètres de la fonction sont liés aux valeurs des arguments de l'appel. A la sortie de la fonction, cette liaison n'est plus valable.

```
> (setq this-is-a-list '(this is a list))
(THIS IS A LIST)
> (defun last-element (this-is-a-list)
  (cond ((null this-is-a-list) nil)
        ((= 1 (length this-is-a-list))
         (car this-is-a-list))
        (T (last-element (cdr this-is-a-list)))))
LAST-ELEMENT
> (last-element '(a b c))
C
> this-is-a-list
(THIS IS A LIST)
```

L'environnement pendant l'appel de `last-element`:

environnement top
`this-is-a-list = (THIS IS A LIST)`

environnement last-element
`this-is-a-list = (A B C)`

Parfois on veut établir une liaison temporaire sans définir une nouvelle fonction. Cela peut être fait par `let`.

La syntaxe de `let`:

```
(let ((<parametre 1> <valeur 1>)
      ...
      (<parametre m> <valeur m>))
     <forme 1>
     ...
     <forme n>)
```

let fonctionne comme suit: <valeur 1> à <valeur m> sont évaluées et liées à <parametre 1> à <parametre m> respectivement. Ensuite <forme 1> à <forme n> sont évaluées en utilisant les liaisons établies par let. Une fois l'évaluation de <forme n> est finie, les liaisons ne sont plus valables.

Les liaisons temporaires établies par let sont valables toutes en même moment. C'est-à-dire on ne peut pas se servir de la première liaisons pour définir la valeur du deuxième paramètre.

```
> (setq another-list '(a b c))
(A B C)
> (let ((another-list '(c d e))
        (second-list (cddr another-list)))
       (append another-list second-list))
(C D E C)
> another-list
(A B C)
```

environnement top
another-list = (A B C)

environnement let
another-list = (C D E)
second-list = (C)

Quand setq est appellé avec une variable var comme premier argument, setq cherche var dans l'environnement de l'appel. Si setq y trouve une variable var, celle-ci reçoit une nouvelle valeur. Sinon, setq cherche au niveau plus haut, etc. Si var n'existe pas encore, setq crée une variable var au niveau le plus haut.

```
> (setq another-list '(a b c))
(A B C)
> (let ((another-list '(c d e))
        (second-list (cddr another-list)))
       (setq another-list (append another-list second-list)))
(C D E C)
> another-list
(A B C)
```

Une fonction qui procède comme let sauf que pendant l'évaluation de la valeur d'un paramètre, les liaisons précédentes sont déjà valables est let*:

```
> (setq another-list '(a b c))
(A B C)
> (let* ((another-list '(c d e))
       (second-list (cddr another-list)))
      (append another-list second-list))
(C D E E)
```

3.3 Fonctions locales

let ouvre un environnement local enchâssé:

```
> (setq new-list '(d e f))
```

```
(D E F)
> (let ((new-list '(a b c)))
       (let ((l '(x y z))) (append new-list l)))
(A B C X Y Z)
```

environnement top
`new-list = (D E F)`

environnement let
`new-list = (A B D)`

environnement let
`l = (X Y Z)`

Par contre, si une fonction est définie au plus haut niveau, l'appel de cette fonction à l'intérieur d'une autre fonction ou à l'intérieur de l'appel d'un `let` ouvre un environnement au plus haut niveau:

```
> (defun fct1 (l) (let ((new-list '(a b c))) (fct2 l)))
FCT1
> (defun fct2 (l) (append new-list l))
FCT2
> (setq new-list '(d e f))
(D E F)
> (fct1 '(x y z))
(D E F X Y Z)
```

environnement top
`fct1 = ...`
`fct2 = ...`
`new-list = (D E F)`

environnement fct1
`l = (X Y Z)`

environnement let
`new-list = (A B C)`

environnement fct2
`l = (X Y Z)`

La fonction `labels` sert de définir des fonctions locales (à l'intérieur d'une autre fonction):

```
(defun <nom procedure exterieure> <par>
   (labels (<definitions des procedures interieures>)
           <corps de labels>))
```

Ici les procédures intérieures sont disponibles seulement dans le corps de `labels`.

Dans les définitions des procédures intérieures on ne met pas de “`defun`”.

Exemple: la fonction `np-step` définie dans 1.4 pourrait être définie comme fonction locale de `np-check`:

```
(defun np-check (wordlist)
  (labels ((np-step (words state)          ; definition locale de
               (cond ....)))               ; np-step
          (np-step wordlist 1)))
```

Chaque appel de `np-step` ouvre un environnement directement sous l'environnement créé par `labels`. P.e. pour `(np-check '(le chien))` on a les environnements suivants:

environnement top
`np-check = ...`

environnement np-check
`wordlist = (le chien)`

environnement labels
`np-step = ...`

environnement np-step
`words = (le chien)`
`state = 1`

environnement np-step
`words = (chien)`
`state = 2`

environnement np-step
`words = ()`
`state = 3`

`labels` est par exemple utile quand la fonction principale d'un programme n'est pas récursive mais consiste en l'appel d'une autre fonction avec quelque valeur initiale. Dans ce cas la fonction appellée par la fonction principale peut être définie comme fonction locale.

En plus, les arguments de la fonction principale sont accessibles pour les fonctions locales. C'est-à-dire les paramètres de la fonction globale peuvent être utilisés comme variables libres dans les fonctions locales. Ceci est utile quand un des arguments de la fonction principale reste constant au cours des appels des fonctions locales.

Exemple: la fonction `recognize` qui doit être définie pendant les td appelle une fonction `recognize-step` et à l'intérieur de `recognize-step` on peut se servir d'une fonction `recognize-transition` pour chercher la transition qui doit être choisie dans un certain état et avec un certain mot prochain. Dans `recognize` et `recognize-step` on a besoin de l'automate (qui est un argument de `recognize`). Si `recognize-step` et `recognize-transition` sont définis comme fonctions locales de `recognize`, ce n'est pas nécessaire de passer l'automate comme argument à chacune de ces fonctions (voir aussi solution des td).

```
(defun recognize (fsa wordlist)
   (labels ((recognize-step (wordlist state)
              ... (recognize-transition state
                                        (transitions fsa)
                                        wordlist) ...
            )
            (recognize-transition (state transitions wordlist)
                 ...))
           (recognize-step wordlist (initial-state (fsa)))))
```

Pendant les appels des fonctions locales de `recognize`, la variable `fsa` est comme une variable globale.

3.4 P-listes

Avec `setq` on peut attribuer une valeur à une variable globale.

A part de cela, on peut aussi définir des traits différents d'un symbole (d'une variable) est leurs assigner des valeurs globales. Ces traits s'appellent *properties* et leurs valeurs sont mises sur les listes P des symboles.

Pour extraire la valeur d'un de ces traits, on utilise `get`:

```
(get <symbole> <trait>)
```

Pour assigner des valeurs, on utilise `setf` et dans l'appel de `setf` le premier argument est le trait auquel on veut assigner une valeur. Ceci est extrait par `get`:

```
(setf (get <symbole> <trait>) <valeur>)
```

Un trait peut être enlevé avec `remprop`:

```
(remprop <symbole> <trait>)
```

Exemple:

	cat	num	gen
chien:	n	sg	m
joli:	a	sg	m

```
> (setf (get 'chien 'cat) 'n)
N
> (setf (get 'chien 'num) 'sg)
SG
> (setf (get 'chien 'gen) 'm)
M
> (setf (get 'joli 'cat) 'a)
A
> (setf (get 'joli 'num) 'sg)
SG
> (setf (get 'joli 'gen) 'm)
M
> (defun accord (w1 w2)
     (let ((n1 (get w1 'num)) (n2 (get w2 'num))
           (g1 (get w1 'gen)) (g2 (get w2 'gen)))
          (and n1 g1
               (eq n1 n2)
               (eq g1 g2))))
ACCORD
> (accord 'chien 'joli)
T
```

Remarque: Quand on utilise des structures de traits dans un programme ou on ne veut pas lier chaque structure à un symbole, il vaut mieux ne pas utiliser les listes P mais par exemple définir des structures de traits comme des listes de la forme suivante:

```
(<trait 1> <valeur 1>
 ...
 <trait n> <valeur n>)
```

On peut définir une fonction `putprop` qui assigne une valeur à un trait sur la liste P d'un symbole:

```
(defun putprop (symbol value property)
       (setf (get symbol property) value))
```

Avec cette définition, on peut écrire

```
(putprop 'chien 'sg 'num)
```

au lieu de

```
(setf (get 'chien 'num) 'sg)
```

La fonction `putprop` existe dans quelques dialectes LISP.

Quand `setf` est utilisé avec deux arguments, la fonction procède comme `setq`:

```
> (setf une-liste '(encore une liste))
(ENCORE UNE LISTE)
> une-liste
(ENCORE UNE LISTE)
```

Comme `setq`, `setf` peut être utilisé pour créer une nouvelle valeur ou pour modifier une valeur.