2 Les fonctions de base

2.1 Les affectations avec `SETQ`

L'appel d'une fonction avec certains arguments donne des valeurs aux paramètres de la fonction. Il s'agit d'une liaison temporaire. Après la sortie de la fonction la liaison n'existe plus.

Pour donner des valeurs à des variables en établant une liaison globale, on peut utiliser `setq`. `setq` a deux arguments: le nom d'une variable et une s-expression. L'effet de l'évaluation de `setq` est de donner à la variable comme nouvelle valeur le résultat de l'évaluation de la s-expression. `setq` retourne la nouvelle valeur de la variable.

```
(setq syntagme-nominal '(le joli grand chien))
                                         →  (LE JOLI GRAND CHIEN)
syntagme-nominal                         →  (LE JOLI GRAND CHIEN)
```

Après avoir établi cette liaison, on peut appeller la fonction `np-check` définie en 2.4 avec comme argument la variable `syntagme-nominal`:

```
(np-check syntagme-nominal)              →  T
```

Au lieu de définir une fonction comme `np-check`, il vaut mieux définir une fonction plus générale qui prend un automate et une liste comme arguments et qui teste si l'automate accepte la séquence representée par la liste.

Un automate consiste d'un état initial, d'un ensemble d'états finaux et d'un ensemble de transitions. Chaque transition contient un noeud ancien, un noeud nouveau et un symbole.

Une structure de donnée possible pour représenter un automate avec n transitions en LISP:

```
(<etat initial>
 (<etats finaux>)
 (<noeud ancien 1> <symbole 1> <noeud nouveau 1>)
 ...
 (<noeud ancien n> <symbole n> <noeud nouveau n>))
```

Maintenant on peut formuler l'automate de 2.1 de cette manière et lui donner un nom:

```
(setq french-np
      '(1 (3)
        (1 un 2)
        (1 le 2)
        (2 grand 2)
        (2 joli 2)
        (2 chat 3)
        (2 chien 3)))
```

2.2 Manipulation des listes

Nous connaissons déjà `car`, `cdr`, `first`, `rest`, et `cons`.

Deux autres fonctions pour construire des listes, à part de `cons`, sont `append` et `liste`. `append` prend des listes comme arguments et construit une nouvelle liste contenante les éléments de ses arguments. `append` ne détruit pas ses arguments.

```
(append '(a b) '(c) '(d))                →  (A B C D)
(setq first-list '(a b c))               →  (A B C)
(append first-list '(a (b c)))           →  (A B C (A (B C)))
first-list                               →  (A B C)
```

La fonction `list` prend des s-expressions comme arguments et construit une liste contenante ces s-expressions. Comme `append`, `list` ne détruit pas ses arguments.

```
(list '(a b) 'c 'd)          →  ((A B) C D)
(list 'a ())                 →  (A ())
```

`cons` ne détruit pas ses arguments non plus.

Pour extraire une partie d'une liste, on peut utiliser `car` et `cdr`.

Des fonctions pour extraire l'état initial, la liste des états finaux et les transition d'un automate:

```
(defun initial-state (fsa) (car fsa))

(defun final-states (fsa) (cadr fsa))

(defun transitions (fsa) (cddr fsa))
```

Une fonction pour chercher un élément dans une liste à partir d'une certaine clé est `assoc`. Elle prend comme argument un atome et une liste qui associe des valeurs à des clés. Si le premier argument de `assoc` figure comme clé dans la liste, `assoc` rend cette clé avec sa valeur. Sinon `assoc` rend `NIL`.

```
(assoc 'det '((det (un le)) (n (chien chat))))   ::> (DET (UN LE))
(assoc 'det '((det un le) (n chien chat)))       ::> (DET UN LE)
(assoc 'det '((art (un le)) (n (chien chat))))   ::> NIL
```

Deux autres fonctions pour manipuler des listes sont `length` et `reverse`. `length` prend une liste et retourne le nombre d'éléments de cette liste:

```
(length '(a b c))                                 ::> 3
(length '(((a b c (d)) e)))                       ::> 1
(length '())                                      ::> 0
```

La fonction `reverse` prend une liste comme argument et retourne une liste qui contient les mêmes éléments en ordre inverse.

```
(reverse '(a b c))                                ::> (C B A)
(reverse '(((a b c (d)) e)))                      ::> (((A B C (D)) E))
(reverse '())                                     ::> NIL
```

`reverse` ne détruit pas son argument.

2.3 Prédicats et structures de contrôle

`eq` test l'égalité de deux noms, `=` teste l'égalité de deux nombres, et `eql` teste les deux.

Un prédicat pour tester l'égalité de deux s-expressions en general est `equal`:

```
> (equal (cdr '(a b c)) '(b c))
T
```

`equal` est plus couteux que `eql` ou `eq`.

La fonction `member` prend un atome et une liste et elle teste si l'atome est un élément de la liste. Dans ce cas elle rend la sous-liste contenante tous les èlèments à partir de cet atome (de sa première occurrence). Sinon elle retourne `NIL`.

```
(member 'a '(premier-element a c))          ::> (A C)
(member 'a '(a a a))                        ::> (A A A)
(member 'd '(a b c))                        ::> NIL
(member '(a) '((a) (b)))                    ::> NIL
```

Pour tester si un objet est un atome, on se sert du prédicat `atom` tandis que `listp` teste si un objet est un doublet, en particulier s'il est une liste.

`NIL` est la liste vide sont équivalents en LISP est sont en même temps atome est liste:

```
> (atom '())
T
> (listp NIL)
T
```

Le prédicat `numberp` teste si un objet est un nombre et `symbolp` teste si un objet est un nom.

D'autres structures de contrôle que `cond` sont `if` and `case`.

`if` a trois arguments (les parties "if", "then" et "else"). C'est-à-dire,

```
(if <teste> <action1> <action2>)
```

est équivalent à

```
(cond (<teste> <action1>)
      (T <action2>))
```

La fonction `case` a la syntaxe suivante:

```
(case <teste forme>
  (<valeur 1> <action 11> ...)
    ...
  (<valeur n> <action n1> ...))
```

Ceci est équivalent à l'appel de `cond` suivant:

```
(cond ((eql <teste forme> '<valeur 1>)
       <action 11> ...)
      ...
      ((eql <teste form> '<valeur n>)
       <action n1> ...))
```

Dans le `case`, `<teste forme>` est évalué tandis que `<valeur 1>`, ..., `<valeur n>` ne sont pas évalués.

`case` peut remplacer un `cond` dans lequel tous les testes teste un objet pour une certaine valeur.