6 Input et output, strings

6.1 PRINT, PRINC, FORMAT et READ

La fonction `print` prend un seul argument, l'évalue et l'imprime sur une nouvelle ligne.

```
> (defun out-n (n)
    (dotimes (count n 'finished)
       (print (+ 1 count))))
OUT-N
> (out-n 5)

1
2
3
4
5
FINISHED
```

`princ`, par contre, ne commence pas de nouvelle ligne:

```
>(dotimes (count 3 'finished) (princ count))
012
FINISHED
```

Avec `princ`, on peut aussi imprimer des strings. Un string est une séquence de caractères entre guillemets : `"this is a string"`. La valeur d'un appel de `princ` est l'objet imprimé:

```
>(princ 'a)
A
A
>(princ "a")
a
"a"
```

Une autre fonction pour imprimer des strings est `format`. La valeur de retour de `format` est `nil`. Pour imprimer sur le terminal avec `format`, il faut utiliser un paramètre `t`:

```
>(format t "Bonjour!")
Bonjour!
NIL
>(format t "Bonjour!~%Tu vas bien?")
Bonjour!
Tu vas bien?
NIL
```

"~%" produit un retour charriot. "~&" produit un retour charriot seulement si on ne se trouve pas encore au début d'une nouvelle ligne.

Pour lire une valeur on utilise `read`. La fonction `read` est appelée sans arguments et après l'appel le système attend jusqu'à ce que l'utilisateur a entré une valeur (une s-expression qui n'est pas évaluée).

```
>(setq mot (read))
bonjour
BONJOUR
> mot
BONJOUR
```

6.2 Compiler et charger des fichiers

On peut évaluer des programmes directement dans l'interpréteur LISP, mais on peut aussi les compiler d'abord et les charger ensuite. Pour tester des petites fonctions il suffit de les charger directement (sans compilation), mais en général il vaut mieux compiler et charger après.

La compilation d'un programme se fait par `compile-file` avec un string (le nom du fichier) comme argument. Cela crée un fichier binaire (`<nom>.o` pour gcl) qui peut être chargé avec `load` après. Dans gcl, quand on omet l'extension du fichier dans un appel de `load` et quand le fichier binaire existe, celui-ci est chargé.

```
>(compile-file "fsa.lisp")
... (quelques messages de compilation)...
>(load "fsa")
... (message que fsa.o est charge)...
T
```

On peut aussi combiner des appels de `load` et `compile-file`. Par exemple, après avoir compilé et chargé un programme, on peut charger d'autres fonctions supplémentaires sans les compiler.

6.3 Lire et écrire dans des fichiers

Pour lire des expressions d'un fichier ou pour écrire quelque chose dans un fichier, on utilise `with-open-file`. Cette fonction crée un *stream*, pour lire d'un fichier un *input stream* et pour écrire dans un fichier un *output stream*. Un input stream connecte un fichier à une fonction `read` tandis qu'un output stream connecte une fonction `print` ou `format` à un fichier.

Syntaxe de `with-open-file`:

```
(with-open-file (<stream name>
                 <file specification>
                 :direction <:input ou :output>)
                <corps>)
```

`with-open-file` ouvre le fichier dont le nom est donné par `<file specification>`, crée un stream et donne le nom `<stream name>` à ce stream. Dans le corps de `with-open-file`, ce stream peut être utilisé

- comme argument de `read` si la direction est `:input`
- comme argument de `print` ou `format` si la direction est `:output`

Exemple:

Fichier `test-data` avec des listes de mots et le résultat expecté pour tester la fonctin `recognize` pour un automate:

Contenu de `test-data`:

```
(le chien)
T
```

```
(le grand chien)
T
(le chien chien)
nil
(le joli grand chat)
T
END
```

Tester la fonction `recognize` avec ces données:

```
> (with-open-file (test-data "test-data" :direction :input)
   (let (new-expr (read test-data))
     (loop
       (setq new-expr (read test-data))
       (if (eq 'END new-expr) (return 'end))
       (format t "~&Nouvelle phrase: ")
       (print new-expr)
       (format t "~&Resultat: ")
       (print (recognize french-np new-expr))
       (setq new-expr (read test-data))
       (format t "~&Resultat expecte: ")
       (print new-expr))))
Nouvelle phrase:
(LE CHIEN)
Resultat:
T
Resultat expecte:
T
Nouvelle phrase:
(LE GRAND CHIEN)
Resultat:
T
Resultat expecte:
T
Nouvelle phrase:
(LE CHIEN CHIEN)
Resultat:
NIL
Resultat expecte:
NIL
Nouvelle phrase:
(LE JOLI GRAND CHAT)
Resultat:
T
Resultat expecte:
T
END
```

Pour faire le même teste mais l'écrire dans un nouveau fichier `"test-results"` (sauf le `END` à la fin), on peut appeler `with-open-file` comme suit:

```
(with-open-file (test-data "test-data" :direction :input)
  (with-open-file (results "test-results" :direction :output)
   (let (new-expr (read test-data))
```

```
    (loop
      (setq new-expr (read test-data))
      (if (eq 'END new-expr) (return 'end))
      (format results "~&Nouvelle phrase: ")
      (print new-expr results)
      (format results "~&Resultat: ")
      (print (recognize french-np new-expr) results)
      (setq new-expr (read test-data))
      (format results "~&Resultat expecte: ")
      (print new-expr results)))))
```

La déstination dans `format` (le premier argument) peut être `t` pour terminal ou le nom d'un stream.

6.4 Manipulation des strings et caractères

Comme une liste, un string consiste d'un ensemble d'éléments qui se trouvent dans un certain ordre. Toutes les deux, listes et strings sont des séquences et beaucoup de fonctions s'appliquent à les deux:

```
> (length "abc")
3
> (reverse "abc")
"cba"
```

Une fonction pour extraire le n-ième élément (on compte à partir de 0) est `elt`:

```
> (elt '(a b c) 0)
a
> (elt "abc" 0)
#\a
```

`#\a` est un caractère.

Une autre fonction qu'on peut appliquer à des listes et à des strings est `search`. Cette fonction teste si une liste (string) est sous-liste (sous-string) d'une deuxième liste (d'un deuxième string). Si oui, `search` retourne la position du début de la sous-liste (du sous-string) en comptant à partir de 0. Sinon, la fonction rend `nil`.

```
> (search "chien" "le grand chien")
9
> (search '(grand chien) '(le grand chien))
1
```

Pour comparer deux strings, il y a les fonction `string=` (qui fait une différence entre minuscules et majuscules) et `string-equal` (qui ne distingue pas entre minuscule et majuscule).

```
> (string= "abc" "xyz")
nil
> (string= "abc" "abc")
T
> (string= "abc" "ABC")
nil
> (string-equal "abc" "ABC")
T
```

Pour la comparaison de deux caractères, on a `char=` et `char-equal` qui se distinguent de la même façon que `string=` et `string-equal`.

```
> (char= #\a #\b)
nil
> (char= #\a #\a)
T
> (char= #\a #\A)
nil
> (char-equal #\a #\A)
T
```

Au lieu de `read` pour lire une expression, on peut lire avec `read-line` une suite de caractères jusqu'à un retour charriot ou jusqu'à la fin d'un fichier, et avec `read-char` on peut lire un seul caractère.

`read-line` retourne deux valeurs, le nouveau string et `nil`, `read-char` retourne le nouveau caractère.

```
> (read-line)
une nouvelle ligne
"une nouvelle ligne" ;
NIL
> (setq nl (read-line))
une nouvelle ligne
"une nouvelle ligne"
> nl
"une nouvelle ligne"
> (read-char)
x
#\x
```