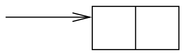
Listes

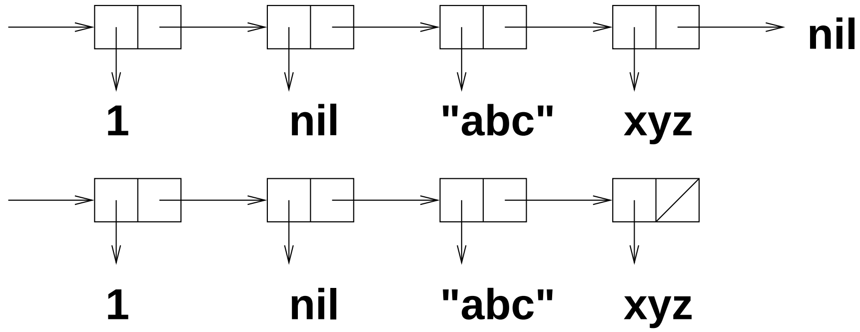
Une list est une séquence

Le type de base est la paire ou le *cons*



Listes

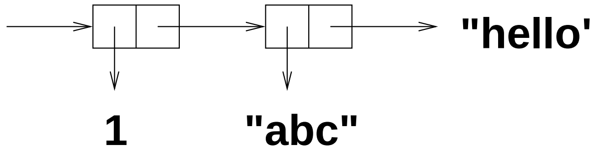
Une liste (anglais: proper list) est une suite de paires terminée par *nil*.



Affichage par *print* : `(1 NIL "abc" XYZ)`

Listes

Une liste généralisée (anglais: dotted list) est terminée par un objet autre que *nil*.



Affichage par *print* : `(1 "abc" . "hello")`

Listes (suite)

Une liste tapée à la boucle REPL est considérée comme une expression composée, et sera donc évaluée. Pour obtenir une liste sans l'évaluer, utiliser *quote*.

Listes (suite)

```
* (+ 3 4)

7
* '(+ 3 4)

(+ 3 4)
* (defvar *my-list* '(+ 3 4))

*MY-LIST*
* *my-list*

(+ 3 4)
*
```

Listes (suite)

Operations de base: *cons*, *car*, *cdr*

```
* (cons 1 (cons 2 (cons 3 nil)))

(1 2 3)
* (cons 'hello '(how are you))

(HELLO HOW ARE YOU)
*
```

Listes (suite)

```
* (setf *my-list* '(how are you))

(HOW ARE YOU)
* (cons 'hello *my-list*)

(HELLO HOW ARE YOU)
* *my-list*

(HOW ARE YOU)
* (car *my-list*)

HOW
* (cdr *my-list*)

(ARE YOU)
*
```

Listes (suite)

Opérations plus complexes : *list*, *append*, *reverse*, ...

```
* (setf *a* 'hello)

HELLO
* (setf *my-list* '(*a* 3 *standard-output*))

(*A* 3 *STANDARD-OUTPUT*)
* (setf *my-list* (list *a* 3 *standard-output*))

(HELLO 3 #<Synonym Stream to SYSTEM:*STDOUT*>)
* (append *my-list* '(a b c))

(HELLO 3 #<Synonym Stream to SYSTEM:*STDOUT*> A B C)
* *my-list*
```

Listes (suite)

```
(HELLO 3 #<Synonym Stream to SYSTEM:*STDOUT*>)
* (reverse *my-list*)

(#<Synonym Stream to SYSTEM:*STDOUT*> 3 HELLO)
* *my-list*

(HELLO 3 #<Synonym Stream to SYSTEM:*STDOUT*>)
*
```

Listes (suite)

Structure de contrôle de base: la récursivité

Utiliser *endp* pour terminer la récursion

```
* (defun greater-than (l x)
    (if (endp l)
        '()
        (if (> (car l) x)
            (cons (car l) (greater-than (cdr l) x))
            (greater-than (cdr l) x))))

GREATER-THAN
* (greater-than '(5 3 6 4 5 3 7) 4)

(5 6 5 7)
*
```

Listes (suite)

Mais on n'a presque jamais besoin de récursion sur les listes.

```
* (remove-if-not (lambda (x) (> x 4)) '(5 3 6 4 5 3 7))

(5 6 5 7)
*
```

Listes (suite)

Attention:

```
* (car nil)

NIL
* (cdr nil)

NIL
*
```

Atomes et listes

```
(defun my-atom (x)
  (not (consp x)))

(defun my-listp (x)
  (or (null x) (consp x)))
```

Égalité

```
* (defparameter *x* (list 'a))

*X*
* (eq *x* *x*)

T
* (eql *x* *x*)

T
* (equal *x* *x*)

T
*
```

Égalité

```
* (eq *x* (list 'a))

NIL
* (eql *x* (list 'a))

NIL
* (equal *x* (list 'a))

T
*
```

Listes (suite)

Construction: *list**, *append*, *copy-list*, *copy-alist*, *copy-tree*, *revappend*, *nconc*, *nreconc*, *push*, *pushnew*, *pop*, *butlast*, *nbutlast*, *ldiff*, *rplaca*, *rplacd*, *subst*, *subst-if*, *subst-if-not*, *nsubst*, *nsubst-if*, *nsubst-if-not*, *sublis*, *member*, *tailp*, *adjoin*,*union*, *intersection*, *set-difference*, *nset-difference*, *set-exclusive-or*, *nset-exclusive-or*, *subsetp*, *acons*, *pairlis*, *assoc*, *assoc-if*, *assoc-if-not*, *find*, *rassoc*, *rassoc-if*, *rassoc-if-not*

Listes (suite)

Accès: *cons*, *list car*, *cdr*, *nth*, *nthcdr*, *last*, *elt*, *cadadr*, *cdaaar*, ... *first*, *second*, ..., *tenth*

Autres: *length*, *subseq*, *copy-seq*, *fill*, *replace*, *count*, *reverse*, *nreverse*, *concatenate*, *position*, *find*, *sort*, *merge*, *map*, *some*, *every*, *notany*, *notevery*, *reduce*, *search*, *remove*, *remove-dupicates*, *delete*, *delete-duplicates*, *substitute*, *nsubstitute*, *mismatch*