# Structures de données linéaires

## Licence 1 MASS - Algorithmique

Sébastien Verel
verel@i3s.unice.fr
www.i3s.unice.fr/∼verel

Équipe ScoBi - Université de Nice Sophia-Antipolis

28 mars 2008

# Objectifs de la séance 10

1. Connaître les opérations de base sur les listes
2. Calculer la longueur d'une liste
3. Extraction de sous-listes d'une liste
4. Utiliser un accumulateur avec une liste

**Questions principales du jour :**

Comment traiter les informations les unes après les autres ?

# Plan

1. Introduction
2. Structure de données liste
   - Définition
   - Liste en maple
3. Algorithmes sur listes

# Exemple
## Liste de tâches

Critique du chapitre
Acheter une salade
Acheter le journal
Ecrire à Michael
Dormir
Répondre à Nicolas
S'occuper de l'annuaire
Donner à manger aux poissons
Rencontrer Andrea
Préparer le conseil de labo
Mettre à jour le site web
Photocopier les sujets POO

# Comment traiter ces données ?

- Faire les tâches les unes après les autres :
  pas toutes en même temps...
  $\longrightarrow$ Traitement séquentiel
- Laquelle ?
  pas de préfèrence (ou priorité cf TP)
  $\longrightarrow$ Lire la première tâche
- Construire la liste de tâches :
  pouvoir ajouter une nouvelle tâche à une liste de tâches, où ?
  $\longrightarrow$ pas de préférence : en tête de la liste
- Retirer une tâche déjà effectuée :
  pour diminuer le nombre de tâches à faire, laquelle ?
  $\longrightarrow$ celle qui vient d'être lue
- Savoir s'il reste une tâche à effectuer

# Algorithme informel de traitement de tâches

**Algorithme** traitement(*l* : liste de taches) : rien
**début**
  **si** *l* est vide **alors**
    ne rien faire
  **sinon**
    exécuter( première tâche de *l* )
    traitement( reste de la liste *l* )
  **fin si**
**fin**

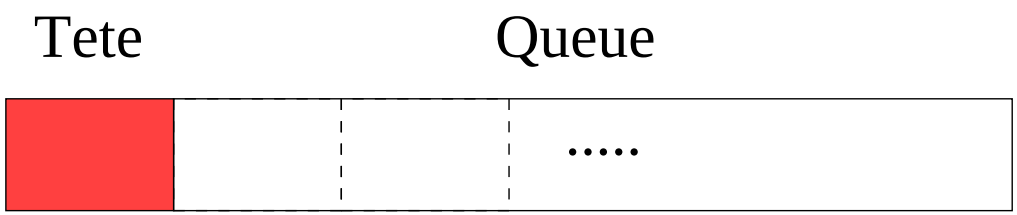
Remarques :

- algorithme récursif ?
- terminaison si la liste est de taille finie

# Définition

## Liste

Suite finie de données (appelée aussi éléments) où il est seulement possible d'ajouter et de lire une donnée au début de la suite (tête de la liste).



# Représentation et notation

liste "usuelle" de tâches : notation verticale
notation des listes (informatique) : de gauche à droite.

Une liste de nombres entiers :

$$(12, 34, 3, 45, 7, 90)$$

Une liste de mots :

$$('farine', 'oeuf', 'beurre', 'sucre', 'mensonge', 'citron')$$

- 12 et $'farine'$ sont en tête des listes.
- $(34, 3, 45, 7, 90)$ et $('oeuf', 'beurre', 'sucre', 'mensonge', 'citron')$ sont les queues des listes

# Primitives

5 fonctions primitives sont nécessaires pour définir les listes :

- listeVide : *rien* $\longrightarrow$ *liste*
  retourne la liste vide
- listeCons : *typeElement* $\times$ *liste* $\longrightarrow$ *liste*
  ajoute un élément de type *typeElement* en tête de la liste
- listeTete : *liste* $\longrightarrow$ *typeElement*
  retourne l'élément en tête de la liste
- listeQueue : *liste* $\longrightarrow$ *liste*
  retourne la queue de la liste
- listeEstVide ? : *liste* $\longrightarrow$ *booleen*
  retourne la valeur Vrai ssi la liste est vide

# Exemple de listes

- listeVide() retourne la liste vide :
  On note () la liste vide
- listeEstVide ?( listeVide() ) :

  Vrai
- listeEstVide ?( $(34, 3, 45, 7, 90)$ ) :

  Faux
- listeCons( 12, $(34, 3, 45, 7, 90)$ ) :

  $(12, 34, 3, 45, 7, 90)$
- listeCons( $'farine'$ ,
  $('oeuf', 'beurre', 'sucre', 'mensonge', 'citron')$ ) :

  $('farine', 'oeuf', 'beurre', 'sucre', 'mensonge', 'citron')$

# Exemples

- listeTete((12, 34, 3, 45, 7, 90))

  12

- listeTete(('farine',' oeuf',' beurre',' sucre',' mensonge',' citron'))

  'farine'

- listeQueue((12, 34, 3, 45, 7, 90))

  (34, 3, 45, 7, 90)

- listeQueue(
  ('farine',' oeuf',' beurre',' sucre',' mensonge',' citron') )

  ('oeuf',' beurre',' sucre',' mensonge',' citron')

# Exemples de construction de listes

- listeCons( 7, listeVide() ) :

$$(7)$$

- listeCons( 34, listeCons( 3 , listeCons( 45 , listeCons( 7, listeCons(90, listeVide() ))))) :

$$(34, 3, 45, 7, 90)$$

- ('*oeuf*','*beurre*','*sucre*','*mensonge*') :
  listeCons( '*oeuf*', listeCons( '*beurre*' , listeCons( '*sucre*' , listeCons( '*mensonge*', listeVide() ))))

# Quand utiliser une liste ?

- A-t-on besoin d'un traitement séquentiel des données ?
- Le nombre de données est-il déterminé à l'avance ?
- Le nombre maximal de données est-il déterminé à l'avance ?
- Les données ont-elles besoin d'être indexées (numérotées) ?
- A-t-on besoin d'accéder à une donnée dont l'indexe doit être déterminé ?
- A-t-on besoin d'accéder à la première donnée ?

si oui, non, non, non, non, oui alors il est conseillé d'utiliser une liste.

# Création d'une liste

Les listes sont des structures de base du langage maple.

Il existe beaucoup de fonctions adaptées aux listes.

La notion de liste est très souple en maple :
on peut par exemple accèder un élément d'indice i d'une liste maple.

Mais alors, ce n'est plus une liste au sens algorithmique...

Dans ce cours, nous restreindrons à utiliser seulement les cinqs primitives sur les listes que nous allons définir.

# Constructeurs

- constructeur de liste vide :

```
listeVide := proc()
# sortie : liste vide

  RETURN( [] );
end;
```

- constructeur ajoutant un élément en tête :

```
listeCons := proc(e, l)
# e : élément à ajouter
# l : liste à complèter
# sortie : liste

  RETURN( [e, op(l)] );
end;
```

op : valeur de l'opérande de l'argument

# Accesseurs

- tete de la liste :

```
listeTete := proc(l)
# l : liste
# sortie : élément, premier élément de la liste

  RETURN( l[1] );
end;
```

- queue de la liste :

```
listeQueue := proc(l)
# l : liste
# sortie : liste contenant la queue

  RETURN( l[2..nops(l)] );
end;
```

nops : nombre d'opérandes

# Test de liste vide

- Test de la liste vide :

```
listeEstVide? := proc(l)
# l : liste
# sortie : booléen, true ssi l est la liste vide

  RETURN( evalb(l = []) );
end;
```

# Calcul de la factorielle

```
Algorithme  factorielle(n : entier) : entier
début
   si n = 0 alors
      retourner 1
   sinon
      retourner  n * factorielle(n-1)
   fin si
fin
```

# Algorithme récursif sur liste

## Traitement de tâches

**Algorithme** traitement(*l* : liste de taches) : rien
**début**
  **si** *l* est vide **alors**
    ne rien faire
  **sinon**
    exécuter( première tâche de *l* )
    traitement( reste de la liste *l* )
  **fin si**
**fin**

# Algorithme récursif sur liste

## Traitement de tâches

**Algorithme** traitement(*l* : liste) : *rien*
**début**
  **si** listeEstVide ?(*l*) **alors**
    exécuter( '*findetache*' )
  **sinon**
    exécuter( listeTete( *l* ) )
    traitement( listeQueue( *l* ) )
  **fin si**
**fin**

En supposant que notre machine comprenne "*exécuter*"

# Squelette d'un algorithme récursif sur liste

**Algorithme** traitement(*l* : liste) : *type*
**début**
  **si** listeEstVide ?(*l*) **alors**
    **retourner** ....
  **sinon**
    ....
    **retourner** ... traitement( ... listeQueue( *l* ) ...)
  **fin si**
**fin**

# Calcul de la longueur d'une liste

longueur( $(34, 7, 42, 121)$ )

4

**Algorithme** longueur($l$ : liste) : entier
**début**
  **si** listeEstVide ?($l$) **alors**
    **retourner** 0
  **sinon**
    **retourner** 1 + longueur( listeQueue( $l$ ) )
  **fin si**
**fin**

# Exécution de l'algorithme

Calcul de longueur( $(34, 7, 42, 121)$ ).

1. longueur$((34, 7, 42, 121)) = 1 +$ longueur$((7, 42, 121))$
2. ——> longueur$((7, 42, 121)) = 1 +$ longueur$((42, 121))$
3. ————-> longueur$((42, 121)) = 1 +$ longueur$((121))$
4. ——————-> longueur$((121)) = 1 +$ longueur$(())$
5. ————————-> longueur$(()) = 0$
6. ——————-> longueur$((121)) = 1 + 0 = 1$
7. ————-> longueur$((42, 121)) = 1 + 1 = 2$
8. ——> longueur$((7, 42, 121)) = 1 + 2 = 3$
9. longueur$((34, 7, 42, 121)) = 1 + 3 = 4$

# Longueur en maple

```
longueur := proc(l)
# l : liste
# sortie : entier, taille de la liste

  if listeEstVide?(l) then
    RETURN( 0 );
  else
    RETURN( 1 + longueur(listeQueue(l)) );
  fi;

end;
```

# Somme des éléments

somme( $(34, 7, 42, 121)$ )

$$204$$

**Algorithme** somme($l$ : liste) : : entier
**début**
  **si** listeEstVide ?($l$) **alors**
    **retourner** 0
  **sinon**
    **retourner** listeTete($l$) + somme( listeQueue( $l$ ) )
  **fin si**
**fin**

# Exécution de l'algorithme

Calcul de somme( $(34, 7, 42, 121)$ ).

1. somme$((34, 7, 42, 121)) = 34 +$ somme$((7, 42, 121))$
2. ——> somme$((7, 42, 121)) = 7 +$ somme$((42, 121))$
3. ————-> somme$((42, 121)) = 42 +$ somme$((121))$
4. ——————-> somme$((121)) = 121 +$ somme$(())$
5. ————————-> somme$(()) = 0$
6. ——————-> somme$((121)) = 121 + 0 = 121$
7. ————-> somme$((42, 121)) = 42 + 121 = 143$
8. ——> somme$((7, 42, 121)) = 7 + 143 = 150$
9. somme$((34, 7, 42, 121)) = 34 + 150 = 184$

# Somme en maple

```
somme := proc(l)
# l : liste
# sortie : réel, somme des éléments de la liste

  if listeEstVide?(l) then
    RETURN( 0 );
  else
    RETURN( listeTete(l) + somme(listeQueue(l)) );
  fi;

end;
```

# Extraction de sous-liste

## Extraire la sous-liste des nombres pairs

listePair( $(34, 7, 42, 121)$ )

$(34, 42)$

**Algorithme** listePair($l$ : liste) : liste
**début**
  **si** listeEstVide ?($l$) **alors**
    **retourner** listeVide()
  **sinon**
    **si** modulo(listeTete($l$), 2) = 0 **alors**
      **retourner** listeCons( listeTete($l$), listePair( listeQueue( $l$ ) ))
    **sinon**
      **retourner** listePair( listeQueue( $l$ ) )
    **fin si**
  **fin si**
**fi**

# Exécution de l'algorithme

Calcul de listePair( $(34, 7, 42, 121)$ ).

1. listePair$((34, 7, 42, 121))$ = listeCons(34,listePair$((7, 42, 121))$)
2. ——> listePair$((7, 42, 121))$ = listePair$((42, 121))$
3. ————-> listePair$((42, 121))$ = listeCons(42, listePair$((121))$)
4. ——————-> listePair$((121))$ = listePair$(())$
5. ————————-> listePair$(())$ = listeVide()
6. ——————-> listePair$((121))$ = listeVide()
7. ————-> listePair$((42, 121))$ = listeCons(42, listeVide())
8. ——> listePair$((7, 42, 121))$ = listeCons(42, listeVide())
9. listePair$((34, 7, 42, 121))$ = listeCons(34, listeCons(42, listeVide()))

# Extraction en maple

```
listePair := proc(l)
# l : liste
# sortie : liste des nombres pairs de la liste l

  if listeEstVide?(l) then
    RETURN( listeVide() );
  else
    if modulo(listeTete(l), 2) = 0 then
      RETURN( listeCons(listeTete(l), listePair(listeQueue(l
    else
      RETURN( listePair(listeQueue(l)) );
    fi;
  fi;

end;
```

# Construction d'une liste de $n$ éléménts identique

consN( 10, 34 )

$$(34, 34, 34, 34, 34, 34, 34, 34, 34, 34)$$

**Algorithme** consN($p$ : entier, *elem* : entier) : liste
**début**
  **si** $p = 0$ **alors**
    **retourner** listeVide()
  **sinon**
    **retourner** listeCons( elem, consN($p - 1$, elem))
  **fin si**
**fin**

# Ajout d'un élément

ajout( 2, 23, $(34, 7, 42, 121)$ )

$(34, 7, 23, 42, 121)$

ajout( 0, 23, $(34, 7, 42, 121)$ )

$(23, 34, 7, 42, 121)$

**Algorithme** ajout($p$ : entier, *elem* : entier, $l$ : liste) : liste
**début**
  **si** listeEstVide ?($l$) **alors**
    **retourner** listeCons(*elem*, $l$)
  **sinon**
    **si** $p = 0$ **alors**
      **retourner** listeCons(*elem*, $l$)
    **sinon**
      **retourner** listeCons(listeTete($l$), ajout( $p - 1$, *elem*, listeQueue( $l$ )))
    **fin si**
  **fin si**

# Supression d'un élément

suppression( 2, $(34, 7, 23, 42, 121)$ )

$(34, 7, 42, 121)$

suppression( 0, $(23, 34, 7, 42, 121)$ )

$(34, 7, 42, 121)$

**Algorithme** suppression($p$ : entier, $l$ : liste) : liste
**début**
  **si** listeEstVide ?($l$) **alors**
    **retourner** $l$
  **sinon**
    **si** $p = 0$ **alors**
      **retourner** listeQueue($l$)
    **sinon**
      **retourner** listeCons( listeTete($l$), suppression($p - 1$, listeQueue( $l$ )))
    **fin si**
  **fin si**

# Objectifs de la séance 10

1. Connaître les opérations de base sur les listes
2. Calculer la longueur d'une liste
3. Extraction de sous-listes d'une liste
4. Utiliser un accumulateur avec une liste

**Questions principales du jour :**

Comment traiter les informations les unes après les autres ?