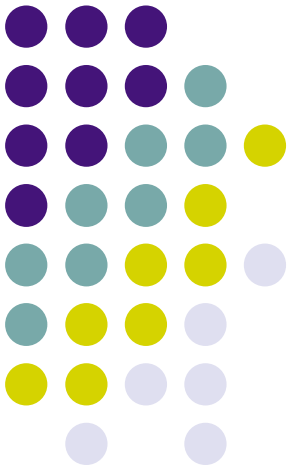
# FSAB1402: Informatique 2

# Algorithmes sur les Listes

**Peter Van Roy**

Département d'Ingénierie Informatique, UCL

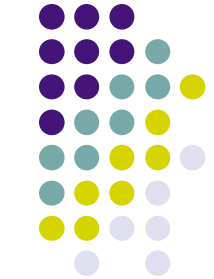
pvr@info.ucl.ac.be



# Ce qu'on va voir aujourd'hui

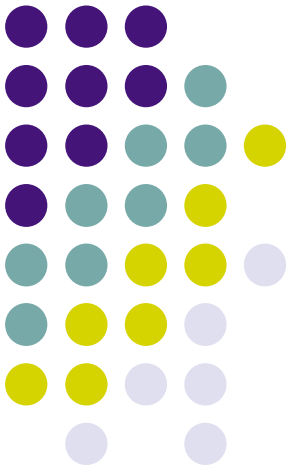
- Résumé des derniers cours
- Techniques de programmation
  - Utilisation de variables non-liées
  - Amélioration de l'efficacité avec un accumulateur
  - Utilisation du type pour construire une fonction récursive
- Algorithme de tri: Mergesort
- Programmer avec plusieurs accumulateurs
  - Programmer avec un état

# Suggestions de lecture pour ce cours

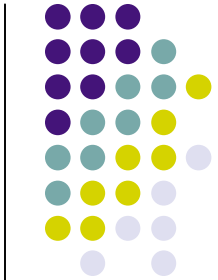


- Chapitre 1 (sections 1.4-1.6):
  - Listes et fonctions sur les listes, fonctions correctes
- Chapitre 3 (section 3.4.1):
  - Notation des types
- Chapitre 3 (section 3.4.2):
  - Programmer avec les listes
- Chapitre 3 (section 3.4.3):
  - Les accumulateurs

# Résumé des deux derniers cours



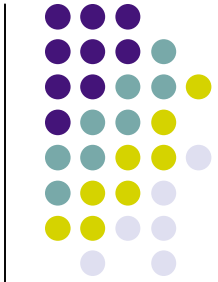
# Récursion sur les listes



- Définir une fonction {Nth L N} qui renvoie la nième élément de L
- Raisonnement:
  - Si N==1 alors le résultat est L.1
  - Si N>1 alors le résultat est {Nth L.2 N-1}
- Voici la définition complète:

```
fun {Nth L N}
   if N==1 then L.1
   elseif N>1 then
      {Nth L.2 N-1}
   end
end
```

# Pattern matching (correspondance des formes)

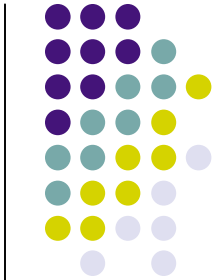


- Voici une fonction avec plusieurs formes:

```
fun {Length Xs}
   case Xs
   of nil then 0
   [] X|Xr then 1+{Length Xr}
   [] X1|X2|Xr then 2+{Length Xr}
   end
end
```

- Comment les formes sont-elles choisies?

# Complexité calculatoire



- Complexité temporelle et spatiale
- Meilleur cas, pire cas et cas moyen
- Analyse asymptotique
- La notation *O*
- Les notations $\Theta$, $\Omega$
  - Attention: $\Theta$ est plus difficile que *O* et $\Omega$!
  - *O* : borne supérieure
  - $\Omega$ : borne inférieure
  - $\Theta$ : borne inférieure et supérieure
- Complexité temporelle d'un algorithme récursif
- Complexité en moyenne

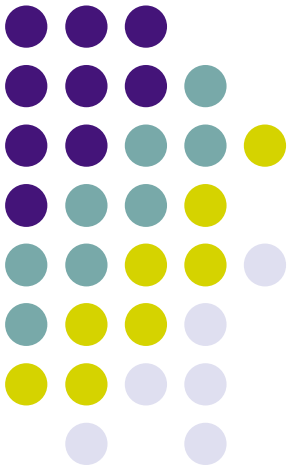
# Complexité polynomiale

- Voici une version efficace pour calculer le triangle de Pascal:

```
fun {FastPascal N}
   if N==1 then [1]
   else L in
      L={FastPascal N-1}
      {AddList {ShiftLeft L} {ShiftRight L}}
   end
end
```

- La complexité en temps est $O(n^2)$
- Complexité polynomiale: un polynome en $n$

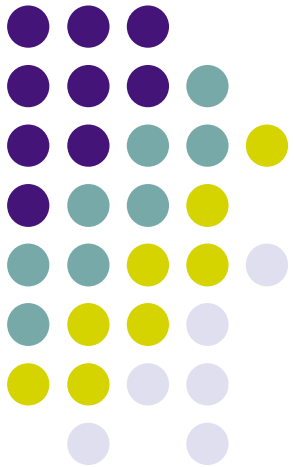
# Techniques de programmation



# Techniques de programmation

- Nous avons déjà vu quelques fonctions récursives sur les listes
- Nous allons maintenant approfondir les techniques de programmation sur les listes
  - Utiliser les variables non-liées pour faciliter la récursion terminale (exemple d'Append)
  - Utiliser un accumulateur pour augmenter l'efficacité d'un algorithme (exemple de Reverse)
  - Utiliser un type pour définir une fonction récursive (exemple de Flatten)

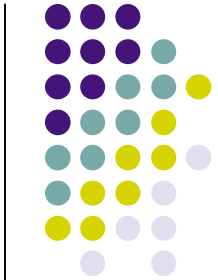
# Récursion terminale avec les listes



# La fonction Append

- Définissons une fonction {Append Xs Ys} qui construit une liste qui est la concaténation de Xs et Ys
  - Le résultat contient les éléments de Xs suivi par les éléments de Ys
- Nous faisons un raisonnement inductif sur le premier argument Xs
  - Si Xs==nil alors le résultat est Ys
  - Si Xs==X|Xr alors le résultat est X|{Append Xr Ys}
- Faites un dessin pour comprendre cette induction!

# Exécution d'Append



- {Append [1 2] [a b]}
  → 1| {Append [2] [a b]}
  → 1| 2| {Append nil [a b]}
  → 1| 2| [a b]

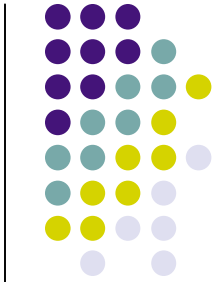
# Définition d'Append

- Voici la définition de Append:

```
fun {Append Xs Ys}
      case Xs
      of nil then Ys
      [] X|Xr then X|{Append Xr Ys}
      end
end
```

- Est-ce que cette définition fait la récursion terminale?
- Pour le savoir, il faut la traduire en langage noyau

# Append en langage noyau (1)

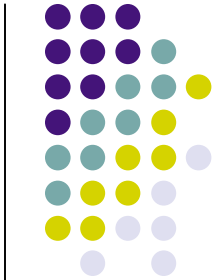


- Voici une traduction naïve de cette définition:

```
proc {Append Xs Ys Zs}
   case Xs
   of nil then Zs=Ys
   [] X|Xr then
      local Zr in
         {Append Xr Ys Zr}
         Zs=X|Zr
      end
   end
end
```

- L'appel récursif n'est pas le dernier appel!

# Append en langage noyau (2)

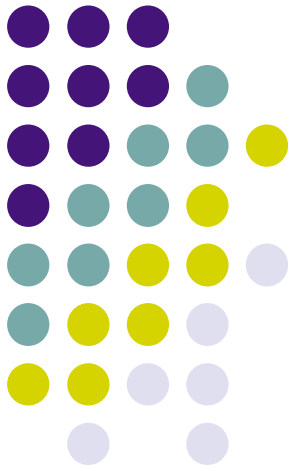


- Voici la vraie traduction de la définition d'Append:

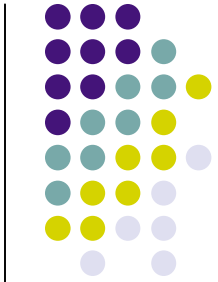
```
proc {Append Xs Ys Zs}
      case Xs
      of nil then Zs=Ys
      [] X|Xr then
          local Zr in
              Zs=X|Zr
              {Append Xr Ys Zr}
          end
      end
end
```

- L'appel récursif est le dernier appel!
- On peut faire Zs=X|Zr avant l'appel parce que Zr est une variable qui n'est pas encore liée (un "trou" dans la liste X|Zr)
- Technique: dans la construction de listes, on peut utiliser les variables non-liées pour assurer la récursion terminale

# Les accumulateurs avec les listes



# La fonction Reverse

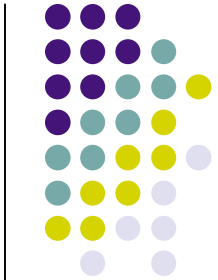


- Définissons une fonction qui prend une liste et qui renvoie une liste avec les éléments dans l'ordre inversé
  - {Reverse [1 2 3]} = [3 2 1]
- Voici une définition qui utilise Append:

```
fun {Reverse Xs}
      case Xs
      of nil then nil
      [] X|Xr then {Append {Reverse Xr} [X]}
      end
end
```

- Cette définition fait deux boucles imbriquées
  - Quelles sont ces boucles?

# Exécution de Reverse



- Xs=[1 2 3 4]
- X|Xr=[1 2 3 4]
  → X=1, Xr=[2 3 4]
- {Reverse Xr}=[4 3 2]
- {Append {Reverse Xr} [X]}
  → {Append [4 3 2] [1]}
  → [4 3 2 1]

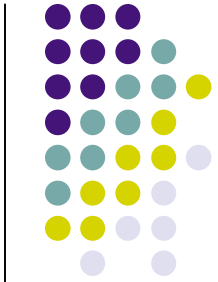
# Complexité temporelle de Reverse

- La fonction {Reverse Xs} a un temps d'exécution qui est $O(n^2)$ avec $n$=|*Xs*|
- C'est curieux que le calcul de l'inverse d'une liste de longueur $n$ prend un temps proportionnel au carré de $n$ !
- On peut faire beaucoup mieux en utilisant un accumulateur
  - Notez que notre première définition n'utilise pas d'accumulateur

# Reverse avec un accumulateur

- Il faut un invariant
- Prenons l'invariant suivant:
  $L = reverse(L_2)$ ++ $L_1$
- Ici, **++** et ***reverse*** sont des fonctions mathématiques
  - ++ fait la concaténation des listes
  - Ce ne sont pas des fonctions écrites en Oz!
  - Rappel: un invariant est une formule mathématique
- Nous avons donc une paire $(L_1, L_2)$
  - Quelles sont les transitions de cette paire?

# Vases communicants



- $L_1$=[1 2 3 4], $L_2$=nil
- $L_1$=[2 3 4], $L_2$=[1]
- $L_1$=[3 4], $L_2$=[2 1]
- $L_1$=[4], $L_2$=[3 2 1]
- $L_1$=nil, $L_2$=[4 3 2 1]

# Transitions de l'accumulateur

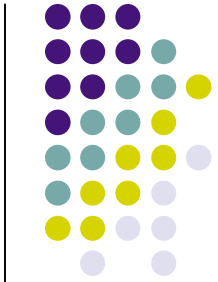
- Nous avons une paire $(L_1, L_2)$
- L'état initial est $(L, \text{nil})$
- La transition est:
  - $(X|L_1, L_2) \Rightarrow (L_1, X|L_2)$
- Ceci est correct parce que si:

  $L = reverse(L_2) ++ (X|L_1)$

  alors nous pouvons vérifier que:

  $L = reverse(X|L_2) ++ L_1$

# Définition de Reverse avec un accumulateur

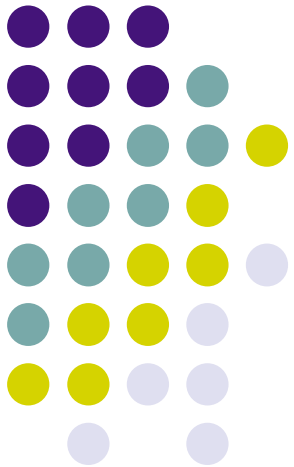


- Voici la nouvelle définition:

```
fun {Reverse L1 L2}
      case L1
      of nil then L2
      [] X|M1 then {Reverse M1 X|L2}
      end
end
```

- Exemple d'un appel: {Reverse [1 2 3] nil}
- La complexité de cette définition est *O(n)* avec *n*=|*L1*|

# Utiliser le type pour la récursion



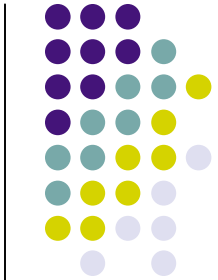
# La fonction Flatten

- Pour compléter le parcours des trois techniques, voici une fonction un peu plus compliquée
- Nous voulons définir la fonction {Flatten Xs} qui prend une liste Xs qui peut contenir des éléments qui sont eux-mêmes des listes, et ainsi de suite, et qui renvoie une liste de tous ces éléments
- Exemple: {Flatten [a [[b]] [c nil d]]} = [a b c d]

# Utiliser un type pour définir une fonction

- Pour définir {Flatten Xs}, nous allons d'abord définir le type de l'argument Xs, <NestedList T>
  - En suivant le type, la définition sera simple
- <NestedList T> ::= nil
  | <NestedList T> '|' <NestedList T>
  | T '|' <NestedList T>
- Pour que le choix de l'alternatif soit non-ambigu, il faut que T ne soit ni nil ni une liste élémentaire (un "cons")
  - **fun** {IsCons X} **case** X **of** _ | _ **then true else false end end**
  - **fun** {IsList X} X==nil **orelse** {IsCons X} **end**

# Définition de Flatten

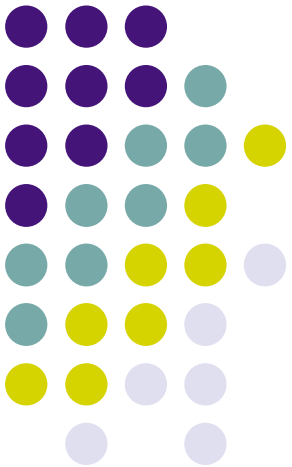


- Voici la définition:

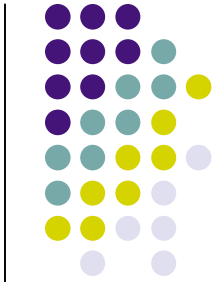
```
fun {Flatten Xs}
   case Xs
   of nil then nil
   [] X|Xr andthen {IsList X} then
         {Append {Flatten X} {Flatten Xr}}
   [] X|Xr then
         X|{Flatten Xr}
   end
end
```

- Pour les avertis: faites une version de Flatten qui utilise un accumulateur!

# Algorithmes de tri



# Algorithmes de tri

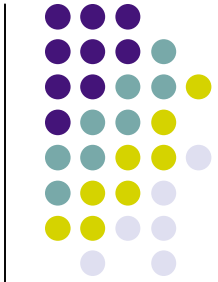


- Un algorithme de tri prend une liste d'éléments et renvoie une liste avec les mêmes éléments rangés selon un ordre
- Il y a beaucoup d'algorithmes de tri différents
  - Tri par sélection, tri par insertion
  - Mergesort (tri par divisions et fusions récursives)
  - Heapsort (tri par construction de tas ("heap") )
  - Quicksort (tri par partitionnement récursif)

# Mergesort

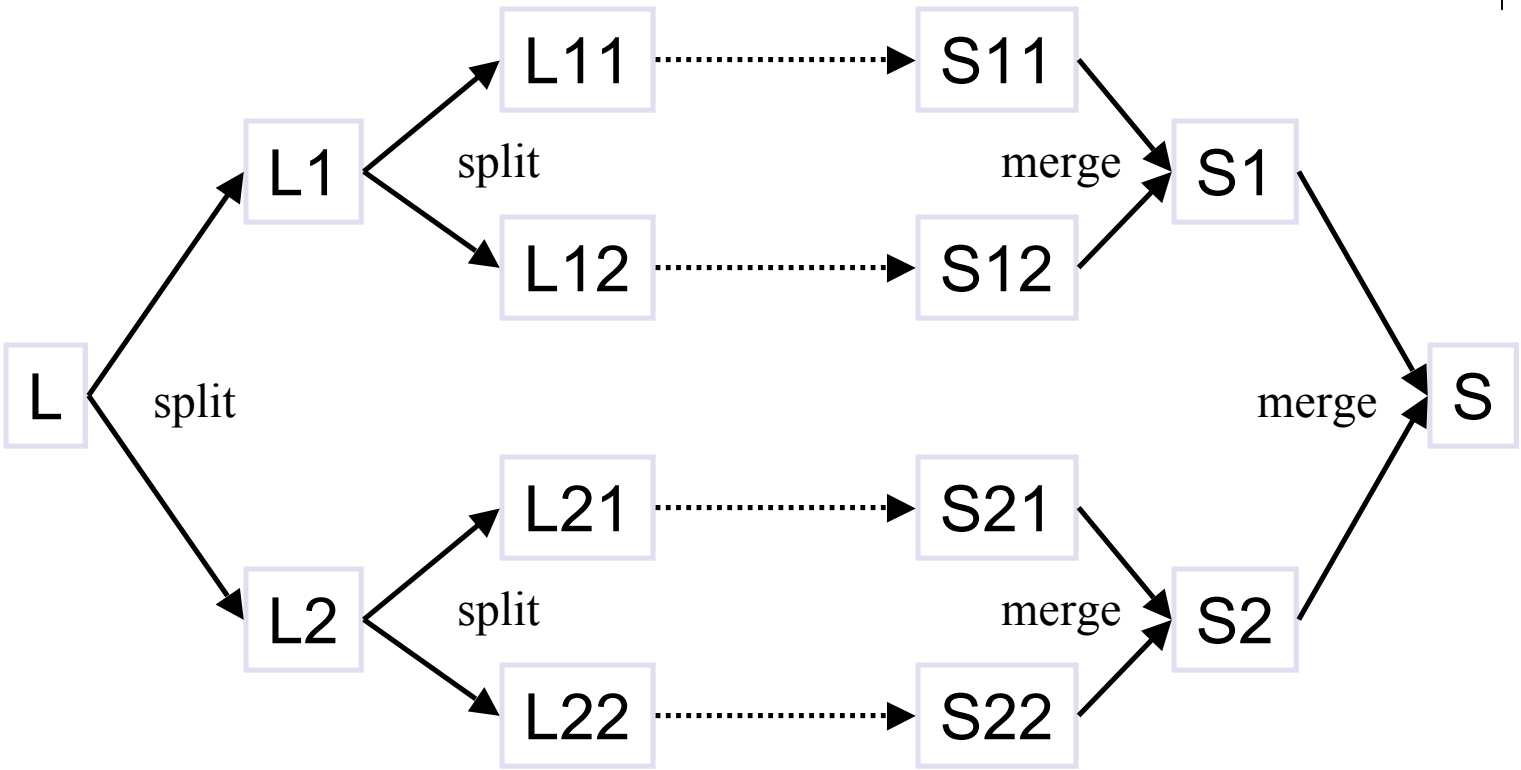
- Cet algorithme peut trier une liste de taille *n* en un temps maximal de *O(n* log *n)*
- L'algorithme peut facilement être programmé dans le modèle déclaratif
- L'algorithme utilise une technique générale appelée "diviser pour régner"
  - Diviser la liste en deux listes
  - Utiliser mergesort récursivement pour trier les deux listes
  - Fusionner les deux listes pour obtenir le résultat

# Exemple de Mergesort (1)



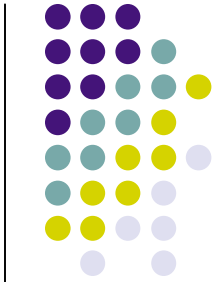
- Prenons la liste L=[5 2 6 4 3]
- Diviser L en deux listes:
  - L1=[5 2], L2=[6 4 3]
- Trier chacune des deux listes:
  - S1=[2 5], S2=[3 4 6]
- Fusionner les deux listes S1 et S2:
  - Ceci est la clé de l'algorithme!
  - On peut le faire en traversant chaque liste au maximum une fois (voir dessin sur le tableau)
- Le résultat est la liste triée S=[2 3 4 5 6]

# Exemple de Mergesort (2)



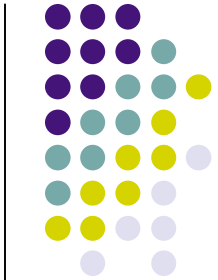

- Observez comment est faite la récursion!

# Définition de Mergesort



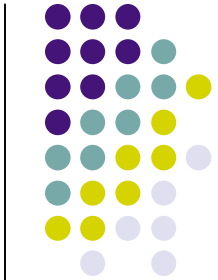
```
fun {Mergesort Xs}
   case Xs
   of nil then nil
   [] [X] then [X]
   else Ys Zs in
      {Split Xs Ys Zs}
      {Merge {Mergesort Ys} {Mergesort Zs}}
   end
end
```

# Définition de Split



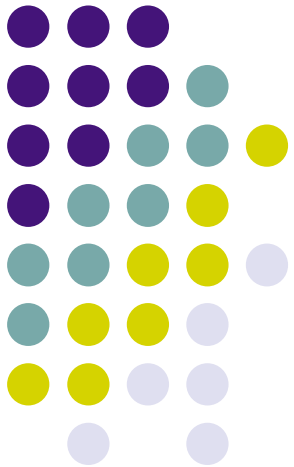
```
proc {Split Xs Ys Zs}
   case Xs
   of nil then Ys=nil Zs=nil
   [] [X] then Ys=Xs Zs=nil
   [] X1|X2|Xr then Yr Zr in
      Ys=X1|Yr
      Zs=X2|Zr
      {Split Xr Yr Zr}
   end
end
```

# Définition de Merge



- **fun** {Merge Xs Ys}
  **case** Xs#Ys
  **of** nil#Ys **then** Ys
  **[]** Xs#nil **then** Xs
  **[]** (X|Xr)#(Y|Yr) **then**
  **if** X<Y **then** X|{Merge Xr Ys}
  **else** Y|{Merge Xs Yr} **end**
  **end**
  **end**

# Programmer avec des accumulateurs



# C’est quoi en fait un accumulateur?

- Un accumulateur peut être vu comme une paire de **deux arguments**, une entrée et une sortie
- Par exemple, dans la fonction Reverse, nous avons (1) l’argument L2 et (2) le résultat de la fonction:

```
fun {Reverse L1 L2}
      … {Reverse M1 X|L2}
end
```

- On voit mieux les deux arguments si on écrit Reverse en langage noyau: *(comme toujours, on voit mieux en langage noyau!)*

```
proc {Reverse L1 L2 R2}
      … {Reverse M1 X|L2 R2}
end
```

- L’accumulateur est la paire (L2,R2)
  - L2 est l’entrée, R2 est la sortie

# Programmer avec plusieurs accumulateurs

- Nous avons déjà vu comment écrire une fonction avec un seul accumulateur
  - L'accumulateur = un des arguments de la fonction + le résultat de la fonction
  - En langage noyau, on voit bien qu'un accumulateur est une paire de deux arguments, une entrée et une sortie
- Nous avons vu que l'utilisation d'un accumulateur est une bonne idée pour l'efficacité (récursion terminale ⇒ une boucle)
- Maintenant, nous allons voir comment on peut écrire un programme avec plusieurs accumulateurs
  - On verra plus tard que l'utilisation d'accumulateurs n'est rien d'autre que programmer avec un état: chaque accumulateur correspond à une variable à affectation multiple

# Programmer avec accumulateurs = programmer avec un état

- L'état d'un programme est l'ensemble de données importantes pour le programme à un instant donné
  - L'état est passé partout dans le programme et transformé successivement pour obtenir un résultat
  - L'état = la valeur de tous les accumulateurs
- L'état S est fait de plusieurs parties, qui sont en fait des accumulateurs:

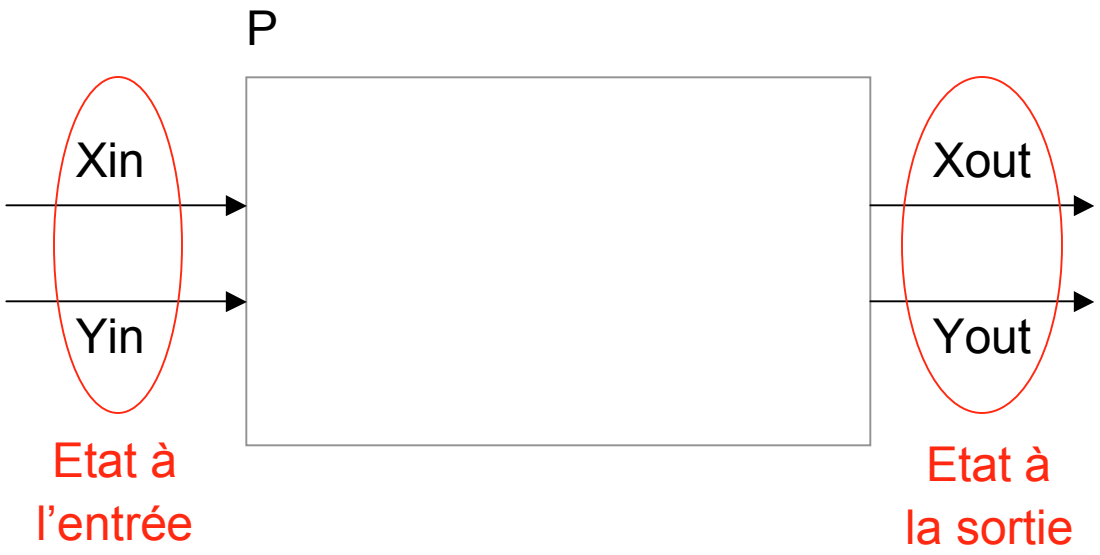
  S=(X,Y,Z, …)

- Pour chaque procédure P, l'entête devient:

  **proc** {P $X_{in}$ $X_{out}$ $Y_{in}$ $Y_{out}$ $Z_{in}$ $Z_{out}$ …}

# Schéma général d'une procédure (1)

- Voici un diagramme qui montre une procédure P avec deux accumulateurs (quels sont ces accumulateurs?)



# Schéma général d'une procédure (2)

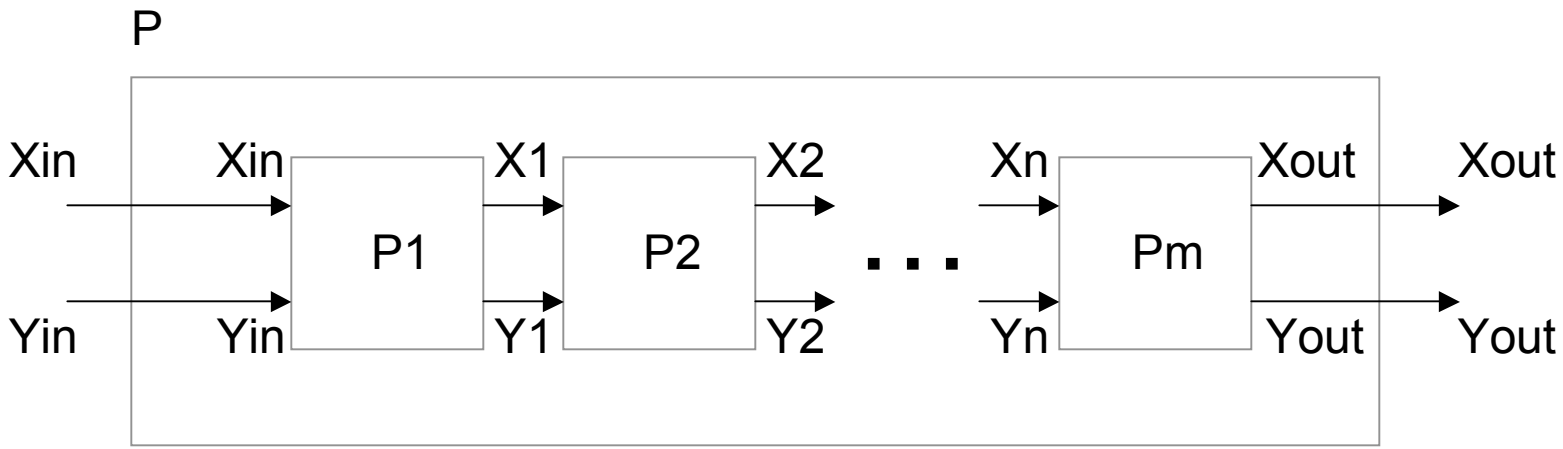
- L'état S de cet exemple contient deux parties:
  S=(X,Y)
- Voici une définition possible de la procédure P:

```
proc {P Xin Xout Yin Yout}
        {P1 Xin X1 Yin Y1}
        {P2 X1 X2  Y1 Y2}
        ....
        {Pm Xn Xout  Yn Yout}
end
```

- Si le nombre d'accumulateurs est plus grand qu'un, comme ici, alors il est plus facile d'utiliser des procédures au lieu des fonctions

# Schéma général d'une procédure (3)

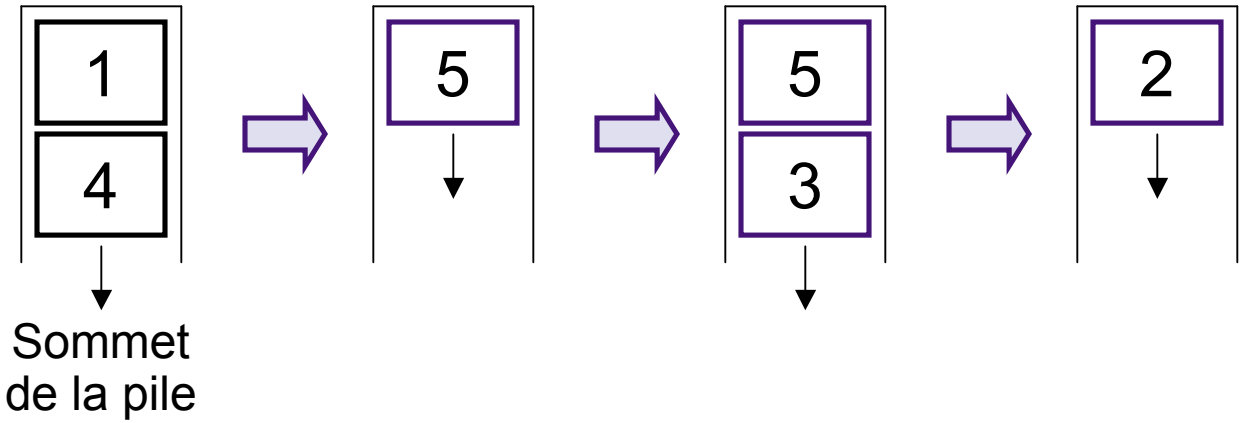
- Voici un diagramme qui montre la définition de la procédure P qui a deux accumulateurs



# Exemple avec deux accumulateurs

- Supposons qu'on dispose d'une machine à pile pour évaluer des expressions arithmétiques
- Par exemple: (1+4)-3
- La machine exécute les instructions suivantes:

push(1)
push(4)
plus
push(3)
minus



# Compilateur pour machine à pile (1)

- Définissez une procédure qui prend une expression arithmétique, exprimée comme une structure de données, et qui calcule deux résultats: (1) une liste d'instructions pour une machine à pile et (2) un compte du nombre d'instructions
  - L'expression (1+4)-3 est exprimée comme [[1 plus 4] minus 3]
- La procédure a l'entête suivante:

  **proc** {ExprCode Expr Cin Cout Nin Nout}
- Il y a deux accumulateurs, C et N:
  - Cin: liste d'instructions initiale
  - Cout: liste d'instructions finale
  - Nin: compte d'instructions initial
  - Nout: compte d'instructions final

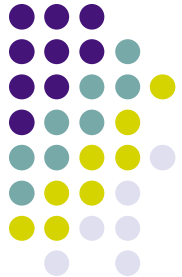
# Compilateur pour machine à pile (2)

- **proc** {ExprCode Expr C0 C N0 N}
  **case** Expr
  **of** [E1 plus E2] **then** C1 N1 **in**
  C1=plus|C0
  N1=N0+1
  {SeqCode [E2 E1] C1 C N1 N}
  **[]** [E1 minus E2] **then** C1 N1 **in**
  C1=minus|C0
  N1=N0+1
  {SeqCode [E2 E1] C1 C N1 N}
  **[]** I **andthen** {IsInt I} **then**
  C=push(I)|C0
  N=N0+1
  **end**
  **end**

# Compilateur pour machine à pile (3)

- **proc** {SeqCode Es C0 C N0 N}
  **case** Es
  **of** nil **then** C=C0 N=N0
  **[]** E|Er **then** C1 N1 **in**
  {ExprCode E C0 C1 N0 N1}
  {SeqCode Er C1 C N1 N}
  **end**
  **end**

# Un autre exemple (1)



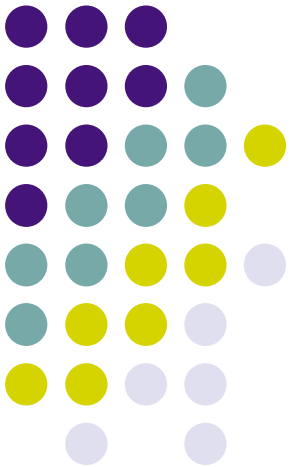
- On peut faire une version de Mergesort qui utilise un accumulateur
- **proc** {Mergesort1 N S0 S Xs}
  - N est un entier
  - S0 est l'entrée: une liste à trier
  - S est la sortie: le reste de S0 après que les premiers N éléments sont triés
  - Xs est la liste triée des premiers N éléments de S0
- La paire (S0,S) est un accumulateur
- La définition utilise une syntaxe de procédure parce qu'elle a deux sorties, S et Xs

# Un autre exemple (2)

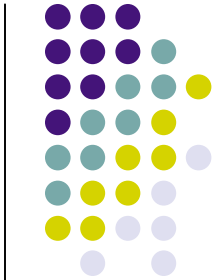
```
fun {Mergesort Xs}
Ys in
   {Mergesort1 {Length Xs} Xs _ Ys}
   Ys
end
```

```
proc {Mergesort1 N S0 S Xs}
   if N==0 then S=S0 Xs=nil
   elseif N==1 then
      case S0 of X|S1 then
         S=S1 Xs=[X]
      end
   else S1 Xs1 Xs2 NL NR in
      NL=N div 2
      NR=N-NL
      {Mergesort1 NL S0 S1 Xs1}
      {Mergesort1 NR S1 S Xs2}
      Xs={Merge Xs1 Xs2}
   end
end
```

# Résumé



# Résumé



- Techniques de programmation
  - Utilisation d'une variable non-liée pour construire une liste avec l'appel récursif en dernier
  - Utilisation d'un accumulateur pour augmenter l'efficacité
  - Utilisation d'un type pour construire une fonction
- Algorithme de Mergesort
  - Exemple de diviser pour règner
- Programmer avec plusieurs accumulateurs
  - Programmer avec un état qui est passé partout dans un programme
  - Exemple d'un compilateur pour machine à pile