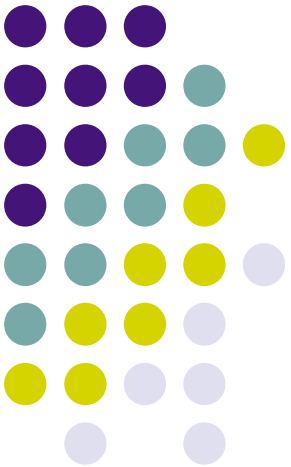
FSAB1402: Informatique 2

Récursion sur les Listes

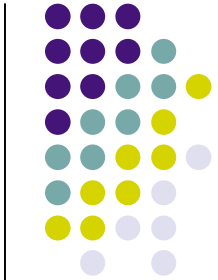
Peter Van Roy

Département d'Ingénierie Informatique, UCL

pvr@info.ucl.ac.be



Ce qu'on va voir aujourd'hui

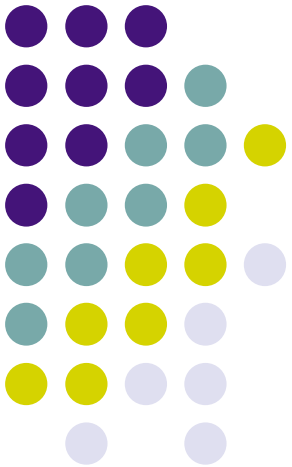


- Résumé du dernier cours
- Les listes
- Récursion sur les listes
- Pattern matching
- Représentation des listes en mémoire

Lecture
pour le troisième cours

- Chapitre 1 (sections 1.4 et 1.5):
  - Listes et fonctions sur les listes
- Chapitre 2 (section 2.3):
  - Langage noyau du modèle déclaratif
- Chapitre 2 (section 2.6):
  - Traduction d'un programme en langage noyau
- Chapitre 3 (section 3.4.1):
  - Notation des types
- Chapitre 3 (section 3.4.2):
  - Programmer avec les listes

Résumé du dernier cours



Programmer dans le modèle déclaratif

- Règle d'or: (sauf cas particuliers) les appels récursifs doivent toujours être les dernières instructions (la récursion terminale)
  - Si l'appel récursif est la dernière instruction, alors la fonction récursive se comporte exactement comme une boucle dans un langage impératif
    - Espace mémoire constant, temps d'exécution proportionnel au nombre d'itérations
  - Attention: si vous avez deux boucles imbriquées, alors vous avez besoin de deux fonctions récursives (une fonction qui appelle l'autre)!
    - Erreur fréquente: un programmeur débutant qui essaie de forcer une seule boucle à faire le travail de plusieurs boucles
- La programmation avec accumulateurs est recommandée
  - Avec un accumulateur, il est facile d'utiliser la récursion terminale
- Quand la boucle est compliquée, il faut utiliser un invariant
  - On peut utiliser un invariant pour dériver un accumulateur
  - Comment trouver un bon invariant? Les vases communicants!

Comparaison des boucles en déclaratif et en impératif

- Une boucle dans le modèle déclaratif:

```
fun {While S}
   if {IsDone S} then S
   else {While {Transform S}} end /* récursion */
end
```

- Une boucle dans un langage impératif (= un langage avec affectation multiple):

```
state whileLoop(state s) {
   while (!isDone(s))
      s=transform(s); /* affectation */
   return s;
}
```

- Dans les deux cas, il faut raisonner avec un invariant
- Exercice pour vous: utiliser la récursion dans le langage impératif

Comment trouver un bon invariant?

- Principe des vases communicants
  - Une formule en deux parties
  - Une partie “disparaît”; l’autre “accumule” le résultat
- Exemple: calcul efficace des nombres de Fibonacci
  - Définition: $F_0=0$, $F_1=1$, $F_n=F_{n-1}+F_{n-2}$ si $n>1$
  - Invariant: le triplet $(n-i,F_{i-1},F_i)$
    - “Invariant” parce que les trois parties du triplet doivent toujours satisfaire cette relation
  - Un pas de l’exécution: $(k,a,b) \Rightarrow (k-1,b,a+b)$
  - Valeur initiale: $(n-1,0,1)$

Calcul efficace des nombres de Fibonacci

- Raisonnement sur l'invariant
  - Invariant: le triplet $(n-i,F_{i-1},F_i)$
  - Un pas de l'exécution: $(k,a,b) \Rightarrow (k-1,b,a+b)$
  - Valeur initiale: (n-1,0,1)
  - Valeur finale: $(0,F_{n-1},F_n)$
- Définition de la fonction:

```
fun {Fibo K A B}
      if K==0 then B
      else {Fibo K-1 B A+B} end
end
```

- Appel initial: {Fibo N-1 0 1}

Environnement contextuel d'une procédure (1)

- Quand on définit une procédure, on fait deux choses
  - Dans le monde visible: Déclaration de l'identificateur
  - Dans le monde invisible: Création de la procédure en mémoire avec le code de la procédure et son environnement contextuel
- Une procédure se souvient de l'endroit de sa naissance (c'est son "environnement contextuel")
- Quel est l'environnement contextuel de la procédure Iterate avec cette définition?

```
declare
fun {Iterate Si}
      if {IsDone Si} then Si
      else {Iterate {Transform Si}} end
end
```

Environnement contextuel d'une procédure (2)

- Pour bien distinguer (1) la déclaration de l'identificateur et (2) la création de la procédure en mémoire, on peut écrire:

(2) procédure en mémoire

(1) identificateur

```
declare
Iterate = fun {$ Si}
             if {IsDone Si} then Si
             else {Iterate {Transform Si}} end
          end
```

- La syntaxe **fun** {$ Si} … **end** représente une fonction en mémoire (sans identificateur: c'est une **fonction anonyme**)
  - En Oz: Le symbole "$" prend la place de l'identificateur
- L'environnement contextuel contient donc {IsDone, Iterate, Transform}

La fonction en langage noyau: tout devient visible

- Voici la fonction Iterate de nouveau:

```
Iterate = fun {$ Si}
             if {IsDone Si} then Si
             else {Iterate {Transform Si}} end
          end
```

- En langage noyau cela donne:

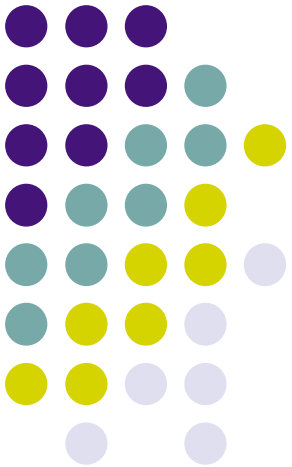
```
Iterate = proc {$ Si R}
             local B in
                {IsDone Si B}
                if B then R=Si else
                   local Sj in
                      {Transform Si Sj}
                      {Iterate Sj R}
                   end
                end
             end
          end
```

- Toutes les variables intermédiaires deviennent visibles, avec de nouveaux identificateurs: {B, Sj, R}

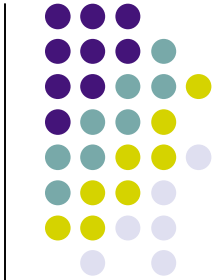
Le langage noyau du modèle déclaratif (en partie)

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | …
- <v> ::= <number> | <procedure> | …
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**

Les listes

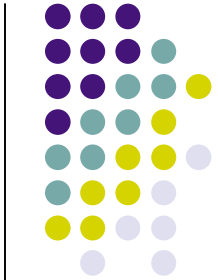


Pourquoi les listes?



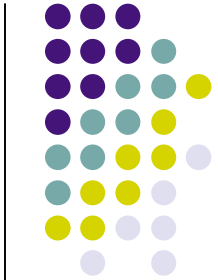
- Dans le dernier cours, on a vu la récursion sur les entiers
  - Mais un entier est assez limité
  - On voudrait faire des calculs avec beaucoup d'entiers en même temps!
- La liste
  - Une collection ordonnée d'éléments (une séquence)
  - Une des premières structures utilisées dans les langages symboliques (Lisp, dans les années 50)
  - La plus utile des structures composées

Définition intuitive



- Une liste est
  - la liste vide, ou
  - une paire (un *cons*) avec une *tête* et une *queue*
    - La tête est le premier élément
    - La queue est une liste (les éléments restants)

La syntaxe d'une liste

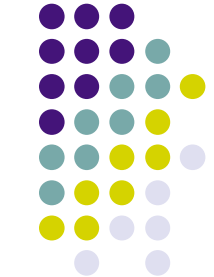


- Avec la notation EBNF:

  <List T> ::= nil | T '|' <List T>

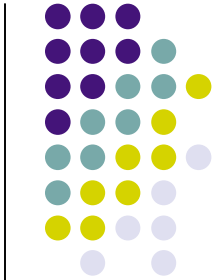
- <List T> représente une liste d'éléments de type T et T représente un élément de type T
- Attention à la différence entre | et '|'

Notation pour les types

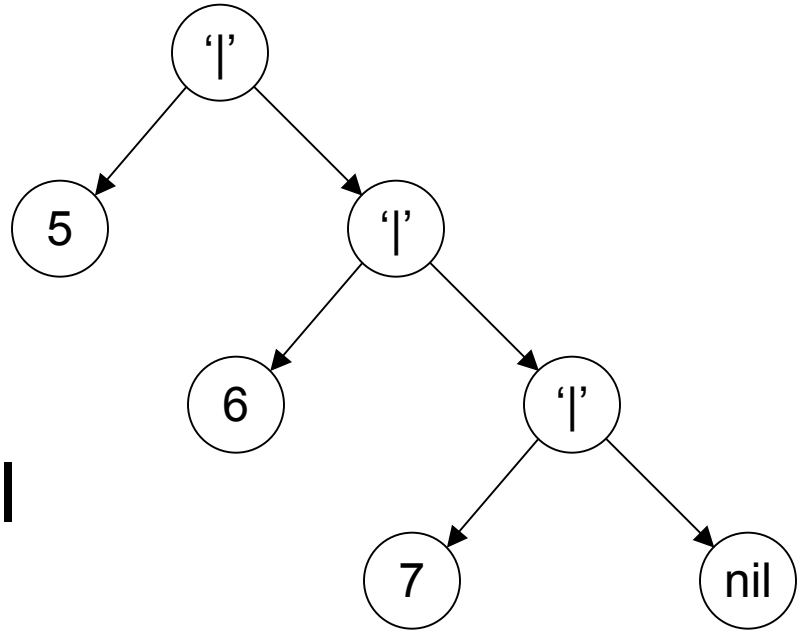


- <Int> représente un entier; plus précisément l'ensemble de toutes les représentations syntaxiques de tous les entiers
- <List <Int>> représente l'ensemble de toutes les représentations syntaxiques des listes d'entiers
- T représente l'ensemble des représentations syntaxiques de tous les éléments de type T; nous disons que T est une variable de type
  - Ne pas confondre avec une variable en mémoire ou un identificateur!

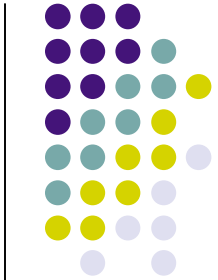
Syntaxe pour les listes (1)



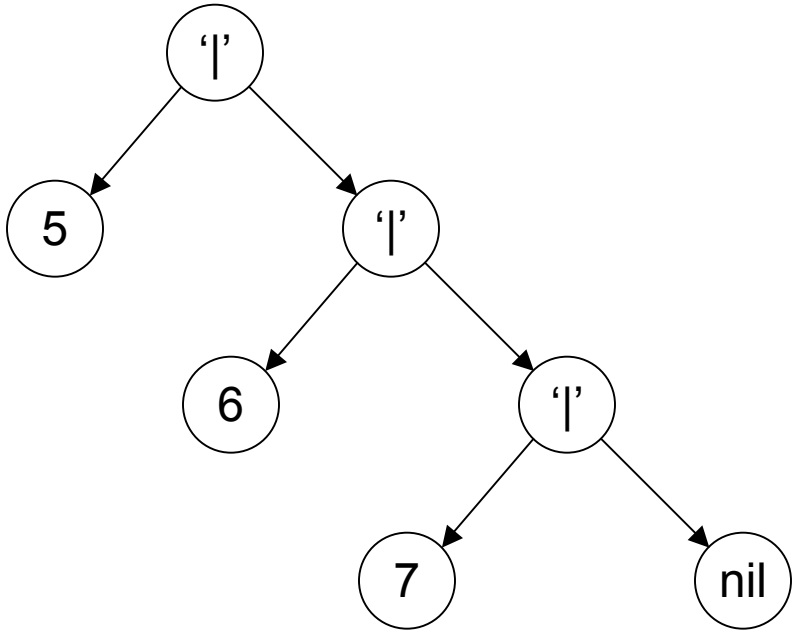
- Liste vide: nil
- Liste non-vide: H|T
  - L'opérateur infixe '|'
- nil, 5|nil, 5|6|nil, 5|6|7|nil
- nil, 5|nil, 5|(6|nil), 5|(6|(7|nil))



Syntaxe pour les listes (2)



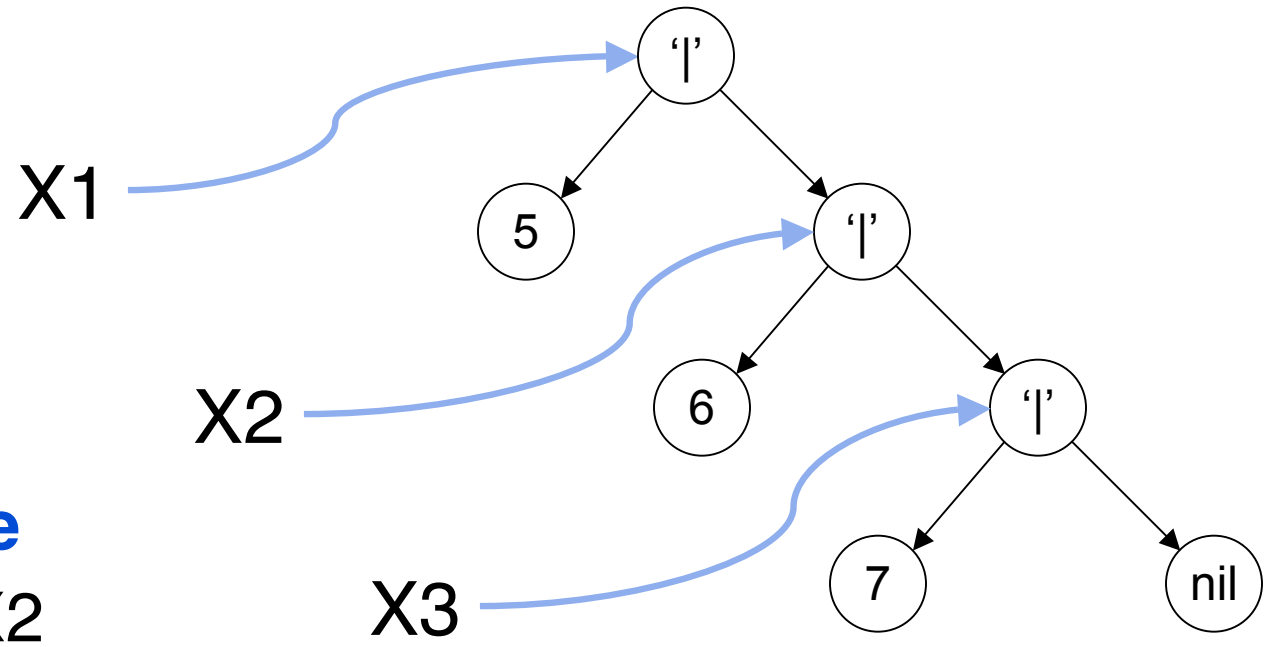
- Il existe un sucre syntaxique plus court
  - Sucre syntaxique = raccourci de notation qui n'a aucun effet sur l'exécution
- nil, [5], [5 6], [5 6 7]
- Attention: dans la mémoire de l'ordinateur, [5 6 7] et 5|6|7|nil sont identiques!



La syntaxe complète: une liste est un tuple

- Une liste est un cas particulier d'un tuple
- Syntaxe préfixe (l'opérateur '|' devant)
  - nil
    '|'(5 nil)
    '|'(5 '|'(6 nil))
    '|'(5 '|'(6 '|'(7 nil)))
- Syntaxe complète
  - nil,
    '|'(1:5 2:nil)
    '|'(1:5 2:'|'(1:6 2:nil))
    '|'(1:5 2:'|'(1:6 2:'|'(1:7 2:nil)))

La construction d'une liste



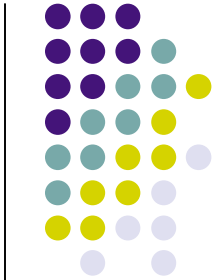
declare

X1=5|X2

X2=6|X3

X3=7|nil

Résumé des syntaxes possibles



- On peut écrire

  X1=5|6|7|nil

  qui est un raccourci pour

  X1=5|(6|(7|nil))

  qui est un raccourci pour

  X1='|'(5 '|'(6 '|'(7 nil)))

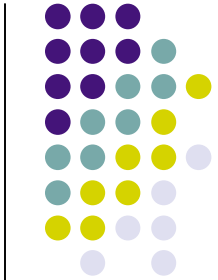
  qui est un raccourci pour

  X1='|'(1:5 2:'|'(1:6 2:'|'(1:7 2:nil)))

- La syntaxe la plus courte (attention au 'nil'!)

  X1=[5 6 7]

Calculer avec les listes



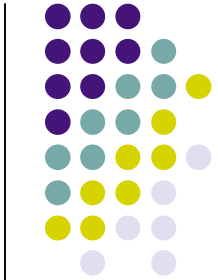
- Attention: une liste non vide est une paire!
- Accès à la tête

  X.1
- Accès à la queue

  X.2
- Tester si la liste X est vide:

  if X==nil **then** … **else** … **end**

La tête et la queue

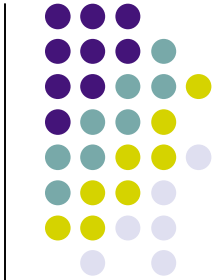


- On peut définir des fonctions

```
fun {Head Xs}
   Xs.1
end

fun {Tail Xs}
   Xs.2
end
```

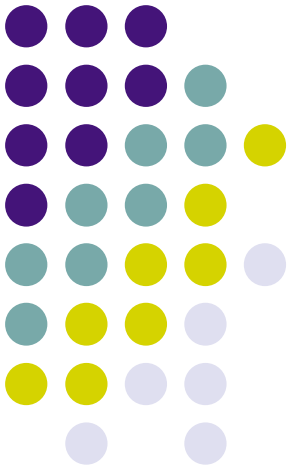
Exemple avec Head et Tail



- {Head [a b c]}
  donne a
- {Tail [a b c]}
  donne [b c]
- {Head {Tail {Tail [a b c]}}}
  donne c

- Dessinez les arbres!

Récursion sur les listes



Exemple de récursion sur une liste

- On a une liste d'entiers
- On veut calculer la somme de ces entiers
  - Définir la fonction Sum
- Définition inductive sur la structure de liste
  - Sum de la liste vide est 0
  - Sum d'une liste non vide L est

    {Head L} + {Sum {Tail L}}

Somme des éléments d'une liste (méthode naïve)

```
fun {Sum L}
   if L==nil then
      0
   else
      {Head L} + {Sum {Tail L}}
   end
end
```

Somme des éléments d'une liste (avec accumulateur)

```
fun {Sum2 L A}
   if L==nil then
      A
   else
      {Sum2 {Tail L} A+{Head L}}
   end
end
```

Transformer le programme pour obtenir l'accumulateur

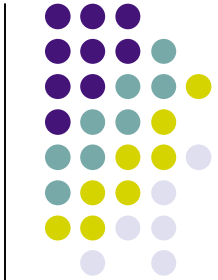
- Arguments:
  - {Sum L}
  - {Sum2 L A}
- Appels récursifs
  - {Head L} + {Sum {Tail L} }
  - {Sum2 {Tail L} A+{Head L} }
- Cette transformation marche parce que l'addition est associative
  - Sum fait (1+(2+(3+(4+0)))), Sum2 fait ((((0+1)+2)+3)+4)

Autre exemple: la fonction Nth

- Définir une fonction {Nth L N} qui renvoie le nième élément de L
- Le type de Nth est:

  <**fun** {$ <List T> <Int>}:<T>>
- Raisonnement:
  - Si N==1 alors le résultat est {Head L}
  - Si N>1 alors le résultat est {Nth {Tail L} N-1}

La fonction Nth

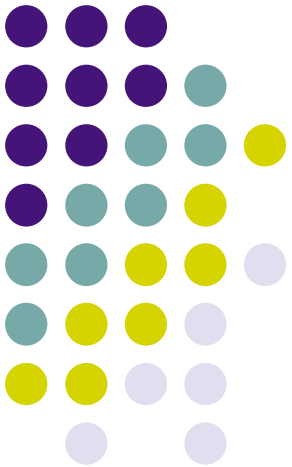


- Voici la définition complète:

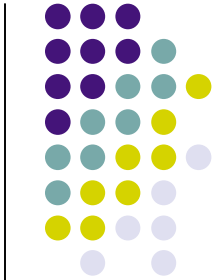
```
fun {Nth L N}
   if N==1 then {Head L}
   elseif N>1 then
      {Nth {Tail L} N-1}
   end
end
```

- Qu'est-ce qui se passe si le nième élément n'existe pas?

Pattern matching
(correspondance des formes)

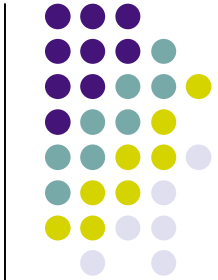


Sum avec pattern matching



```
fun {Sum L}
   case L
   of nil then 0
   [] H|T then H+{Sum T}
   end
end
```

Sum avec pattern matching

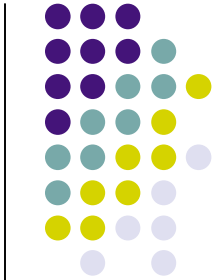


```
fun {Sum L}
   case L
   of nil then 0
   [] HlT then H+{Sum T}
   end
end
```

Une clause

- "nil" est la *forme* (*pattern*) de la clause

Sum avec pattern matching



```
fun {Sum L}
   case L
   of nil then 0
   [] H|T then H+{Sum T}
   end
end
```

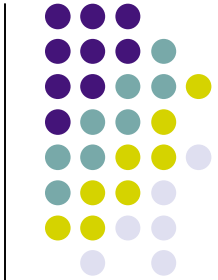
Une clause

- "H|T" est la *forme* (*pattern*) de la clause

Pattern matching (correspondance des formes)

- La première clause utilise **of**, les autres **[]**
- Les clauses sont essayées dans l'ordre
- Une clause correspond si sa forme correspond
- Une forme correspond, si l'étiquette (label) et les arguments correspondent
  - Les identificateurs dans la forme sont alors affectés aux parties correspondantes de la liste
- La première clause qui correspond est exécutée, pas les autres

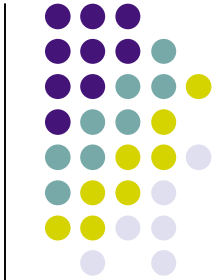
Longueur d'une liste



- Définition inductive
  - Longueur d'une liste vide est 0
  - Longueur d'une paire est 1 + longueur de la queue

```
fun {Length Xs}
   case Xs
   of nil then 0
   [] XlXr then 1+{Length Xr}
   end
end
```

Longueur d'une liste en langage noyau



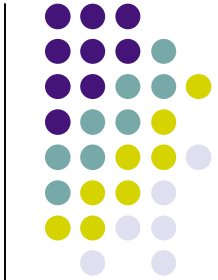
- Une version simple de l'instruction **case** fait partie du langage noyau

```
proc {Length Xs R}
   case Xs
   of nil then R=0
   else
      case Xs
      of XlXr then
         local R1 in
            {Length Xr R1}
            R=1+R1
         end
      else skip end
   end
end
```

Le langage noyau du modèle déclaratif (complet!)

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <procedure> | <record>
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
- <record>, <p> ::= <lit>(<f>$_1$:<x>$_1$ … <f>$_n$:<x>$_n$)

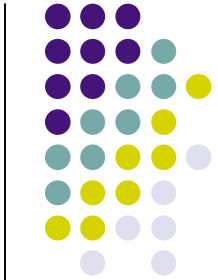
Longueur d'une liste (2)



- Une version avec une forme en plus!

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

Longueur d'une liste (3)



- Quelle forme ne sera jamais choisie?

```
fun {Length Xs}
   case Xs
   of nil then 0
   [] XlXr then 1+{Length Xr}
   [] X1lX2lXr then 2+{Length Xr}
   end
end
```

Pattern matching en général

- Le pattern matching peut être utilisé pour beaucoup plus que les listes
- Toute valeur, y compris nombres, atomes, listes, tuples, enregistrements
  - Nous allons voir les tuples dans le prochain cours
- Les formes peuvent être imbriquées
- Certains langages connaissent des formes encore plus générales (expressions régulières)

Calculs sur les listes

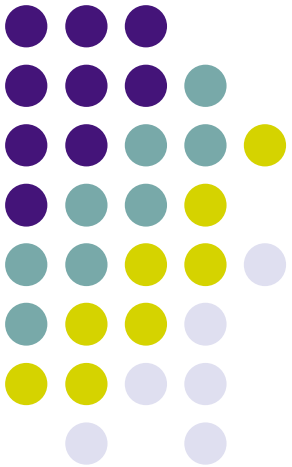
- Une liste est une liste vide ou une paire avec une tête et une queue
- Un calcul avec une liste est un calcul récursif
- Le pattern matching est une bonne manière d'exprimer de tels calculs

Méthode générale pour la récursion sur les listes

- Il faut traiter les listes de façon récursive
  - Cas de base: la liste est vide (nil)
  - Cas inductif: la liste est une paire (cons)

- Une technique puissante et concise
  - Le *pattern matching (correspondance des formes)*
  - Fait la correspondance entre une *liste* et une *forme* (“*pattern*”) et lie les identificateurs dans la forme

Représentation des listes en mémoire



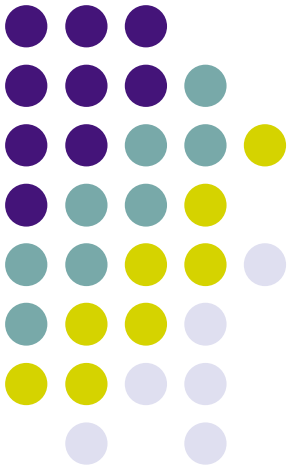
Représentation des listes en mémoire

- Comment est-ce qu'une liste est représentée en mémoire?
- On a déjà vu que la mémoire contient des variables
  - Les variables peuvent être liées ou non-liées
  - Une variable *x* peut être liée a un nombre ou un atome, *x*=*23* ou *x*=nil
- Pour les listes, on ajoute un seul concept: la liste élémentaire (un *cons* ou une *paire*)
  - Une variable *x* peut être liée a une liste élémentaire, *x*=*y*|*z*
  - La syntaxe *x=y|z* est un raccourci pour *x='|'(1:y 2:z)*
- Toute liste peut être décomposée en listes élémentaires

Décomposition en listes élémentaires

- Considérons l'instruction X=[1 2 3]
- Supposons que X correspond à la variable *x*
- En mémoire, il y aura donc ceci:
  - *x=a|y, y=b|z, z=c|w, w=nil, a=1, b=2, c=3*
  - *a, b, c, x, y, z, w* sont toutes des variables
- La correspondance des identificateurs avec les variables en mémoire se fait comme avant
  - Par exemple, R=X.2 fait une correspondance entre R et *y*

Résumé



Le langage noyau du modèle déclaratif (complet)

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <procedure> | <record>
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
- <record>, <p> ::= <lit>(<f>$_1$:<x>$_1$ … <f>$_n$:<x>$_n$)

Résumé

- Le langage noyau du modèle déclaratif
- Les listes
  - Différentes syntaxes sont possibles, mais il y a toujours la même représentation en mémoire
  - Récursion sur les listes
- Pattern matching avec l'instruction **case**
  - Fonctions sur les listes
- Représentation des listes en mémoire
  - Décomposition en listes élémentaires