

# FSAB 1402: Informatique 2

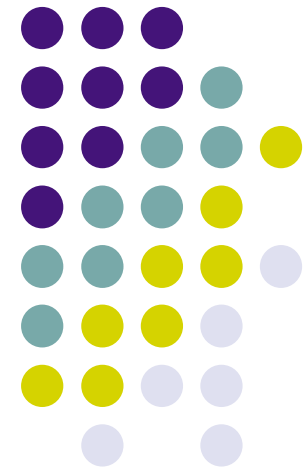
## Récursion sur les Entiers



Département d'Ingénierie Informatique, UCL

**Peter Van Roy**

[pvr@info.ucl.ac.be](mailto:pvr@info.ucl.ac.be)



# Ce qu'on va voir aujourd'hui



- Résumé du premier cours
- Récursion sur les entiers
- Introduction aux listes

# Lecture

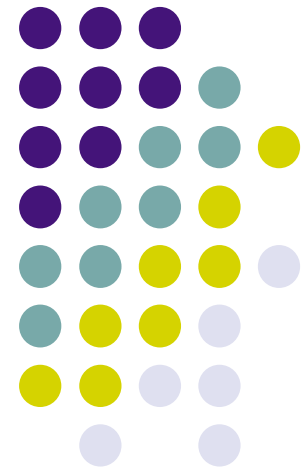
## pour le deuxième cours



- Chapitre 1 (sections 1.3-1.6)
  - Fonctions, listes, exactitude
- Chapitre 2 (section 2.1.1)
  - Syntaxe des langages
- Chapitre 2 (sections 2.3 et 2.4.1)
  - Le langage noyau du modèle déclaratif
  - Une introduction aux types de base
  - Procédures et références externes
- Chapitre 3 (sections 3.2, 3.3, 3.4.1)
  - Calcul itératif
  - Calcul récursif
  - Sauf section 3.3.2 qui sera vue plus tard avec la sémantique
  - Introduction aux listes et à la notation pour les types

# Résumé du premier cours

---





# Identificateurs et variables

- Souvenez-vous des deux mondes: **le monde visible** (du programmeur) et **le monde invisible** (à l'intérieur de l'ordinateur)
  - Un **identificateur** fait partie du monde visible: c'est un texte, fait pour le programmeur
  - Une **variable (en mémoire)** faite partie du monde invisible: c'est ce qu'utilise l'ordinateur pour faire ses calculs
  - Une variable n'est jamais vu directement par le programmeur, mais indirectement par l'intermédiaire d'un identificateur
    - Le rôle clé de **l'environnement**
    - Un même identificateur peut désigner des variables différentes à des endroits différents du programme
    - Des identificateurs différents peuvent désigner la même variable



# Une question de portée...

```
local P Q in
```

```
  proc {P} {Browse 100} end
```

```
  proc {Q} {P} end
```

```
local P in
```

```
  proc {P} {Browse 200} end
```

```
  {Q}
```

```
end
```

```
end
```

- Qu'est-ce qui est affiché par ce petit programme?
- **Utilisez la définition de portée lexicale:** chaque occurrence d'un identificateur correspond à une déclaration précise!



# Une question de portée...

```
local P Q in
  proc {P} {Browse 100} end
  proc {Q} {P} end
  local P in
    proc {P} {Browse 200} end
    {Q}
  end
end
```

- Quelle est la portée du **P** rouge?
- Alors, dans cette portée, où se trouve la définition de P?



# Procédures et portée lexicale

**local P Q in**

```
proc {P} {Browse 100} end
```

```
proc {Q} {P} end
```

**local P in**

```
proc {P} {Browse 200} end
```

```
{Q}
```

**end**

**end**

- “Une procédure ou une fonction se souvient toujours de l’endroit de sa naissance”
- La définition de Q utilise la **première** définition de P
- La seconde définition de P, à côté de l’appel de Q, n’est pas utilisée!



# Un autre raisonnement...

- Souvenez-vous de la définition du modèle déclaratif
  - Un programme qui marche aujourd'hui marchera demain
  - Un programme est un ensemble de fonctions
    - Chaque fonction ne change jamais son comportement, tout comme chaque variable ne change jamais son affectation
- Donc, l'appel de Q donnera **forcément** toujours le même résultat
  - Q doit toujours utiliser la même variable pour P!
  - Comment cela est-il implémenté?
  - Avec **l'environnement contextuel**: un environnement qui fait partie de la définition de la procédure Q et qui mémorise l'identificateur P lors de la définition de Q

# L'environnement contextuel (1)



- Voici la définition de la procédure Q:  
**proc** {Q} {P} **end**
- Quand Q est définie, un environnement  $E_c$  est créé qui contient P, et gardé avec la définition de Q
  - $E_c = \{P \rightarrow p\}$
- Alors, quand Q est appelée,  $E_c$  est utilisé pour trouver la variable  $p$ 
  - Nous sommes sûrs de toujours trouver la même variable, même s'il y a une autre définition de P à côté de l'appel de Q
- Les identificateurs dans  $E_c$  sont les identificateurs dans la définition de Q qui ne sont pas déclarés dans cette définition
  - Ceux-ci s'appellent les **identificateurs libres**

# L'environnement contextuel (2)



```
local Add2 Two in
```

```
  Two=2
```

```
fun {Add2 N} N+Two end
```

```
local X in
```

```
  X={Add2 10}
```

```
  {Browse X}
```

```
end
```

```
end
```

- Quel est l'environnement contextuel de la fonction Add2?
- Il faut regarder la définition de Add2: les occurrences d'identificateurs qui sont définies **en dehors** de Add2?
  - Ce sont les identificateurs libres de Add2
- $E_c = \dots ?$

# L'environnement contextuel (3)



```
local Add2 Two in
```

```
  Two=2
```

```
  fun {Add2 N} N+Two end
```

```
  local X Two in
```

```
    Two=9999
```

```
    X={Add2 10}
```

```
    {Browse X}
```

```
  end
```

```
end
```

- Quel est le résultat du Browse maintenant?
- C'est toujours 12!
- Pourquoi?

# Fonctions, procédures et le langage noyau



- Souvenez-vous: comment est-ce qu'on peut comprendre un langage riche, avec un grand nombre d'outils pour le programmeur?
  - On utilise un langage simple, le **langage noyau**
  - Le langage riche est traduit vers le langage noyau
- Exemple: dans notre langage noyau, il n'y a que des procédures, pas de fonctions
  - Une fonction est traduite vers une procédure avec un argument de plus
  - **fun** {Inc X} X+1 **end**  $\Rightarrow$  **proc** {Inc X Y} Y=X+1 **end**
  - $A=\{\text{Inc } 10\} \Rightarrow \{\text{Inc } 10 A\}$

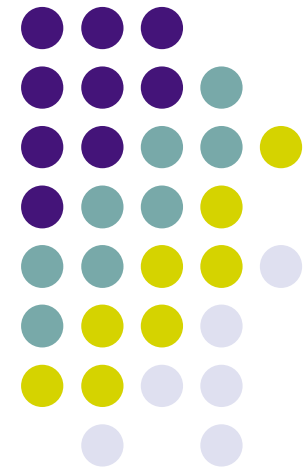
# Le langage noyau du modèle déclaratif (en partie)



- $\langle s \rangle ::=$ 
  - skip**
  - $\langle s \rangle_1 \langle s \rangle_2$
  - local**  $\langle x \rangle$  **in**  $\langle s \rangle$  **end**
  - $\langle x \rangle_1 = \langle x \rangle_2$
  - $\langle x \rangle = \langle v \rangle$
  - if**  $\langle x \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end**
  - $\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$
  - ...
- $\langle v \rangle ::=$   $\langle \text{number} \rangle$  |  $\langle \text{procedure} \rangle$  | ...

# Réursion sur les entiers

---



# Réursion



- Idée: résoudre un grand problème en utilisant des solutions aux problèmes plus petits
- Il faut savoir ranger les solutions selon la taille du problème qu'ils résolvent
  - Pour aller de Barbe 91 à Louvain-la-Neuve au restaurant Hard Rock Café à Stockholm
    - Découpe en de problèmes de base solubles: voyage de Louvain-la-Neuve à Zaventem (train), voyage Bruxelles-Stockholm (avion), voyage aéroport Stockholm-centre ville (train), voyage au Hard Rock Café (métro)
- Il faut savoir résoudre les problèmes de base directement! (train, métro, avion, voiture)



# Exemple: calcul de factorielle

**declare**

**fun** {Fact N}

**if** N==0 **then** 1

**else** N \* {Fact N-1} **end**

**end**

Grand problème: {Fact N}

Problème plus petit: {Fact N-1}

Problème de base: {Fact 0} = 1



# Récursion et invariants

- A chaque fonction récursive, on peut associer une formule mathématique, **l'invariant**
  - L'invariant exprime la relation entre les arguments et le résultat de la fonction
- Par exemple, pour la fonction Fact
  - On met d'abord Fact en langage noyau:  
**proc** {Fact N R} ... **end**  
(comme ça le résultat R devient visible)
  - L'invariant est simplement  **$n!=r$**
- Nous allons voir qu'il est très intéressant de raisonner avec les invariants
  - Pour **l'exactitude** et pour **l'efficacité**



# Une autre factorielle (1)

- En utilisant un **autre invariant** on peut faire une autre définition de factorielle:
  - $n! = i! * a$
  - On commence avec  $i=n$  et  $a=1$
  - On réduit  $i$  et on augmente  $a$ , tout en gardant vrai l'invariant
    - **Principe des vases communicants**: quand le niveau dans un vase descend, le niveau dans l'autre monte
  - Quand  $i=0$  c'est fini et le résultat est  $a$
- Exemple avec  $n=4$ :
  - $4! = 4! * 1$
  - $4! = 3! * 4$
  - $4! = 2! * 12$
  - $4! = 1! * 24$
  - $4! = 0! * 24$



## Une autre factorielle (2)

```
declare  
fun {Fact2 I A}  
  if I==0 then A  
  else {Fact2 I-1 I*A} end  
end
```

```
declare  
F={Fact2 4 1}  
{Browse F}
```

# Quelques informations sur l'invariant



- Voici encore une fois l'invariant qu'on a utilisé:
  - $n! = i! * a$
- Un **invariant** est une formule logique qui est vraie à chaque appel récursif (par exemple, {Fact2 I A}) pour les arguments de cet appel (ici, I et A)
- L'invariant contient à la fois des informations globales ( $n$ ) et locales (les arguments  $i$  et  $a$ )
  - Pour le programme, une information globale est une constante
- Si on traduit en langage noyau, le résultat R devient visible et on a **proc** {Fact2 I A R}
  - L'invariant complet est  $(n! = i! * a) \wedge (n! = r)$

# Comparaison de Fact et Fact2



- Quand on regarde l'appel récursif dans les deux cas, on voit une différence:
  - Dans Fact:  $N \times \{\text{Fact } N-1\}$
  - Dans Fact2:  $\{\text{Fact2 } I-1 \ I * A\}$
- Dans Fact, on fait la multiplication **après** l'appel récursif
  - On peut voir ceci plus clairement en traduisant Fact en **langage noyau**
- Dans Fact2, on fait la multiplication **avant** l'appel récursif
  - L'appel récursif ne revient pas
  - Exercice pour vous: mettre Fact2 en langage noyau pour vérifier que l'appel récursif est bien le dernier appel dans Fact2



# Fact en langage noyau

**declare**

**proc** {Fact N R}

**if** N==0 **then** R=1

**else**

**local** N1 R1 **in**

      N1=N-1

      {Fact N1 R1} % Ce n'est pas le dernier appel

      R=N\*R1 % On doit revenir! Pas de chance.

**end**

**end**

**end**



# Les accumulateurs

- Dans Fact2, on “**accumule**” le résultat petit à petit dans A
- L’argument A de Fact2 est donc appelé un **accumulateur**
  - Quand on utilise un invariant pour faire un programme, cela fait apparaître des accumulateurs
- Dans la programmation déclarative, on utilise souvent des invariants et des accumulateurs
  - Les invariants sont les mêmes qu’on utilise dans les boucles des langages impératifs comme Java
- Une fonction récursive qui utilise un accumulateur est comme une boucle en langage impératif!
  - **Une boucle en langage impératif = une fonction récursive avec accumulateur**



# L'importance des accumulateurs

- Quand on regarde l'appel récursif dans les deux cas, on voit une différence:
  - Dans Fact:  $N*\{\text{Fact } N-1\}$
  - Dans Fact2:  $\{\text{Fact2 } I-1 \ I*A\}$
- C'est une grande différence!
  - Pendant l'exécution, Fact doit garder en mémoire des informations sur **tous les appels récursifs**, jusqu'à la fin des appels récursifs
  - Fact2 ne doit garder en mémoire que **l'appel récursif actuellement en cours**, ce qui est une économie importante
- Pour l'efficacité, **l'appel récursif doit être la dernière instruction!**
  - Alors la taille de mémoire sera constante (comme une boucle)
  - C'est pourquoi les accumulateurs sont importants
  - (On rendra cette intuition plus exacte quand on verra la sémantique)

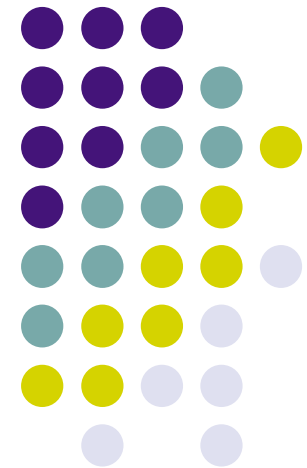


# La récursion terminale

- On appelle **récursion terminale** quand la dernière instruction est l'appel récursif
- Parce qu'on ne doit pas revenir de l'appel récursif, la taille de la pile est constante
  - L'exécution est aussi efficace qu'une boucle en langage impératif
- **C'est la règle principale à suivre dans le modèle déclaratif**
- **Attention:** certaines implémentations reviennent quand même de l'appel récursif, et ne gagnent donc rien de la récursion terminale
  - C'est le cas pour certaines implémentations de Java et de C++: il faut vérifier dans la documentation!
  - Tous les langages symboliques (Scheme, ML, Haskell, Oz, Prolog, ...) implémentent bien la récursion terminale

# Racine carrée avec la méthode de Newton

---



# La racine carrée avec la méthode itérative de Newton



- On va utiliser une méthode itérative, la méthode de Newton, pour calculer la racine carrée
- Cette méthode est basée sur l'observation que si  $g$  est une approximation de  $\sqrt{x}$ , alors la moyenne entre  $g$  et  $x/g$  est une meilleure approximation:
  - $g' = (g + x/g)/2$

# Pourquoi la méthode de Newton marche



- Pour vérifier que l'approximation améliorée est vraiment meilleure, calculons l'erreur  $e$ :  
$$e = g - \text{sqrt}(x)$$
- Alors:  
$$e' = g' - \text{sqrt}(x) = (g + x/g)/2 - \text{sqrt}(x) = e^2/2g$$
- Si on suppose que  $e^2/2g < e$  (l'erreur devient plus petite), on peut déduire la condition suivante:  
$$g + \text{sqrt}(x) > 0$$
- Cette condition est toujours vraie (parce que  $g > 0$ )
  - C'est donc toujours vrai que l'erreur diminue!



# Itération générique

- Nous allons utiliser un itérateur générique:

```
fun {Iterate Si}  
  if {IsDone Si} then Si  
  else local Sj in  
    Sj={Transform Si}  
    {Iterate Sj}  
  end end  
end
```

- Cet itérateur utilise un accumulateur Si
- Il faut remplir *IsDone* et *Transform*

# L'amélioration d'une approximation (*Transform*)



```
fun {Transform Guess}
  (Guess + X/Guess) / 2.0
end
```

- Attention: X n'est pas un argument de Transform!
- X est un “**identificateur libre**” dans *Transform* qui doit être défini dans le contexte de *Transform*
  - Quel est l'environnement contextuel de *Transform*?
- (Petite remarque: X, Guess et 2.0 sont tous des nombres en virgule flottante. Pour garder l'exactitude des entiers, il n'y a aucune conversion automatique avec les entiers.)



# La fin de l'itération (*IsDone*)

```
fun {IsDone Guess}  
  {Abs (X-Guess*Guess)}/X < 0.00001  
end
```

- De nouveau, X est un **identificateur libre** dans *IsDone*
- La fonction Abs fait partie des modules de base de Mozart
- L'erreur relative 0.00001 est “câblée” dans la routine; on peut en faire un paramètre (exercice!)
- Attention: X doit être différent de 0.0!



# Définition complète

```
fun {NewtonSqrt X}  
  fun {Transform Guess}  
    (Guess + X/Guess)/2.0  
  end  
  fun {IsDone Guess}  
    {Abs X-Guess*Guess}/X < 0.00001  
  end  
  fun {Iterate Guess}  
    if {IsDone Guess} then Guess  
    else {Iterate {Transform Guess}} end  
  end  
in  
  {Iterate 1.0}  
end
```



# Spécification de NewtonSqrt

- $S = \{\text{NewtonSqrt } X\}$  satisfait:
  - Les arguments  $S$  et  $X$  sont des nombres en virgule flottante; l'entrée  $X$  et la sortie  $S$  sont positifs ( $>0.0$ )
  - La relation entre eux est:  
 $|x - s*s| < 0.00001$
- Exercice: étendre la fonction pour satisfaire à une spécification où  $X=0.0$  est permis

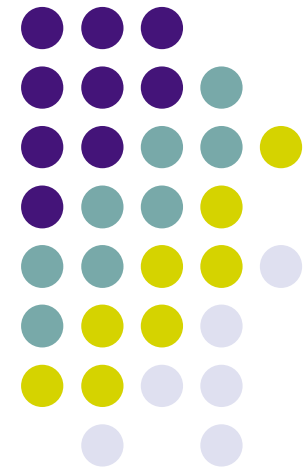


# Structure de NewtonSqrt

- Vous remarquez que Transform, IsDone et Iterate sont des fonctions locales à NewtonSqrt
  - Elles sont cachées de l'extérieur
- Transform et IsDone sont dans la portée de X, elles connaissent donc la valeur de X
- D'autres définitions sont possibles (voir le livre)!
  - Par exemple, vous pouvez mettre leurs définitions à l'extérieur de NewtonSqrt (exercice!)
  - Avec **local** ... **end**, vous pouvez quand même cacher ces définitions du reste du programme (comment?)

# Calcul récursif de la puissance

---



# Une récursion un peu plus compliquée: la puissance



- Nous allons définir une fonction {Pow X N} qui calcule  $X^N$  ( $N \geq 0$ ) avec une méthode efficace
- Définition mathématique naïve de  $x^n$ :  
$$x^0 = 1$$
$$x^n = x * (x^{n-1}) \text{ si } n > 0$$

- Cette définition donne la fonction suivante:

```
fun {Pow X N}  
  if N==0 then 1  
  else X*{Pow X N-1} end  
end
```

- Cette définition est très inefficace en espace et en temps! **Pourquoi?**



# Une autre définition de $X^N$

- Voici une autre définition de  $x^n$ :

$$x^0 = 1$$

$$x^{2n+1} = x * x^{2n}$$

$$x^{2n} = y^2 \text{ où } y=x^n \text{ (} n>0\text{)}$$

- Comme la première définition, nous pouvons programmer cette définition tout de suite!
- Les deux définitions sont aussi des **spécifications**
  - Ce sont des définitions **purement mathématiques**
  - La deuxième définition est plus élaborée que la première, mais c'est quand même une spécification



# Définition de {Pow X N}

```
fun {Pow X N}  
  if N==0 then 1  
  elseif N mod 2 == 1 then  
    X*{Pow X (N-1)}  
  else Y in  
    Y={Pow X (N div 2)}  
    Y*Y  
  end  
end
```



# Pow avec un accumulateur

- La définition précédente n'utilise pas d'accumulateur
- Est-ce qu'on peut faire une autre définition avec un accumulateur?
- Il faut d'abord un invariant!
  - Dans l'invariant, une partie "accumulera" le résultat et une autre partie tendra à disparaître
  - Qu'est-ce qui est "accumulé" dans Pow?



# Pow avec un accumulateur

- Voici un invariant:  
$$x^n = y^i * a$$
- Cet invariant peut être représenté par un triplet  
 $(y, i, a)$
- Initialement:  $(y, i, a) = (x, n, 1)$
- Il faut faire diminuer  $i$ , tout en gardant vrai l'invariant
- Il y a deux types d'itérations:
  - $(y, i, a)$  devient  $(y*y, i/2, a)$  (si  $i$  est pair)
  - $(y, i, a)$  devient  $(y, i-1, y*a)$  (si  $i$  est impair)
- Quand  $i=0$  le résultat est  $a$

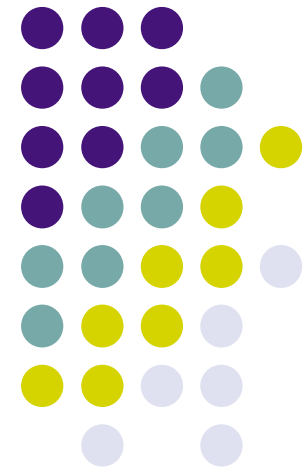
# Définition de {Pow X N} avec un accumulateur



```
fun {Pow2 X N}
  fun {PowLoop Y I A}
    if I==0 then A
    elseif I mod 2 == 0 then
      {PowLoop Y*Y (I div 2) A}
    else {PowLoop Y (I-1) Y*A} end
  end
in
  {PowLoop X N 1}
end
```

# Introduction aux listes

---





# Introduction aux listes

- Une liste est une **structure composée** avec une définition récursive:  
*Une liste est une liste vide ou un élément suivi par une autre liste*
- Voici une règle de grammaire en notation EBNF:

$$\langle \text{List } T \rangle ::= \text{nil} \mid T \mid \langle \text{List } T \rangle$$

- $\langle \text{List } T \rangle$  représente une liste d'éléments de type  $T$  et  $T$  représente un élément de type  $T$
- Attention à la différence entre  $\mid$  et  $\mid\prime$



# 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**



# Syntaxe pour les listes (1)

- Syntaxe simple (l'opérateur '|' au milieu)
  - nil, 5|nil, 5|6|nil, 5|6|7|nil
  - nil, 5|nil, 5|(6|nil), 5|(6|(7|nil))
- Sucre syntaxique (la plus courte)
  - nil, [5], [5 6], [5 6 7]



## Syntaxe pour les listes (2)

- 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)))



# Opérations sur les listes

- Extraire le premier élément  
L.1
- Extraire le reste de la liste  
L.2
- Comparaison  
L==nil  
L1==L2



# Longueur d'une liste

- La longueur d'une liste est le nombre d'éléments dans la liste
- On peut la calculer avec une fonction récursive:

```
fun {Longueur L}  
    if L==nil then 0  
    else 1+{Longueur L.2} end  
end
```

- Exercice: faire une version de Longueur avec accumulateur!

# Résumé



- Réursion sur les entiers
- Spécification d'une fonction
  - La définition **mathématique** de la fonction; parfois inductive
  - A partir d'une spécification on peut faire une implémentation en langage de programmation
- Invariant d'une fonction
  - Une formule logique qui est toujours vraie pour les arguments et le résultat à chaque appel récursif de la fonction
- Boucle
  - Quand **l'appel récursif est la dernière instruction (récursion terminale)**, la mémoire utilisée est constante. Dans ce cas, la fonction récursive est une **boucle**.
  - Attention: il existe des systèmes qui n'ont pas implémenté cette propriété (surtout parmi les implémentations des langages impératifs). Pour les systèmes qui ont ce défaut, la récursion est à utiliser avec modération.
- Accumulateur
  - Avec un accumulateur on est sûr d'avoir la récursion terminale
  - Un accumulateur est toujours lié à un invariant
- Liste et fonction récursive sur une liste