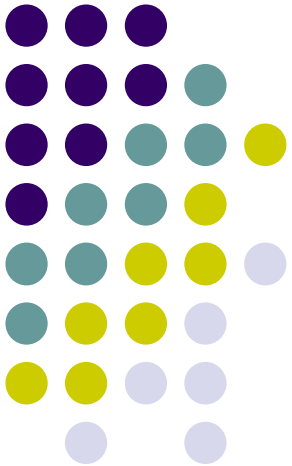
MI-GLSD-M1 -UEM213 : Programming languages paradigms

Chapitre III: paradigme fonctionnel

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Matière

- Cours basé sur le livre

  Concepts, Techniques, and Models of Computer Programming, MIT Press, 2004

A.HARICHE a.hariche@univ-dbkm.dz

Objectif du cours

Donner une introduction à la discipline de la programmation, en trois thèmes:

1. La programmation fonctionnelle (3 semaines)
   - Un programme est une fonction mathématique
   - La base de tous les autres modèles (“paradigmes”) de la programmation
2. La sémantique formelle des langages (1 semaine)
   - On ne peut pas maîtriser ce qu’on ne comprend pas
3. L’abstraction de données (2 semaines)
   - Partitionner un problème pour maîtriser la complexité
   - Les deux approches: objets et types abstraits

A.HARICHE a.hariche@univ-dbkm.dz

Il y a beaucoup de manières de programmer un ordinateur!

- Programmation déclarative
  - Programmation fonctionnelle ou programmation logique
- Programmation avec état
- Programmation orienté-objet
- Programmation concurrente
  - Par échange de messages ou par données partagées

A.HARICHE a.hariche@univ-dbkm.dz

Modèles de programmation (“paradigmes”)

- Mettre ensemble
  - Des types de données et leurs opérations
  - Un langage pour écrire des programmes
- Chaque modèle/paradigme permet d’autres techniques de programmation
  - Les paradigmes sont complémentaires
- Le terme “paradigme de programmation”
  - Très utilisé dans le monde commercial; attention au sens (buzzword)!

Langages de programmation

- Différents langages soutiennent différents modèles/paradigmes
  - Java: programmation orienté-objet
  - Haskell: programmation fonctionnelle
  - Erlang: programmation concurrente pour systèmes fiables
  - Prolog: programmation logique
  - …
- Nous voudrions étudier plusieurs paradigmes!
- Est-ce qu'on doit étudier plusieurs langages?
  - Nouvelle syntaxe…
  - Nouvelle sémantique…
  - Nouveau logiciel…

La solution pragmatique…

- Un seul langage de programmation
  - Qui soutient plusieurs modèles de programmation
  - Parfois appelé un langage “multi-paradigme”
- Notre choix est Oz
  - Soutient ce qu’on voit dans le cours, et plus encore
- L’accent sera mis sur
  - Les modèles de programmation!
  - Les techniques et les concepts!
  - **Pas** le langage en soi!

A.HARICHE a.hariche@univ-dbkm.dz

Comment présenter plusieurs modèles de programmation?

- Basé sur un langage noyau
  - Un langage simple
  - Un petit nombre de concepts **significatifs**
  - Buts: simple, minimaliste

- Langage plus riche au dessus du langage noyau
  - Exprimé en le traduisant vers le langage noyau
  - But: soutenir la programmation pratique

Approche incrémental

- Commencer par un langage noyau simple
  - Programmation déclarative
- Ajouter des concepts
  - Pour obtenir les autres modèles de programmation
  - Très peu de concepts!
  - Très peu à comprendre!

A.HARICHE a.hariche@univ-dbkm.dz

Les modèles que vous connaissez!

- Vous connaissez tous le langage Java, qui soutient
  - La programmation avec état
  - La programmation orienté-objet

- C’est clair que ces deux modèles sont importants!

A.HARICHE a.hariche@univ-dbkm.dz

Pourquoi les autres modèles?

- Un nouveau modèle qu'on verra dans ce cours
  - Programmation déclarative (fonctionnelle)
- D'autres modèles pas vu dans ce cours
  - Programmation concurrente (dataflow, échange de messages, données partagées)
  - Programmation logique (déterministe et nondéterministe)
  - Programmation par contraintes
  - …
- Est-ce que tous ces modèles sont importants?
  - Bien sûr!
  - Ils sont importants dans beaucoup de cas, par exemple pour les systèmes **complexes**, les **systèmes multi-agents**, les systèmes à **grande taille**, les systèmes à **haute disponibilité**, etc.
  - On reviendra sur ce point plusieurs fois dans le cours

A.HARICHE a.hariche@univ-dbkm.dz

Notre premier modèle

- La programmation **déclarative**
  - On peut la considérer comme la base de tous les autres modèles
- Approche
  - Introduction informelle aux concepts et techniques importants, avec exemples interactifs
  - Introduction au langage noyau
  - Sémantique formelle basée sur le langage noyau
  - Étude des techniques de programmation, surtout la récursion (sur entiers et sur listes)

A.HARICHE a.hariche@univ-dbkm.dz

La programmation déclarative

- L'idéal de la programmation déclarative
  - Dire uniquement **ce que** vous voulez calculez
  - Laissez l'ordinateur trouvez **comment** le calculer

- De façon plus pragmatique
  - Demandez plus de soutien de l'ordinateur
  - Libérez le programmeur d'une partie du travail

A.HARICHE a.hariche@univ-dbkm.dz

Propriétés du modèle déclaratif

- Un programme est une fonction au sens mathématique
  - Un calcul est l'évaluation d'une fonction sur des arguments qui sont des structures de données
- Très utilisé
  - Langages fonctionnels: LISP, Scheme, ML, Haskell, …
  - Langages logiques: Prolog, Mercury, …
  - Langages de représentation: XML, XSL, …
- Programmation "sans état"
  - Aucune mise à jour des structures de données!
  - Permet la simplicité

A.HARICHE a.hariche@univ-dbkm.dz

L'utilité du modèle déclaratif

- Propriété clé: "Un programme qui marche aujourd'hui, marchera demain"
  - Les fonctions ne changent pas de comportement, les variables ne changent pas d'affectation
- Un style de programmation qui est à encourager dans tous les langages
  - "Stateless server" dans un contexte client/server
  - "Stateless component"
- Pour comprendre ce style, nous utilisons le modèle déclaratif!
  - Tous les programmes dans ce modèle sont ipso facto déclaratif: une excellent manière de l'apprendre

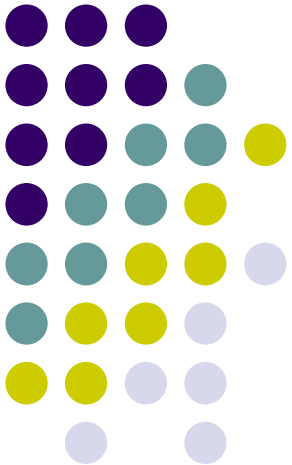
A.HARICHE a.hariche@univ-dbkm.dz

Introduction aux Concepts de Base

paradigme fonctionnel

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Un langage de programmation

- Réalise un modèle de programmation
- Peut décrire des programmes
  - Avec des **instructions**
  - Pour calculer avec des **valeurs**

- Regardons de plus près
  - instructions
  - valeurs

A.HARICHE a.hariche@univ-dbkm.dz

Système interactif

```
declare
X = 1234 * 5678
{Browse X}
```

- Sélectionner la région dans le buffer Emacs
- Donner la région sélectionnée au système
  - La région est compilée
  - La région compilée est exécutée
- Système interactif: à utiliser comme une calculatrice
- Essayez vous-même après ce cours!

A.HARICHE a.hariche@univ-dbkm.dz

Système interactif: Qu'est-ce qui se passe?

```
declare
X = 1234 * 5678
{Browse X}
```

- **Déclarer** ("créer") une variable **désignée** par X
- **Affecter** à la variable la valeur 7006652
  - Obtenu en faisant le calcul 1234*5678
- **Appliquer** la procédure Browse à la valeur désignée par X
  - Fait apparaître une fenêtre qui montre 7006652

A.HARICHE a.hariche@univ-dbkm.dz

Variables

- Des raccourcis pour des valeurs
- Peuvent être affectées une fois au plus
- Sont dynamiquement typées
  - En Java elles sont statiquement typées
- Attention: il y a **deux** concepts cachés ici!
  - L'identificateur: le nom que vous tapez sur le clavier

    une chaine de caractères qui commence avec une majuscule

    `Var, A, X123, OnlyIfFirstIsCapital`
  - La variable en mémoire: une partie de la mémoire du système

    initialement, une boîte vide

A.HARICHE a.hariche@univ-dbkm.dz

Déclaration d'une variable

```
declare
X = …
```

- `Instruction declare` ("statement")
  - Crée une nouvelle variable en mémoire
  - Fait le lien entre l'identificateur `X` et la variable en mémoire
- Troisième concept: l'environnement
  - Une fonction des identificateurs vers les variables
  - Fait la correspondance entre chaque identificateur et une variable (et donc sa valeur aussi)
  - Le même identificateur peut correspondre à différentes variables en différents endroits du programme!

A.HARICHE a.hariche@univ-dbkm.dz

Affectation

```
declare
X = 42
```

- L'affectation prend une variable en mémoire et la lie avec une valeur
- Dans le texte d'un programme, l'affectation est exprimée avec les identificateurs!
- Après l'affectation X=42, la variable qui correspond à l'identificateur X sera liée à 42

A.HARICHE a.hariche@univ-dbkm.dz

Affectation unique

- Une variable ne peut être affectée qu'à une seule valeur
  - On dit: **variable à affectation unique**
  - Pourquoi? Parce que nous sommes en modèle déclaratif!

- Affectation incompatible: lever une erreur

```
x = 43
```

- Affectation compatible: ne rien faire

```
x = 42
```


A.HARICHE a.hariche@univ-dbkm.dz

La redéclaration d'une variable (en fait: d'un identificateur!)

```
declare
x = 42
declare
x = 11
```

- Un identificateur peut être redéclaré
  - Ça marche parce qu'il s'agit des variables en mémoire différentes! Les deux occurrences de l'identificateur correspondent à des variables en mémoire différentes.
- L'environnement interactif ne gardera que la dernière variable
  - Ici, X correspondera à 11

A.HARICHE a.hariche@univ-dbkm.dz

La portée d'un identificateur

```
local
      X
in
      X = 42   {Browse X}
      local
            X
      in
            X = 11   {Browse
   X}
      end
      {Browse X}
end
```

- L'instruction
  local X **in** <stmt> **end**
  déclare X qui existera entre **in** et **end**
- La portée d'un identificateur est la partie d'un programme pour laquelle cet identificateur correspond à la même variable en mémoire.
- (Si la portée est déterminée par une inspection du code d'un programme, elle s'appelle portée lexicale ou portée statique.)
- Pourquoi il n'y a pas de conflit entre X=42 et X=11?
- Le troisième Browse affichera quoi?

A.HARICHE a.hariche@univ-dbkm.dz

Structures de données (valeurs)

- Structures de données simples
  - Entiers `42, ~1, 0`
    `Notez (!): “~” pour entier négatif`
  - Virgule flottante `1.01, 3.14`
  - Atomes (constantes) `foo, ‘Paul’, nil`
- Structures de données composées
  - Listes
  - Tuples, records (enregistrements)
- Dans ce cours on utilisera principalement les entiers et les listes

A.HARICHE a.hariche@univ-dbkm.dz

Fonctions

- Définir une fonction
  - Donner une instruction qui définit ce que doit faire la fonction
- Appliquer une fonction
  - Utiliser la fonction pour faire un calcul selon sa définition
  - On dit aussi: appeler une fonction

A.HARICHE a.hariche@univ-dbkm.dz

Notre première fonction

- Pour calculer la négation d'un entier
  - Prend un argument: l'entier
  - Rend une valeur: la négation de l'entier

- En notation mathématique:

$$\text{minus:} \begin{cases} \text{Integer} \rightarrow \text{Integer} \\ n \mapsto \sim n \end{cases}$$

A.HARICHE a.hariche@univ-dbkm.dz

La définition de la fonction Minus

```
declare
fun {Minus X}
   ~X
end
```

- L'identificateur Minus sera lié à la fonction
  - Déclarer une variable qui est liée à l'identificateur Minus
  - Affecter cette variable à la fonction en mémoire
- La portée de l'argument X est le corps de la fonction

A.HARICHE a.hariche@univ-dbkm.dz

L'application de la fonction Minus

```
declare
Y = {Minus ~42}
{Browse Y}
```

- Y est affecté à la valeur calculée par l'application de Minus à l'argument ~42

A.HARICHE a.hariche@univ-dbkm.dz

Syntaxe

- Définition d'une fonction

```
fun {Identificateur Arguments}
    corps de la fonction
end
```

- Application d'une fonction

```
X = {Identificateur Arguments}
```


A.HARICHE a.hariche@univ-dbkm.dz

Fonction de Maximum

- Calculer le maximum de deux entiers
  - Prend deux arguments: entiers
  - Rend une valeur: l'entier le plus grand

- En notation mathématique:

$$\max: \begin{cases} \text{Integer x Integer} \rightarrow \text{Integer} \\ n, m \mapsto n, \; n > m \\ \qquad\qquad\quad m, \; \text{otherwise} \end{cases}$$

A.HARICHE a.hariche@univ-dbkm.dz

Définition de la fonction Max

```
declare
fun {Max X Y}
   if X>Y then X
   else Y
   end
end
```

- Nouvelle instruction: conditionnel (if-then-else)

A.HARICHE a.hariche@univ-dbkm.dz

Application de la fonction Max

```
declare
X = {Max 42 11}
Y = {Max X 102}
{Browse Y}
```


A.HARICHE a.hariche@univ-dbkm.dz

Maintenant le Minimum

- Une possibilité: couper et coller
  - Répéter ce qu'on a fait pour Max
- Mieux: la composition de fonctions
  - Reutiliser ce qu'on a fait avant
  - C'est une bonne idée de reutiliser des fonctions compliquées
- Pour le minimum de deux entiers:

$$\min(n,m) = \sim\max(\sim n,\sim m)$$

A.HARICHE a.hariche@univ-dbkm.dz

Définition de la fonction Min

```
declare
fun {Min X Y}
   {Minus {Max {Minus X}
               {Minus Y}}}
end
```


A.HARICHE a.hariche@univ-dbkm.dz

Définition de la fonction Min (avec ~)

```
declare
fun {Min X Y}
   ~{Max ~X ~Y}
end
```


A.HARICHE a.hariche@univ-dbkm.dz

Définition inductive d'une fonction

- Fonction de factorielle $n!$
  - La définition inductive est

    $0! = 1$

    $n! = n * ((n-1)!)$
  - Programmer ceci en tant que fonction Fact

- Comment procéder?
  - Utiliser l'application **récursive** de Fact!

A.HARICHE a.hariche@univ-dbkm.dz

Définition de la fonction Fact

```
fun {Fact N}
   if N==0 then % Test d'égalité
      1
   else
      N * {Fact N-1} % Appel
  récursif
   end
end
```


A.HARICHE a.hariche@univ-dbkm.dz

Récursion

- Structure générale
  - Cas de base
  - Cas récursif

- Pour un nombre naturel *n*, souvent:
  - Cas de base: *n* est zéro
  - Cas récursif: *n* est différent de zéro
    n est plus grand que zéro

- Beaucoup plus: la semaine prochaine!

A.HARICHE a.hariche@univ-dbkm.dz

Une fonction est un cas particulier d'une procédure

- Le concept général est la **procédure**
- Une fonction est une procédure avec un argument en plus, qui est utilisée pour rendre le résultat
- Z={Max X Y} *(fonction Max)*

  correspond à:

  {Max X Y Z} *(procédure Max, avec argument de plus)*

A.HARICHE a.hariche@univ-dbkm.dz

Résumé

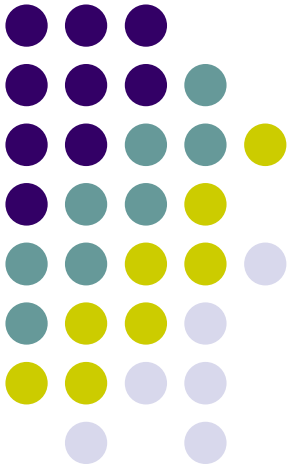
- Système interactif
- “Variables”
  - Déclaration
  - Affectation unique
  - Identificateur
  - Variable en mémoire
  - Environnement
  - Portée d’une variable
- Structures de données
  - Entiers
- Fonctions
  - Définition
  - Appel (application)
- Récursion

Résumé du premier cours

paradigme fonctionnel

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



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?

A.HARICHE a.hariche@univ-dbkm.dz

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”
- Donc, la définition de Q utilise la première définition de P
- Portée lexicale: dans la définition de Q, de quel P s’agit-il?
- La 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 toujours le même résultat
  - Comment cela est-il réalisé?
    Avec l'environnement contextuel

Identificateurs et variables

- Souvenez-vous des deux mondes: le programmeur et l'ordinateur
  - Un identificateur est un nom textuel, fait pour le programmeur
  - Une variable (en mêmoire) 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

A.HARICHE a.hariche@univ-dbkm.dz

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 de procédures, pas de fonctions
  - Une fonction est traduite vers une procédure avec un argument de plus
  - **fun** {Inc X} X+1 **end** *devient* **proc** {Inc X Y} Y=X+1 **end**
  - A={Inc 10} *devient* {Inc 10 A}

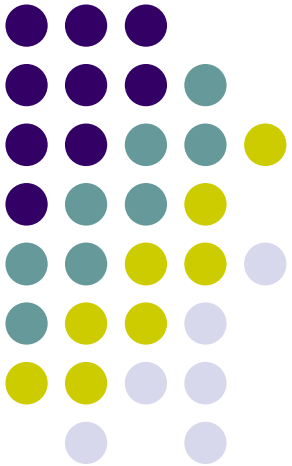
A.HARICHE a.hariche@univ-dbkm.dz

Récursion sur les entiers

paradigme fonctionnel

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Récursion

- 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 plus petits et 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 les plus petits directement! (train, métro, avion, voiture)

A.HARICHE a.hariche@univ-dbkm.dz

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}
Solution directe: {Fact 0} = 1
```


A.HARICHE a.hariche@univ-dbkm.dz

Une autre factorielle

- En utilisant un invariant:
  - *n! = i! * a*
  - On commence avec *i=n* et *a=1*
  - On réduit *i* et on augmente *a*, tout en gardant vrai l'invariant
  - Quand *i=0* c'est fini et le résultat c'est *a*
- Exemple avec *n=4*:
  - 4! = 4! * 1
  - 4! = 3! * 4
  - 4! = 2! * 12
  - 4! = 1! * 24
  - 4! = 0! * 24

Le programme

```
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}
```


A.HARICHE a.hariche@univ-dbkm.dz

L'invariant

- Voici l'invariant qu'on a utilisé:
  - $n! = i! * a$
- Un invariant est une formule logique qui est vrai à chaque appel récursif pour les arguments de cet appel
- L'invariant contient à la fois des informations globales (n) et locales (les arguments i et a)

A.HARICHE a.hariche@univ-dbkm.dz

Les accumulateurs

- Dans Fact2, on “accumule” le résultat petit à petit dans A
- L’argument A de Fact2 est donc appelé un accumulateur
- 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
- (Notez que Fact2 est aussi une fonction récursive: la programmation avec accumulateurs est un cas particulier de la programmation récursive)

A.HARICHE a.hariche@univ-dbkm.dz

Comparaison de Fact et Fact2

- Quand on regarde l'appel récursif dans les deux cas, on voit une différence:
  - Dans Fact: N*{Fact N-1}
  - Dans Fact2: {Fact2 I-1 I*A}
- Dans Fact, après l'appel récursif on doit revenir pour faire la multiplication avec N
- Dans Fact2, on ne revient pas après l'appel récursif

A.HARICHE a.hariche@univ-dbkm.dz

L'importance des accumulateurs

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

A.HARICHE a.hariche@univ-dbkm.dz

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:
  - $g' = (g + x/g)/2$

A.HARICHE a.hariche@univ-dbkm.dz

Pourquoi la méthode de Newton marche

- Pour vérifier que l'approximation améliorée est 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:

  $g + \text{sqrt}(x) > 0$

- C'est donc toujours vrai que l'erreur diminue!

A.HARICHE a.hariche@univ-dbkm.dz

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*

A.HARICHE a.hariche@univ-dbkm.dz

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*
- (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.)

A.HARICHE a.hariche@univ-dbkm.dz

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*
- 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!

A.HARICHE a.hariche@univ-dbkm.dz

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
```


A.HARICHE a.hariche@univ-dbkm.dz

Spécification de NewtonSqrt

- S={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

A.HARICHE a.hariche@univ-dbkm.dz

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 appropriée 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?)

A.HARICHE a.hariche@univ-dbkm.dz

Une récursion un peu plus compliquée: la puissance

- Nous allons définir une fonction {Pow X N} qui calcule X^N (N≥0) avec une méthode efficace
- D'abord, une méthode naïve:

```
fun {Pow X N}
    if N==0 then 1.0
    else X*{Pow X N-1} end
end
```

- Cette définition est très inefficace!

A.HARICHE a.hariche@univ-dbkm.dz

Définition inductive de X^N

- La définition inductive est

$$x^0 = 1$$

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

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

- Nous pouvons programmer cette définition tout de suite!
- Notez que cette définition inductive est aussi une spécification
  - C'est une définition purement mathématique
  - Plus élaborée, certes, que la spécification naïve, mais quand même une spécification

A.HARICHE a.hariche@univ-dbkm.dz

Définition de {Pow X N}

```
fun {Pow X N}
   if N==0 then 1.0
   elseif N mod 2 == 1 then
         X*{Pow X (N-1)}
   else Y in
         Y={Pow X (N div 2)}
         Y*Y
   end
end
```


A.HARICHE a.hariche@univ-dbkm.dz

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,1.0)*
- 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* alors le résultat est *a*

A.HARICHE a.hariche@univ-dbkm.dz

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 1.0}
end
```

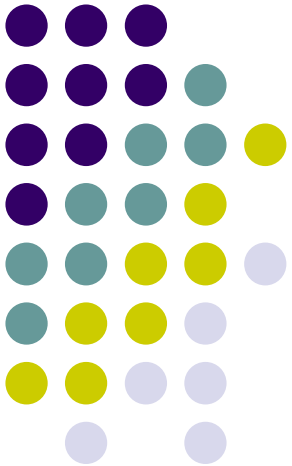

A.HARICHE a.hariche@univ-dbkm.dz

Introduction aux listes

paradigme fonctionnel

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Introduction aux listes

- Une liste est une structure composée avec une définition récursive:
  Une liste est où une liste vide où un élément suivi par une autre 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 '|'

A.HARICHE a.hariche@univ-dbkm.dz

Notation pour les types

- <Int> représente un entier; plus précisément l'ensemble de tous les entiers
- <List <Int>> représente l'ensemble de toutes les listes d'entiers
- T représente l'ensemble de tous les éléments de type T; nous disons que T est une variable de type

A.HARICHE a.hariche@univ-dbkm.dz

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)))

A.HARICHE a.hariche@univ-dbkm.dz

Opérations sur les listes

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

A.HARICHE a.hariche@univ-dbkm.dz

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
```


A.HARICHE a.hariche@univ-dbkm.dz

Résumé

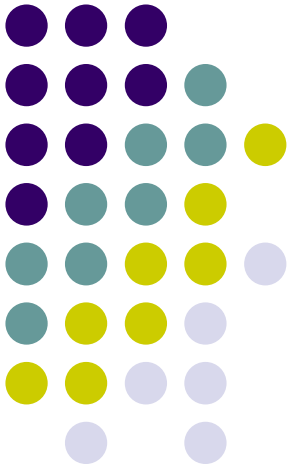
- Récursion sur les entiers
- Spécification
  - Définition mathématique de la fonction; parfois inductive
  - La réalisation d'une spécification est sa définition en un langage de programmation
- Invariant
  - Une formule logique qui est toujours vrai pour les arguments à chaque appel récursif
- Accumulateur
  - Quand l'appel récursif est le dernier appel, la mémoire utilisée est constante (à revoir avec la sémantique!)
  - Un accumulateur est toujours lié à un invariant
- Liste et fonction récursive sur une liste

Résumé du dernier cours

paradigme fonctionnel

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Principes

- Une fonction récursive est comme une boucle dans un langage impératif, mais plus générale
  - Attention: si vous avez deux boucles imbriquées, vous avez besoin de deux fonctions récursives!
  - Pourquoi plus générale? (tuyau: regardez mergesort)
- Si l'appel récursif est le dernier appel, alors la fonction récursive est exactement comme une boucle
  - Espace mémoire constante, temps proportionnel au nombre d'itérations
- La programmation avec accumulateurs est recommandée
  - Pour assurer que l'appel récursif soit le dernier appel
  - On peut utiliser un invariant pour dériver un accumulateur
  - Comment trouver un bon invariant?

A.HARICHE a.hariche@univ-dbkm.dz

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
  - Invariant: le triplet $(n-i,F_{i-1},F_i)$
    - “Invariant” parce que les trois parties du triplet doivent toujours être dans cette relation
  - Un pas de l’exécution: (k,a,b) *devient* (k-1,b,a+b)
  - Valeur initiale: (n-1,0,1)

A.HARICHE a.hariche@univ-dbkm.dz

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) *devient* (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}

A.HARICHE a.hariche@univ-dbkm.dz

Environnement contextuel d'une procédure (1)

- Quand on définit une procédure, on fait deux choses
  - Déclaration de l'identificateur
  - Création de la procédure en mémoire
- Quel est l'environnement contextuel de cette procédure?

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


A.HARICHE a.hariche@univ-dbkm.dz

Environnement contextuel d'une procédure (2)

- Pour bien distinguer la déclaration de l'identificateur et la création de la fonction en mémoire, on peut écrire:

```
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; une fonction anonyme)
- L'environnement contextuel contient donc {IsDone, Iterate, Transform}

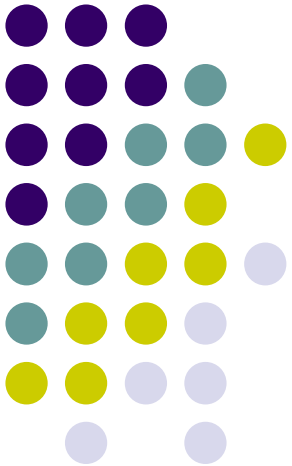
A.HARICHE a.hariche@univ-dbkm.dz

Récursion sur les listes

paradigme fonctionnel

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



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 des structures plus complexes et 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

A.HARICHE a.hariche@univ-dbkm.dz

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)

A.HARICHE a.hariche@univ-dbkm.dz

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 ‘|’

A.HARICHE a.hariche@univ-dbkm.dz

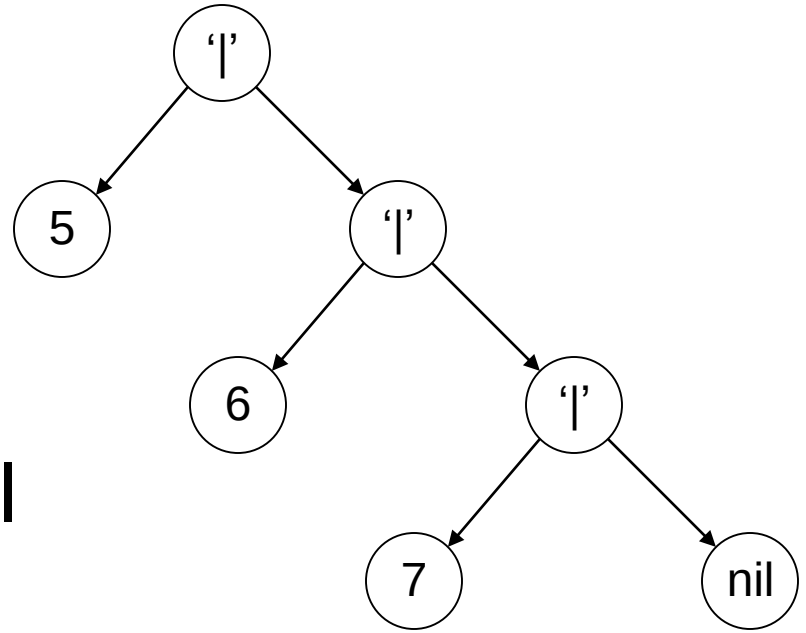
Notation pour les types

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

A.HARICHE a.hariche@univ-dbkm.dz

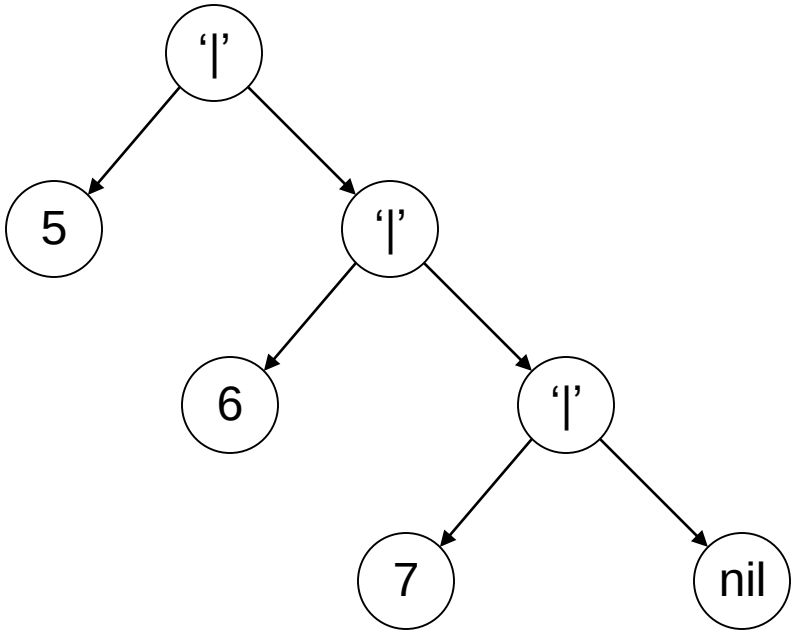
Syntaxe pour les listes (1)

- 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: pour le système, [5 6 7] et 5|6|7|nil sont identiques!



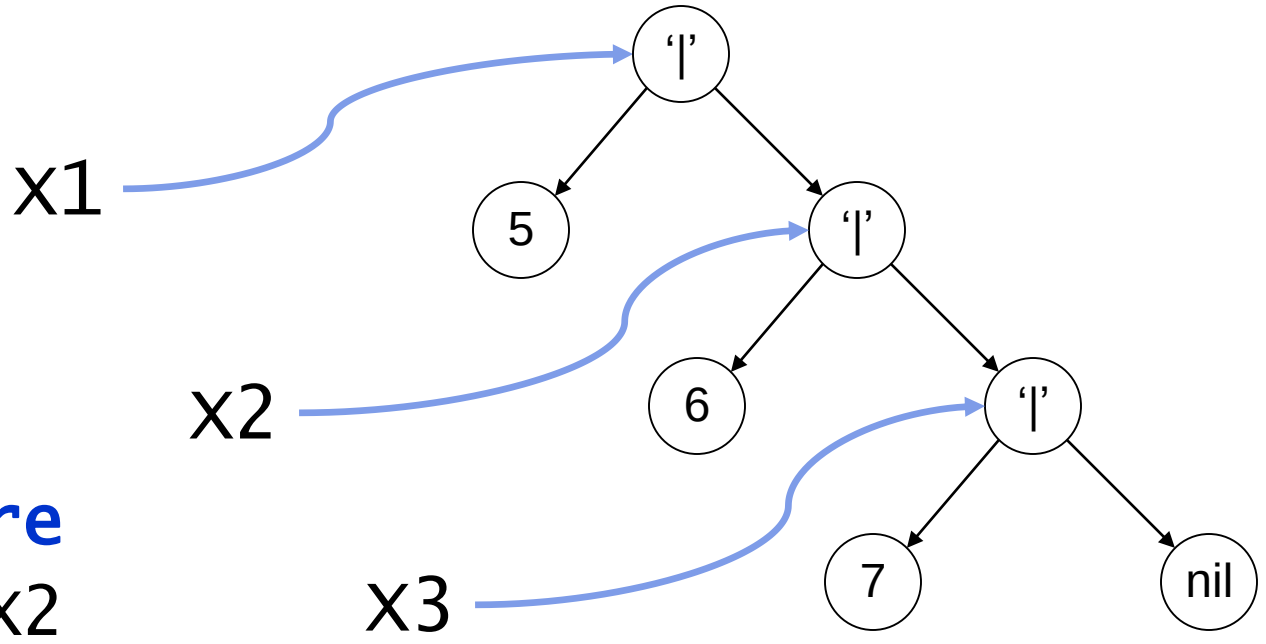
A.HARICHE a.hariche@univ-dbkm.dz

Au-delà des listes

- 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)))

A.HARICHE a.hariche@univ-dbkm.dz

Exemple d'une exécution




```
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)))
  ```

- La plus courte

  ```
  x1=[5 6 7]
  ```


A.HARICHE a.hariche@univ-dbkm.dz

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
```


A.HARICHE a.hariche@univ-dbkm.dz

La tête et la queue

- On peut définir des fonctions

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

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


A.HARICHE a.hariche@univ-dbkm.dz

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!

A.HARICHE a.hariche@univ-dbkm.dz

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}}
```


A.HARICHE a.hariche@univ-dbkm.dz

Somme des éléments d'une liste (méthode naïve)

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


A.HARICHE a.hariche@univ-dbkm.dz

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
```


A.HARICHE a.hariche@univ-dbkm.dz

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))), Sum2 fait ((((0+1)+2)+3)+4)

A.HARICHE a.hariche@univ-dbkm.dz

Méthode générale pour les fonctions 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*
  - Fait la correspondance entre une *liste* et une *forme* (“*pattern*”) et lie les identificateurs dans la forme

A.HARICHE a.hariche@univ-dbkm.dz

Sum avec pattern matching

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


A.HARICHE a.hariche@univ-dbkm.dz

Sum avec pattern matching

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

Une clause

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

A.HARICHE a.hariche@univ-dbkm.dz

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

A.HARICHE a.hariche@univ-dbkm.dz

Pattern matching

- 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
  - À ce moment, les identificateurs dans la forme sont affectées aux parties correspondentes 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
   [] X|Xr then 1+{Length Xr}
   end
end
```


A.HARICHE a.hariche@univ-dbkm.dz

Pattern matching en général

- Le pattern matching peut être utilisé pour beaucoup plus que les listes
- Toute valeur, y compris nombres, atomes, tuples, enregistrements
- Les formes peuvent être imbriquées
- Certains langages connaissent des formes encore plus générales (expressions régulières)
- Dans ce cours nous ne verrons que les listes

A.HARICHE a.hariche@univ-dbkm.dz

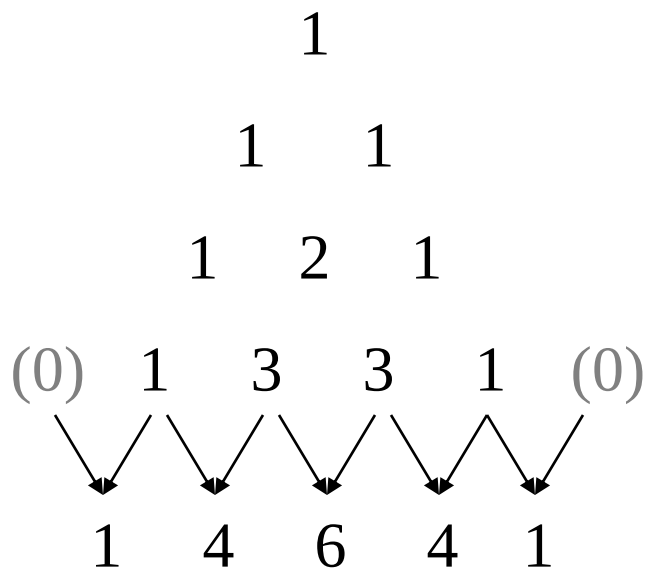
Résumé des listes

- 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

A.HARICHE a.hariche@univ-dbkm.dz

Fonctions sur les listes (1)

- Nous allons définir la fonction {Pascal N}
- Cette fonction prend un entier N et donne la Nième rangée du triangle de Pascal, représentée comme une liste d'entiers
- Une définition classique est que {Pascal N} est la liste des coefficients dans l'expansion de $(a+b)^n$

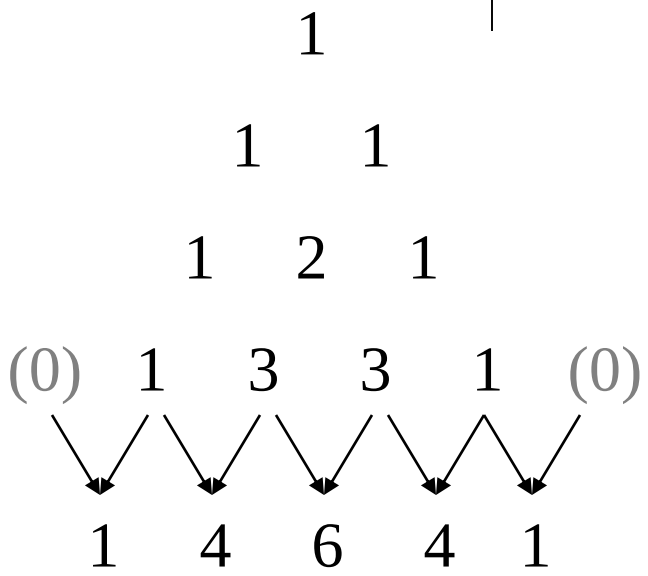


A.HARICHE a.hariche@univ-dbkm.dz

Fonctions sur les listes (2)

- Calculez la fonction {Pascal N}

1. Pour la rangée 1, cela donne [1]
2. Pour la rangée N, déplacez à gauche la rangée N-1 et déplacez à droite la rangée N-1
3. Alignez les deux rangées déplacées et additionnez-les élément par élément pour obtenir la rangée N

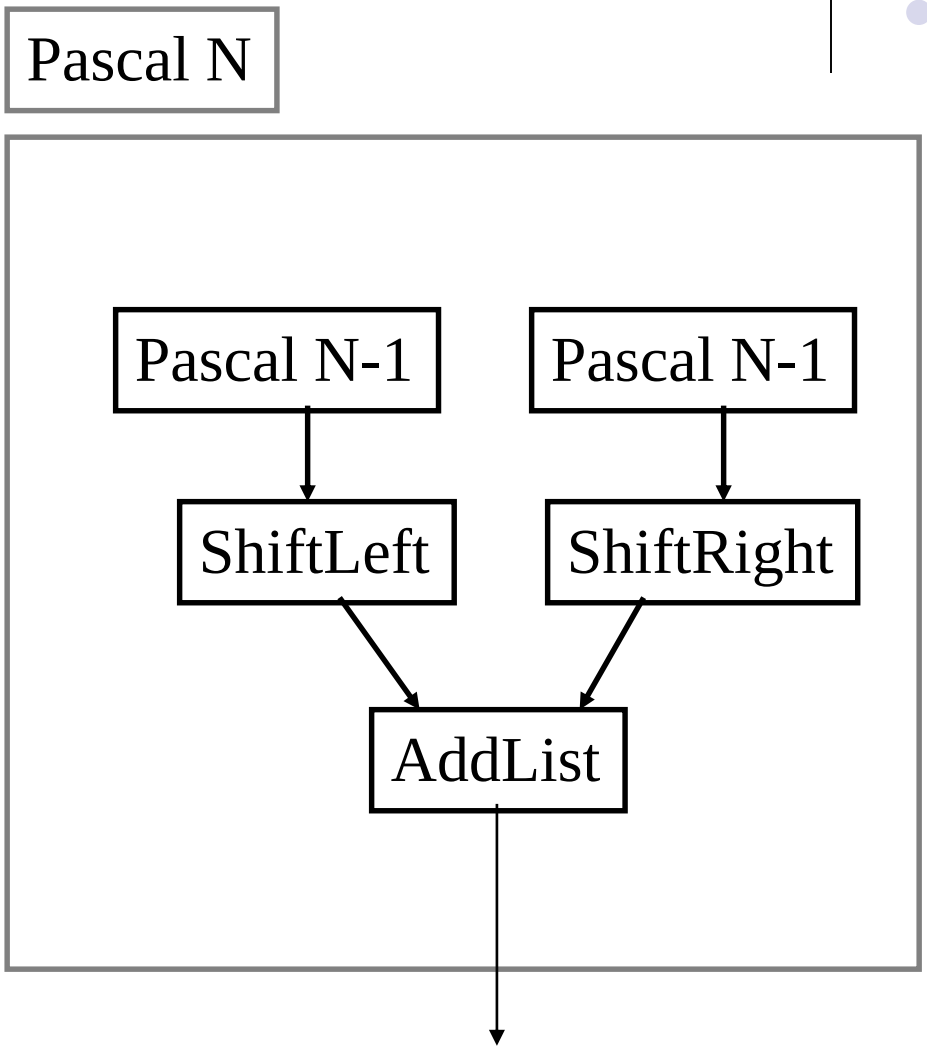


Déplacez à droite: [0 1 3 3 1]

Déplacez à gauche: [1 3 3 1 0]

A.HARICHE a.hariche@univ-dbkm.dz

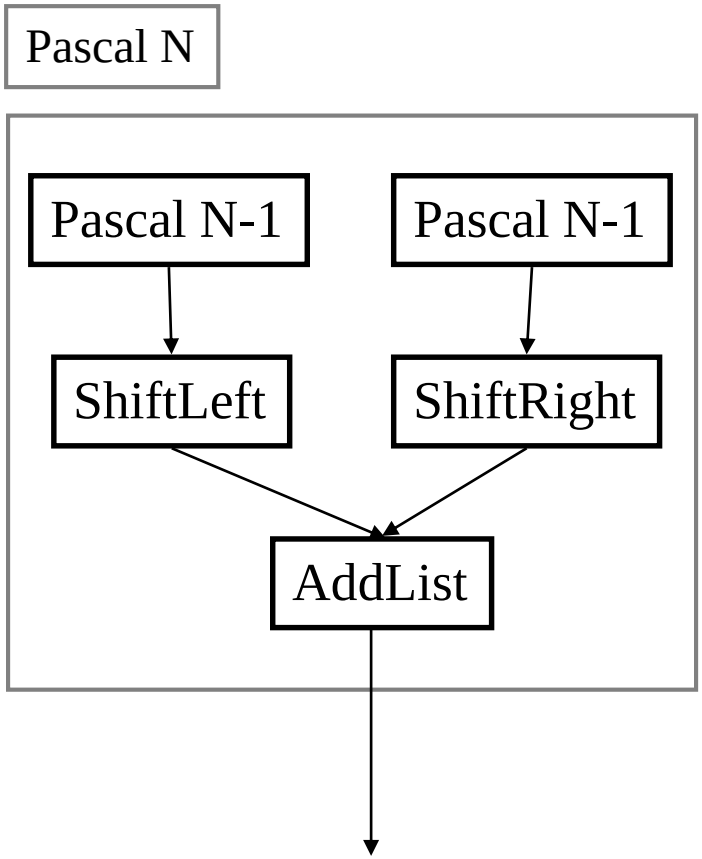
Fonctions sur les listes (3)



A.HARICHE a.hariche@univ-dbkm.dz

Fonctions sur les listes (4)

```
declare
fun {Pascal N}
   if N==1 then [1]
   else
      {AddList
        {ShiftLeft {Pascal
   N-1}}
        {ShiftRight
   {Pascal N-1}}}
   end
end
```



A.HARICHE a.hariche@univ-dbkm.dz

Fonctions sur les listes (5)

```
fun {ShiftLeft L}
   case L of H|T then
      H|{ShiftLeft T}
   else [0]
   end
end

fun {ShiftRight L}  0|L end
```


A.HARICHE a.hariche@univ-dbkm.dz

Fonctions sur les listes (6)

```
fun {AddList L1 L2}
   case L1 of H1|T1 then
      case L2 of H2|T2 then
         H1+H2|{AddList T1 T2}
      end
   else nil end
end
```


A.HARICHE a.hariche@univ-dbkm.dz

Développement descendant (”top-down”)

- Ce que nous avons fait avec Pascal est un exemple de développement descendant (”top-down”)
- D’abord, il faut comprendre comment résoudre le problème à la main
- On résoud le problème en le décomposant en de tâches plus simples
- La fonction principale (Pascal) est résolue en utilisant des fonctions auxiliaires (ShiftLeft, ShiftRight et AddList)
- On complète la solution en définissant les fonctions auxiliaires

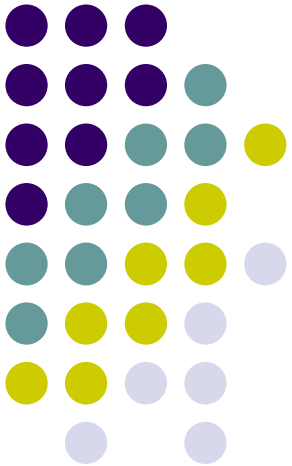
A.HARICHE a.hariche@univ-dbkm.dz

Introduction à la sémantique

paradigme fonctionnel

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Votre programme est-il correct?

- ”Un programme est correct quand il fait ce qu’on veut qu’il fasse”
- Comment se rassurer de cela?
- Il y a deux points de départ :
  - **La spécification du programme**: une définition du résultat du programme en termes de l’entrée (typiquement une fonction ou relation mathématique)
  - **La sémantique du langage**: un modèle précis des opérations du langage de programmation
- On doit prouver que la spécification est satisfaite par le programme, quand il exécute selon la sémantique du langage

A.HARICHE a.hariche@univ-dbkm.dz

Induction mathématique

- Pour les programmes récursifs, la preuve utilisera l'induction mathématique
  - Un programme récursif est basé sur un ensemble ordonné, comme les entiers et les listes
  - On montre d'abord l'exactitude du programme pour les cas de base
  - Ensuite, on montre que si le programme est correct pour un cas donné, alors il est correct pour le cas suivant
- Pour les entiers, le cas de base est souvent *0* ou *1*, et pour un entier *n* le cas suivant est *n+1*
- Pour les listes, le cas de base est nil ou une liste avec un ou plusieurs éléments, et pour une liste T le cas suivant est H|T

A.HARICHE a.hariche@univ-dbkm.dz

Exemple: exactitude de la factorielle

- La spécification de {Fact N} (purement mathématique)

  $0! = 1$

  $n! = n * ((n-1)!)$ si $n>0$

- Le programme (en langage de programmation)

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

- Où est la sémantique du langage?
  - On la verra la semaine prochaine!
  - Aujourd'hui le raisonnement sera intuitif

A.HARICHE a.hariche@univ-dbkm.dz

Raisonnement pour la factorielle

- Il faut démontrer que l'exécution de {Fact N} donne *n!* pour tout *n*
- Cas de base: *n=0*
  - La spécification montre *0!=1*
  - L'exécution de {Fact 0} avec la sémantique montre {Fact 0}=*1*
- Cas inductif: *(n-1)* $\rightarrow$ *n*
  - L'exécution de {Fact N}, selon la sémantique, montre que {Fact N} = N*{Fact N-1}
  - Avec l'hypothèse de l'induction, on sait que {Fact N-1}*=(n-1)!*
  - Si la multiplication est exacte, on sait donc {Fact N}=N**((n-1)!)*
  - Selon la définition mathématique de la factorielle, on peut déduire {Fact N}=*n!*
- Pour finir la preuve, il faut la sémantique du langage!

A.HARICHE a.hariche@univ-dbkm.dz

Machine abstraite

- Comment peut-on définir la sémantique d'un langage de programmation?
- Une manière simple et puissante est la machine abstraite
  - Une construction mathématique qui modélise l'exécution
- Nous allons définir une machine abstraite pour notre langage
  - Cette machine est assez générale; elle peut servir pour presque tous les langages de programmation
  - Plus tard dans le cours, nous verrons par exemple comment définir des objets et des classes
- Avec la machine abstraite, on peut répondre à beaucoup de questions sur l'exécution
  - On peut prouver l'exactitude des programmes ou comprendre l'exécution des programmes compliqués ou calculer le temps d'exécution d'un programme

A.HARICHE a.hariche@univ-dbkm.dz

Concepts de la machine abstraite

- Mémoire à affectation unique $\sigma = \{x_1=10, x_2, x_3=20\}$
  - Variables et leurs valeurs
- Environnement $E = \{X \rightarrow x, Y \rightarrow y\}$
  - Lien entre identificateurs et variables en mémoire
- Instruction sémantique $(<s>,E)$
  - Une instruction avec son environnement
- Pile sémantique $ST = [(<s_1>,E_1), \ldots, (<s_n>,E_n)]$
  - Une pile d'instructions sémantiques
- Exécution $(ST_1,\sigma_1) \rightarrow (ST_2,\sigma_2) \rightarrow (ST_3,\sigma_3) \rightarrow \ldots$
  - Une séquence d'états d'exécution (pile + mémoire)

A.HARICHE a.hariche@univ-dbkm.dz

Environnement

- Environnement E
  - Correspondance entre identificateurs et variables
  - Un ensemble de paires $X \rightarrow x$
    - Identificateur X, variable en mémoire x
- Exemple d'un environnement
  - $E=\{X \rightarrow x, Y \rightarrow y\}$
  - $E(X)=x$
  - $E(Y)=y$

A.HARICHE a.hariche@univ-dbkm.dz

L'environnement pendant l'exécution

- Prenons une instruction:

  (E_0) **local** X Y **in** (E_1)
  X=2 Y=3
  local X **in** (E_2)
  X=Y*Y
  {Browse X}
  end (E_3)
  end

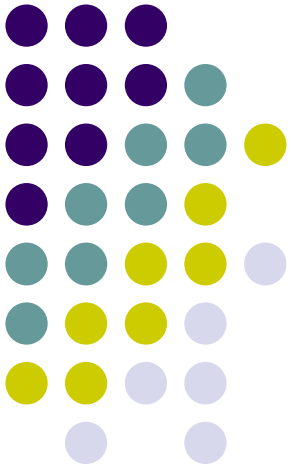
- E_0={Browse $\rightarrow$ *b*}
  E_1={X $\rightarrow$ x_1, Y $\rightarrow$ *y*, Browse $\rightarrow$ *b*}
  E_2={X $\rightarrow$ x_2, Y $\rightarrow$ *y*, Browse $\rightarrow$ *b*}
  E_3=E_1

Résumé

paradigme fonctionnel

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Résumé

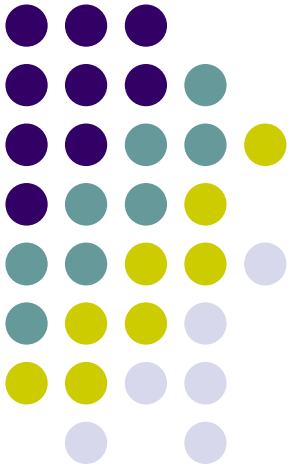
- Listes
  - Récursion sur les listes
  - Pattern matching
- Fonctions sur les listes
- Développement descendant (top-down)
- L'exactitude des programmes
  - Spécification, sémantique du langage, programme
  - Induction mathématique pour un programme récursif
- Machine abstraite
  - Construction mathématique qui modélise l'exécution
  - La semaine prochaine on verra comment elle marche

Résumé du dernier cours

paradigme fonctionnel

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Récursion avec les listes (1)

- Regardons comment on peut écrire une fonction récursive qui fait la concaténation de deux listes
- Voici une définition simple:

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

- L'appel récursif est-il le dernier?
- On dirait non, mais les apparences peuvent tromper
  - Traduisons en langage noyau!

A.HARICHE a.hariche@univ-dbkm.dz

Récursion avec les listes (2)

- Pour faire apparaître l'argument de sortie, on écrit la fonction en tant que procédure, comme par exemple:

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

- Ici l'appel récursif n'est pas le dernier appel
- Mais il y a une autre manière d'écrire la procédure!

A.HARICHE a.hariche@univ-dbkm.dz

Récursion avec les listes (3)

- Le compilateur Oz écrit la procédure autrement:

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

- On créé d'abord le résultat dans Zs et ensuite on passe Zr (une variable libre) à l'appel récursif
- C'est parce qu'on peut construire X|Zr avec une variable libre dedans

A.HARICHE a.hariche@univ-dbkm.dz

Les leçons de cet exemple

- Qu'est-ce que cet exemple nous montre?
- Deux choses:
  1. Pour calculez avec les listes, la fonction naïve est souvent efficace
  2. Pour vraiment comprendre ce que fait un programme, il faut le langage noyau

A.HARICHE a.hariche@univ-dbkm.dz

Pour les listes, la fonction naïve est souvent efficace

- La fonction naïve est efficace
  - On n'a pas besoin d'utiliser un accumulateur
  - Parce qu'on peut construire une liste (Zs=X|Xr) avec une variable libre dedans (Xr)
- De toute façon, l'appel récursif doit être le dernier pour que la fonction soit une boucle
  - Avec les listes, on a d'avantage de possibilités de faire cela qu'avec les entiers
  - Avec les entiers, pour faire un calcul (A=B*C) il faut connaître les arguments, ce qui n'est pas vrai pour les listes

A.HARICHE a.hariche@univ-dbkm.dz

Exercice!

- Ecrivez une fonction Append qui fait la concaténation de deux listes, avec les deux propriétés suivantes:
  - La fonction est aussi efficace qu'une boucle (tous les appels récursifs sont les derniers)
  - La fonction n'utilise pas de variables libres
- C'est plus compliqué qu'un Append naïve
  - Ceci montre l'importance des variables libres

A.HARICHE a.hariche@univ-dbkm.dz

Il faut le langage noyau

- Pour vraiment comprendre ce que fait un programme, il faut le langage noyau
  - La syntaxe d'un langage pratique peut cacher des choses!
    - L'intuition sur le "sens" de la syntaxe peut être trompeuse
    - Plusieurs sens sont souvent possibles
  - Au contraire, le langage noyau ne cache rien
- Pour comprendre une exécution, il y a donc deux étapes:
  1. D'abord, traduire en langage noyau
  2. Ensuite, faire l'exécution avec la sémantique
- Le langage noyau fait apparaître tout
  - Y compris les identificateurs "invisibles" comme l'argument Zs et le résultat intermédiaire Zr

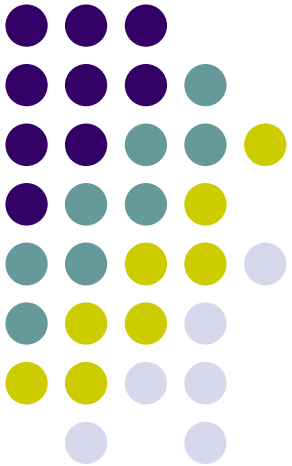
A.HARICHE a.hariche@univ-dbkm.dz

La sémantique du modèle déclaratif

paradigme fonctionnel

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Syntaxe du langage noyau

- Notation EBNF; <s> désigne une instruction

<s> ::= **skip**
| <x>$_1$=<x>$_2$
| <x>=<v>
| **local** <x> **in** <s> **end**
| **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
| { <x> <x>$_1$ … <x>$_n$}
| **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**

<v> ::= …
<p> ::= …

A.HARICHE a.hariche@univ-dbkm.dz

Syntaxe des valeurs

- Notation EBNF; <v> désigne une valeur, <p> désigne une forme (pattern)

  <v> ::= <list> | <number> | <procedure>
  <list>, <p> ::= nil | $<x>_1$ '|' $<x>_2$
  <number> ::= <int> | <float>
  <procedure> ::= **proc** {$ $<x>_1$ … $<x>_n$} <s> **end**

- Ceci est une partie de la syntaxe qui suffit pour ce cours; la syntaxe complète est donnée dans le livre

A.HARICHE a.hariche@univ-dbkm.dz

Les bases de l'exécution

- Pour exécuter une instruction, il faut:
  - Un environnement **E** pour faire le lien avec la mémoire
  - Une mémoire σ qui contient les variables et les valeurs
- Instruction sémantique **(<s>, E)**
  - Remarquez: chaque instruction a son environnement!
    - Pourquoi?
- Etat d'exécution **(*ST*, σ)**
  - Une pile **ST** d'instructions sémantiques avec une mémoire σ
  - Remarquez: il y a la même mémoire σ qui est partagée par toutes les instructions!
    - Pourquoi?

A.HARICHE a.hariche@univ-dbkm.dz

Début de l'exécution

- Etat initial

  ([(<s>,∅)], ∅)

  - Instruction **<s>** avec environnement vide ∅ (pas d'identificateurs libres)
  - Mémoire vide ∅ (pas encore de variables)
- A chaque pas
  - Enlevez l'instruction au sommet de la pile
  - Exécutez l'instruction
- Quand la pile est vide, l'exécution s'arrête

A.HARICHE a.hariche@univ-dbkm.dz

Exemple d'une exécution

```
local X in
   local B in
      B=true
      if B then X=1 else skip end
   end
end
```


A.HARICHE a.hariche@univ-dbkm.dz

Exemple: l'état initial

```
([(local X in
      local B in
         B=true
         if B then X=1 else skip end
      end
   end, ∅)],
 ∅)
```

- Commençons avec une mémoire vide et un environnement vide

A.HARICHE a.hariche@univ-dbkm.dz

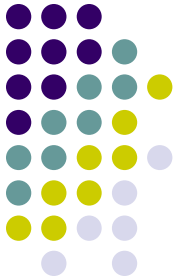
Exemple: l'instruction "local"

```
([[(local B in
       B=true
       if B then X=1 else skip end
   end,
   {X → x})],
 {x})
```

- Créez la nouvelle variable x dans la mémoire
- Continuez avec le nouvel environnement

A.HARICHE a.hariche@univ-dbkm.dz

Exemple: encore un “local”



```
([( (B=true
        if B then X=1 else skip end)
  ,
    {B → b, X → x})],
 {b,x})
```

- Créez la nouvelle variable *b* dans la mémoire
- Continuez avec le nouvel environnement

Exemple: composition séquentielle

$$([(\texttt{B=true},\ \{\texttt{B} \rightarrow b, \texttt{X} \rightarrow x\}),$$
$$(\texttt{if B then X=1}$$
$$\texttt{else skip end},\ \{\texttt{B} \rightarrow b, \texttt{X} \rightarrow x\})],$$
$$\{b,x\})$$

- L'instruction composée devient deux instructions
- La pile contient maintenant deux instructions sémantiques

A.HARICHE a.hariche@univ-dbkm.dz

Exemple: affectation de B

([(if B then X=1
else skip end, {B → *b*, X → *x*})],
{*b*=true, *x*})

- Affectez *b* à true

A.HARICHE a.hariche@univ-dbkm.dz

Exemple: instruction “if”

([(X=1, {B → *b*, X → *x*})],
{*b*=true, *x*})

- Testez la valeur de B
- Continuez avec l’instruction après le **then**

A.HARICHE a.hariche@univ-dbkm.dz

Exemple: affectation de X

([],
 {*b*=true, *x*=1})

- Affectez *x* à 1
- L'exécution termine parce que la pile est vide

A.HARICHE a.hariche@univ-dbkm.dz

Calculs avec l'environnement

- Pendant l'exécution, on fait deux sortes de calculs avec les environnements
- Adjonction: $E_2=E_1+\{X \rightarrow y\}$
  - Ajouter un lien dans un environnement
  - Pour **local** <x> **in** <s> **end**
- Restriction: $CE=E_{|\{B\}}$
  - Enlever des liens d'un environnement
  - Pour le calcul d'un environnement contextuel

A.HARICHE a.hariche@univ-dbkm.dz

L'adjonction

- Pour une instruction **local**

```
local X in (E1)
    X=1
    local X in (E2)
        X=2
        {Browse X}
    end
end
```

- E_1={Browse $\rightarrow b$, X $\rightarrow x$}
- E_2=E_1+{X $\rightarrow y$}
- E_2={Browse $\rightarrow b$, X $\rightarrow y$}

A.HARICHE a.hariche@univ-dbkm.dz

La restriction

- Pour une définition de procédure

```
local A B C in
    A=1 B=2 C=3 (E)
    fun {AddB X}
        X+B
    end
end
```

- E={A → *a*, B → *b*, C → *c*} σ={*a*=1,*b*=2,*c*=3}
- $CE=E_{|\{B\}}=\{B \rightarrow b\}$

A.HARICHE a.hariche@univ-dbkm.dz

Sémantique des instructions

- Pour chaque instruction du langage noyau, il faut préciser ce qu'elle fait avec l'état d'exécution
- Chaque instruction prend un état d'exécution et donne un autre état d'exécution
  - Etat d'exécution = pile sémantique + mémoire
- Nous allons regarder quelques instructions
  - **skip**
  - $\langle s \rangle_1$ $\langle s \rangle_2$ (composition séquentielle)
  - **local** <x> **in** <s> **end**
- Vous pouvez regarder les autres pendant les travaux pratiques

A.HARICHE a.hariche@univ-dbkm.dz

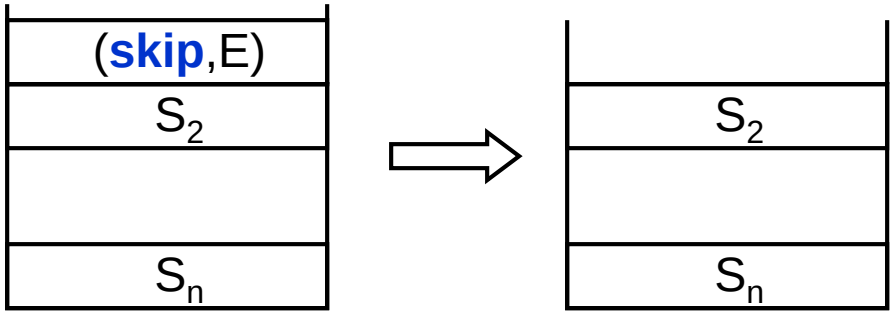
Rappel: les instructions du langage noyau

- Les instructions du langage noyau sont:
  - **skip**
  - <s>$_1$ <s>$_2$ (composition séquentielle)
  - **local** <x> **in** <s> **end**
  - <x>=<v> (affectation)
  - **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end** (conditionnel)
  - Définition de procédure
  - Appel de procédure
  - **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end** (pattern matching)

A.HARICHE a.hariche@univ-dbkm.dz

skip

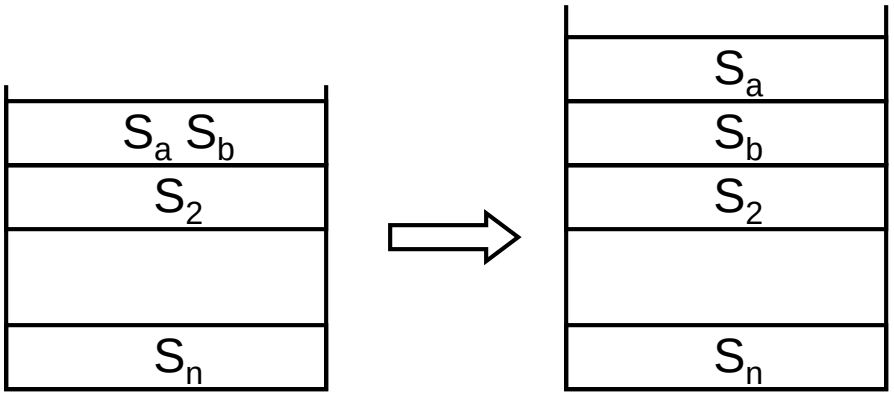
- L'instruction la plus simple
- Elle ne fait "rien"!
- Etat de l'entrée: ([(**skip**,E), S_2, …, S_n], σ)
- Etat de la sortie: ([S_2, …, S_n], σ)
- C'est tout!



A.HARICHE a.hariche@univ-dbkm.dz

$\langle s \rangle_1 \langle s \rangle_2$ (composition séquentielle)

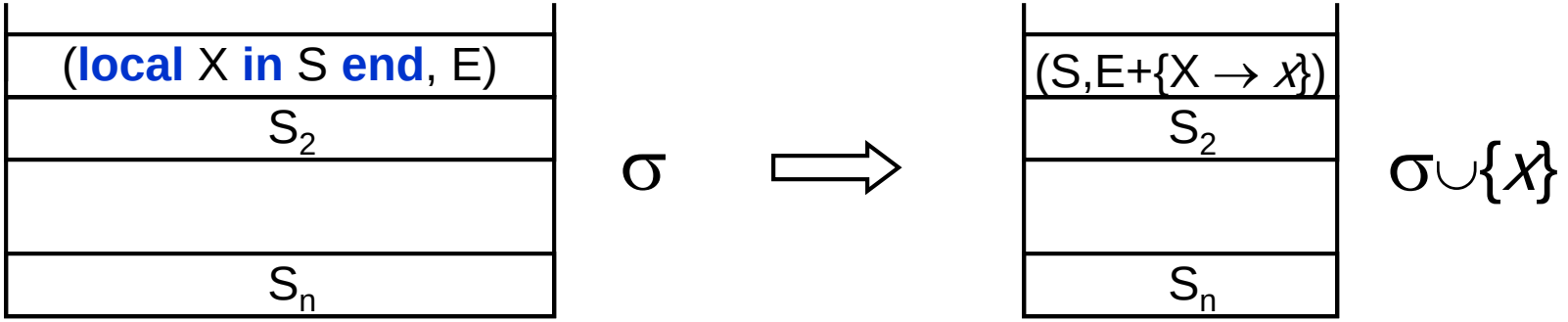
- Presque aussi simple
- L'instruction enlève le sommet de la pile et en ajoute deux nouveaux éléments



A.HARICHE a.hariche@univ-dbkm.dz

local <x> in <s> end

- Créez la nouvelle variable *x*
- Ajoutez la correspondance {X → *x*} à l'environnement E (l'adjonction)



A.HARICHE a.hariche@univ-dbkm.dz

Les autres instructions

- Elles ne sont pas vraiment compliquées
- A vous de regarder tranquillement chez vous
- <x>=<v> (l'affectation)
  - Attention: quand <v> est une procédure, il faut créer l'environnement contextuel!
- **if** <x> **then** $<s>_1$ **else** $<s>_2$ **end** (le conditionnel)
  - Attention: si <x> n'a pas encore de valeur, l'instruction attend ("bloque") jusqu'à ce que <x> est true ou false
  - La condition d'activation: "<x> est lié à une valeur"
- **case** <x> **of** <p> **then** $<s>_1$ **else** $<s>_2$ **end**
  - Attention: les case plus compliqués sont construits à partir de ce case simple

A.HARICHE a.hariche@univ-dbkm.dz

La sémantique des procédures

- Définition de procédure
  - Il faut créer l'environnement contextuel
- Appel de procédure
  - Il faut faire le lien entre les arguments formels et les arguments actuels
  - Il faut utiliser l'environnement contextuel

A.HARICHE a.hariche@univ-dbkm.dz

L’environnement contextuel

```
local Z in
   Z=1
   proc {P X Y} Y=X+Z end
end
```

- Les identificateurs libres (ici, Z) prennent des valeurs externes à la procédure
- Ils sont dans l’environnement contextuel, qui doit faire partie de la procédure!

Définition d'une procédure

- L'instruction sémantique de la définition est
  (<x>=**proc** {$ $\langle x \rangle_1 \ldots \langle x \rangle_n$} <s> **end**, E)
- Arguments formels:
  $\langle x \rangle_1, \ldots, \langle x \rangle_n$
- Identificateurs libres dans <s>:
  $\langle z \rangle_1, \ldots, \langle z \rangle_k$
- Environnement contextuel:
  $CE = E_{|\langle z \rangle 1, \ldots, \langle z \rangle k}$ (restriction de E)
- Valeur en mémoire:
  x=(**proc** {$ $\langle x \rangle_1 \ldots \langle x \rangle_n$} <s> **end**, CE)

A.HARICHE a.hariche@univ-dbkm.dz

Exemple avec procédure

```
local P in local Y in local Z in
   Z=1
   proc {P X} Y=X end
   {P Z}
end end end
```


A.HARICHE a.hariche@univ-dbkm.dz

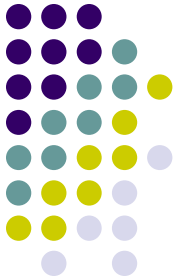
Exemple avec procédure

```
local P Y Z in
   Z=1
   proc {P X} Y=X end
   {P Z}
end
```

- Nous allons tricher un peu: nous allons faire toutes les déclarations en même temps
  - Ce genre de “tricherie” est très utile

A.HARICHE a.hariche@univ-dbkm.dz

Petit exercice: un “local” plus pratique



- On peut définir une version de **local** qui sait faire plusieurs déclarations à la fois
- Il faut modifier la sémantique
- Attention aux détails: `local x x in … end`

A.HARICHE a.hariche@univ-dbkm.dz

Exemple avec procédure

```
([(local P Y Z in
      Z=1
      proc {P X} Y=X end
      {P Z}
   end, ∅)],
∅)
```

- Etat d'exécution initial
  - Environnement vide et mémoire vide

A.HARICHE a.hariche@univ-dbkm.dz

Exemple avec procédure

```
([(local P Y Z in
      Z=1
      proc {P X} Y=X end
      {P Z}
   end, ∅)],
 ∅)
```

- Première instruction

A.HARICHE a.hariche@univ-dbkm.dz

Exemple avec procédure

```
([(local P Y Z in
      Z=1
      proc {P X} Y=X end
      {P Z}
   end, ∅)],
 ∅)
```

- Environnement initial (vide)

A.HARICHE a.hariche@univ-dbkm.dz

Exemple avec procédure

```
([(local P Y Z in
      Z=1
      proc {P X} Y=X end
      {P Z}
   end, ∅)],
∅)
```

- Instruction sémantique

A.HARICHE a.hariche@univ-dbkm.dz

Exemple avec procédure

```
([(local P Y Z in
      Z=1
      proc {P X} Y=X end
      {P Z}
   end, ∅)],
∅)
```

- Pile sémantique initiale (qui contient une instruction sémantique)

A.HARICHE a.hariche@univ-dbkm.dz

Exemple avec procédure

```
([(local P Y Z in
      Z=1
      proc {P X} Y=X end
      {P Z}
   end, ∅)],
∅)
```

- Mémoire initiale (vide)

A.HARICHE a.hariche@univ-dbkm.dz

L'instruction "local"

```
([(local P Y Z in
      Z=1
      proc {P X} Y=X end
      {P Z}
   end, ∅)],
 ∅)
```

- Crééz les nouvelles variables en mémoire
- Etendez l'environnement

A.HARICHE a.hariche@univ-dbkm.dz

L'instruction "local"

```
([(Z=1
    proc {P X} Y=X end
    {P Z},        {P → p, Y → y, Z → z})],
{p, y, z})
```

- Crééz les nouvelles variables en mémoire
- Etendez l'environnement

A.HARICHE a.hariche@univ-dbkm.dz

Composition séquentielle

```
([(Z=1
   proc {P X} Y=X end
   {P Z},{P → p, Y → y, Z → z})],
{p, y, z})
```

- Faire la première composition séquentielle

A.HARICHE a.hariche@univ-dbkm.dz

Composition séquentielle

([(Z=1, $\{P \to p, Y \to y, Z \to z\}$),
(**proc** {P X} Y=X **end**
{P Z}, $\{P \to p, Y \to y, Z \to z\}$)],
$\{p, y, z\}$)

- Faire la première composition séquentielle

A.HARICHE a.hariche@univ-dbkm.dz

Affectation de Z

```
([(proc {P X} Y=X end
    {P Z},        {P → p, Y → y, Z → z})],
{p, y, z=1})
```

- Liez la variable *z* en mémoire

A.HARICHE a.hariche@univ-dbkm.dz

Deuxième composition séquentielle

([(**proc** {P X} Y=X **end**,
{P → *p*,Y → *y*,Z → *z*}),
({P Z},
{P → *p*,Y → *y*,Z → *z*})],
{*p*, *y*, *z*=1})

- Faire la deuxième composition séquentielle

A.HARICHE a.hariche@univ-dbkm.dz

Définition de la procédure

([(proc {P X} Y=X end,
{P → *p*, Y → *y*, Z → *z*}),
({P Z}, {P → *p*, Y → *y*, Z → *z*})],
{*p*, *y*, *z*=1})

- Définition de la procédure
  - **Identificateur libre** Y
  - **Argument formel** X
- L'environnement contextuel est {Y → *y*}
  - Restriction de {P → *p*, Y → *y*, Z → *z*}
- Mettez ensuite la valeur procédurale dans la mémoire

A.HARICHE a.hariche@univ-dbkm.dz

Appel de la procédure

([({P Z}, {P → *p*, Y → *y*, Z → *z*})],
{*p* = **(proc** {$ X} Y=X **end**, {Y → *y*})},
y, *z*=1})

- L'appel {P Z} a l'argument actuel Z et l'argument formel X
- Calcul de l'environnement qui sera utilisé pendant l'exécution de la procédure:
  - Commencez avec {Y → *y*}
  - Adjonction de {X → *z*}

A.HARICHE a.hariche@univ-dbkm.dz

Appel de la procédure

([(Y=X, {Y → *y*, X → *z*})],
{*p* = (**proc** {$ X} Y=X **end**, {Y → *y*}),
y, *z*=1})

A.HARICHE a.hariche@univ-dbkm.dz

L'affectation

([(Y=X, {Y → *y*, X → *z*})],
{*p* = (**proc** {$ X} Y=X **end**, {Y → *y*}),
y, *z*=1})

- Affectation de Y et X
  - La variable qui correspond à Y: *y*
  - La variable qui correspond à X: *z*

A.HARICHE a.hariche@univ-dbkm.dz

Fin de l'exemple

([],
{ p = (**proc** {$ X} Y=X **end**, {Y → y}),
y=1, z=1})

Voilà!

A.HARICHE a.hariche@univ-dbkm.dz

Importance de la valeur procédurale

- Une valeur procédurale contient deux choses
  - Le code de la procédure et l'environnement contextuel
- L'appel de la procédure construit son environnement: on commence avec l'environnement contextuel, en y ajoutant les arguments formels
- La valeur procédurale est un des concepts les plus puissants dans les langages de programmation
  - Première utilisation en Algol 68
  - Existe dans quasi tous les langages, parfois accessible au programmer (notamment dans les langages fonctionnels)
- Plus difficile à utiliser en Java, C, C++ !
  - Procédure/fonction/méthode: c'est juste du code!
  - L'environnement contextuel manque (sauf pour un "inner class")
  - En Java et C++ on peut le simuler avec un objet

A.HARICHE a.hariche@univ-dbkm.dz

Appel de procédure (1)

- L'instruction sémantique est

  $(\{\langle x\rangle\ \langle y\rangle_1 \ldots \langle y\rangle_n\}, E)$

- Si la condition d'activation est fausse ($E(\langle x\rangle)$ pas lié)
  - Suspension (attente d'exécution)
- Si $E(\langle x\rangle)$ n'est pas une procédure
  - Erreur
- Si $E(\langle x\rangle)$ est une procédure mais le nombre d'arguments n'est pas bon ($\neq n$)
  - Erreur

A.HARICHE a.hariche@univ-dbkm.dz

Appel de procédure (2)

- L'instruction sémantique est

  $(\{\langle x\rangle\ \langle y\rangle_1 \ldots \langle y\rangle_n\}, E)$

  avec

  $E(\langle x\rangle) = (\textbf{proc}\ \{\$\ \langle z\rangle_1 \ldots \langle z\rangle_n\}\ \langle s\rangle\ \textbf{end}, CE)$

- Alors empilez

  $(\langle s\rangle, CE + \{\langle z\rangle_1 \rightarrow E(\langle y\rangle_1), \ldots, \langle z\rangle_n \rightarrow E(\langle y\rangle_n)\})$

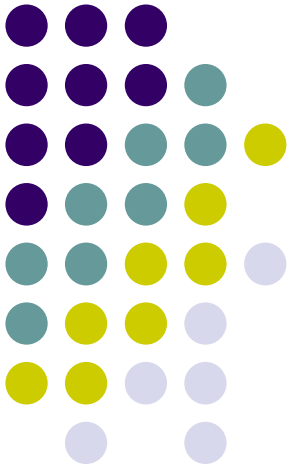
A.HARICHE a.hariche@univ-dbkm.dz

Résumé

paradigme fonctionnel

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Résumé

- Récursion sur les listes
  - Utilisation des variables libres pour simplifier les fonctions
  - Importance du langage noyau
- La sémantique d'un programme
  - Traduction en langage noyau
  - Exécution avec la machine abstraite
- Exemple d'exécution
  - Instruction sémantique, pile sémantique, mémoire
  - Sémantique des instructions du langage noyau
- Calculs avec les environnements
  - Adjonction, restriction
- Exemple d'exécution avec une procédure
  - Une valeur procédurale contient deux choses
  - Définition et appel de procédure

A.HARICHE a.hariche@univ-dbkm.dz

Idées à apprendre

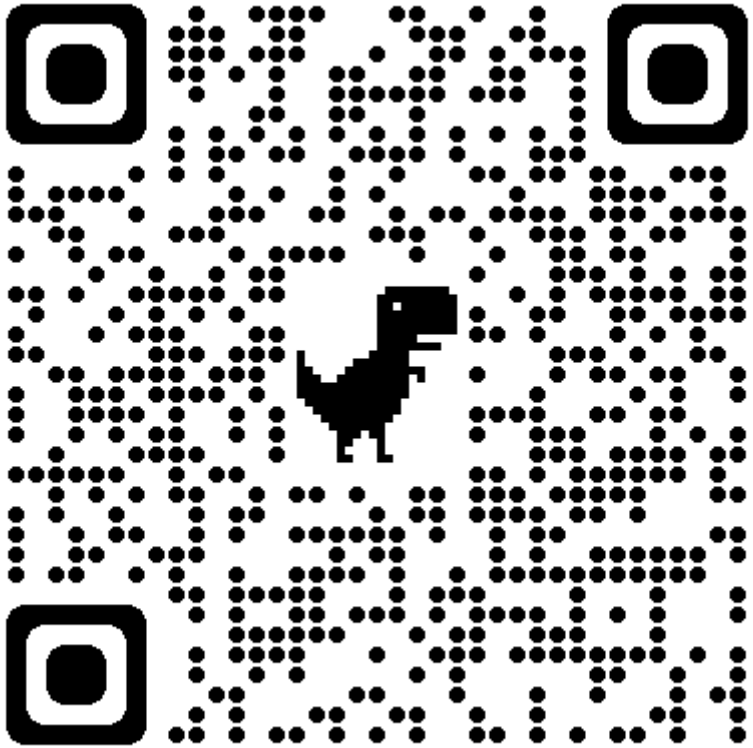
Louv1.1x

- Identifiants et environnements
- Programmation fonctionnelle
- Récursivité
- Programmation invariante
- Listes, arborescences et enregistrements
- Programmation symbolique
- Instanciation
- Généricité
- Programmation d'ordre supérieur
- Complexité et notation Big-O
- Loi de Moore
- Problèmes NP et NP-complets
- Langages du noyau
- Machines abstraites
- Sémantique mathématique

Louv1.2x

- État explicite
- Abstraction des données
- Types de données abstraites et objets
- Polymorphisme
- Héritage
- Héritage multiple
- Programmation orientée objet
- Gestion des exceptions
- Concurrence
- Non-déterminisme
- Ordonnancement et équité
- Synchronisation des flux de données
- Flux de données déterministe
- Agents et flux
- Programmation multi-agents

Lien



https://drive.google.com/drive/folders/1YBCIZzAldeiT19DIfDiREQwP-NAQ1qMN