

FSAB1402: Informatique 2

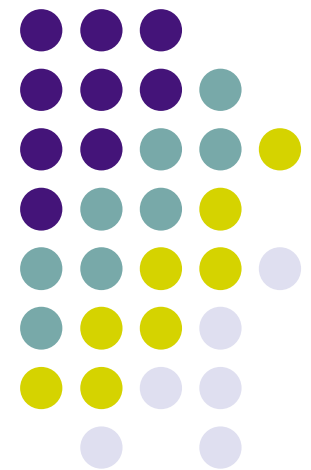
Enregistrements et Arbres

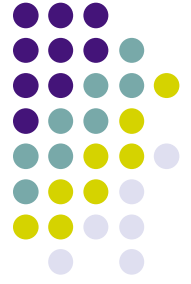


Département d'Ingénierie Informatique, UCL

Peter Van Roy

pvr@info.ucl.ac.be

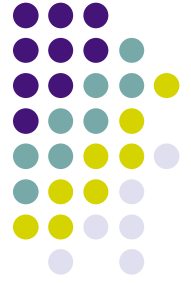




Ce qu'on va voir aujourd'hui

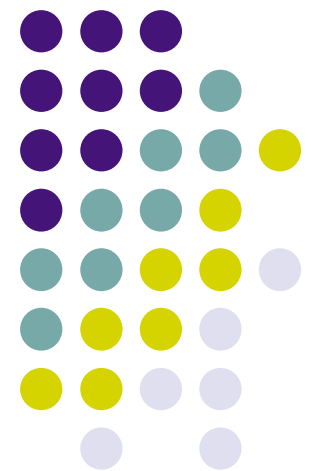
- Résumé du dernier cours
- Les tuples et les enregistrements
 - Une liste est un tuple
 - Un tuple est un enregistrement
- Les arbres
 - Les arbres binaires ordonnés
 - Arbres de recherche
- Introduction à la sémantique

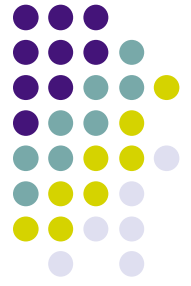
Lecture pour le cinquième cours



- Chapitre 2 (section 2.3)
 - Langage noyau et les types de base (tuples, enregistrements)
- Chapitre 3 (section 3.4.4)
 - Arbres binaires ordonnés
- Chapitre 2 (sections 1.6 et 2.4)
 - Sémantique et machine abstraite

Résumé du dernier cours





Traduction en langage noyau!

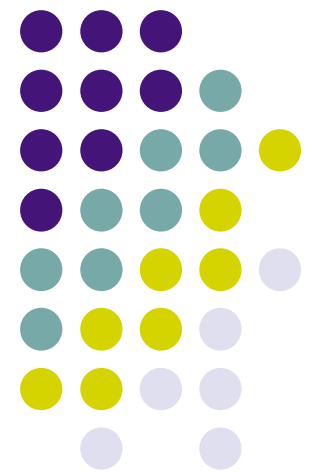
- Je vous donne une traduction en langage noyau
 - Réponse à une question de l'interrogation
- Principes:
 - N'utilisez que des instructions du langage noyau
 - Le langage noyau est un sous-ensemble de Oz
- Conséquences:
 - Toutes les variables intermédiaires deviennent visibles
 - Par de nouveaux identificateurs, bien sûr!
 - Les programmes sont plus longs
 - Les programmes s'exécutent dans le Labo
 - Le langage noyau n'est pas un ensemble de règles de grammaire (EBNF)!



Complexité calculatoire

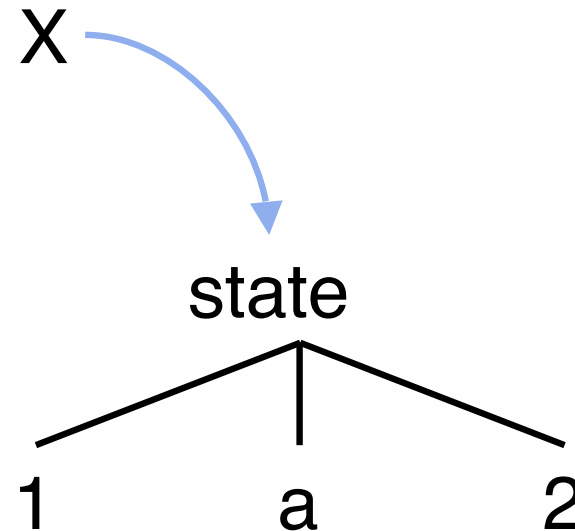
- L'efficacité d'un programme
- Outils pour calculer la complexité
 - Notations O , Ω et Θ
- La loi de Moore
 - Augmentation de la densité des circuits
- Les problèmes NP-complets
 - Vérifier une solution est simple; trouver la solution peut-être pas
- L'optimisation
 - L'optimisation prématurée est la source de tous les maux

Les tuples



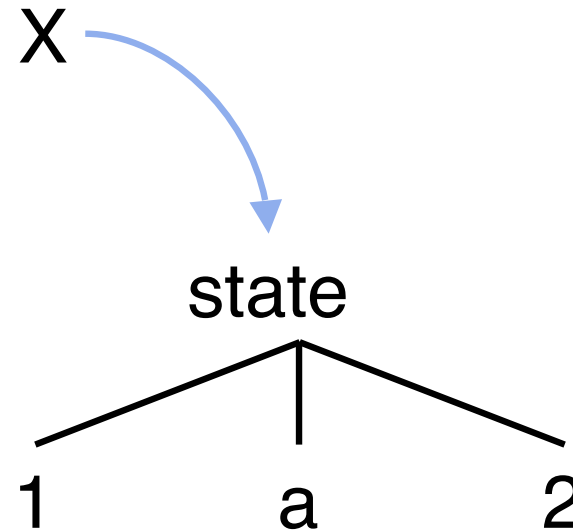
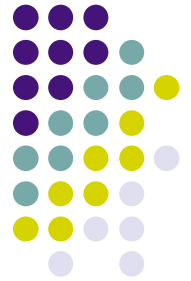
Tuples

$X = \text{state}(1 \ a \ 2)$



- Un tuple permet de combiner plusieurs valeurs
 - Par exemple: 1, a, 2
 - La position est significative: première, deuxième, troisième!
- Un tuple a une **étiquette** (“label”)
 - Par exemple: state

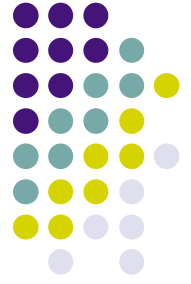
Opérations sur les tuples



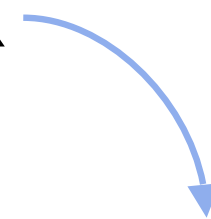
$X = \text{state}(1 \ a \ 2)$

- {Label X} renvoie *l'étiquette* du tuple X
 - Par exemple: state
 - C'est un atome
- {Width X} renvoie *la largeur* (nombre de champs)
 - Par exemple: 3
 - C'est toujours un entier positif ou zéro

Accès aux champs (opération “.”)

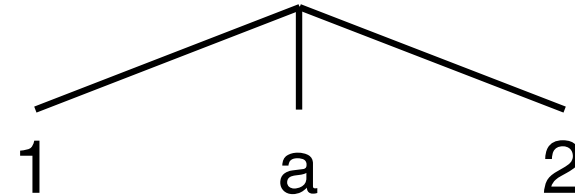


X

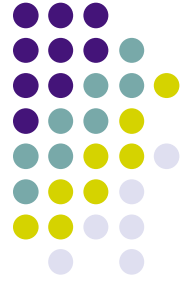


state

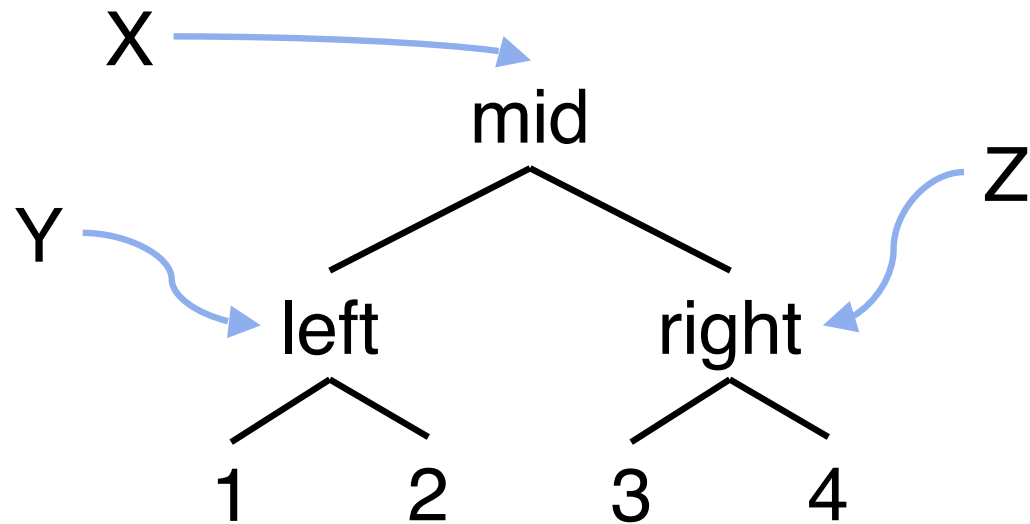
X=state(1 a 2)



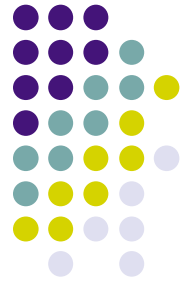
- Les champs sont numérotés de 1 jusqu'à {Width X}
- X.N renvoie le nième *champ* du tuple
 - Par exemple, X.1 renvoie 1
 - Par exemple, X.3 renvoie 2
- Dans l'expression X.N, N s'appelle le nom du champ ("feature")



Construire un arbre

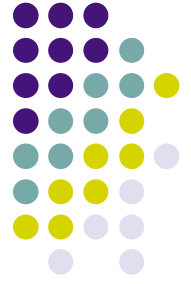


- Un arbre peut être construit avec des tuples:
declare
Y=left(1 2) Z=right(3 4)
X=mid(Y Z)



Opération d'égalité (==)

- Tester l'égalité avec un nombre ou un atome
 - C'est simple: le nombre ou l'atome doit être le même
 - Souvent le pattern matching est plus court
- Tester l'égalité des arbres
 - C'est simple aussi: les deux arbres doivent avoir les mêmes sous-arbres et les mêmes feuilles
 - Attention aux arbres avec des cycles!
 - La comparaison marche, mais un programme écrit naïvement risque d'avoir une boucle infinie
 - Conseil: pour l'instant, éviter ce genre d'arbres



Résumé des tuples

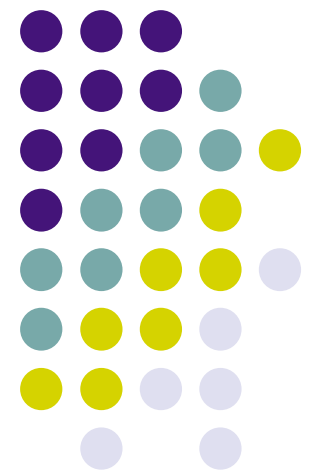
- Tuple
 - Étiquette (“label”)
 - Largeur (“width”)
 - Champ (“field”)
 - Nom de champ (“feature”)
- On peut construire des arbres
- On peut les utiliser dans le pattern matching
- On peut les comparer avec “==”



Une liste est un tuple

- Une liste $H|T$ est un tuple $'|'$ ($H T$)
- Ceci permet la simplicité: au lieu d'avoir deux concepts (tuples et listes), il n'y a qu'un concept (tuple)
- A cause de leur grande utilité, les listes sont soutenues par un sucre syntaxique
 - C'est uniquement pour le confort du programmeur, dans le langage noyau les listes sont des tuples

Les enregistrements ("records")





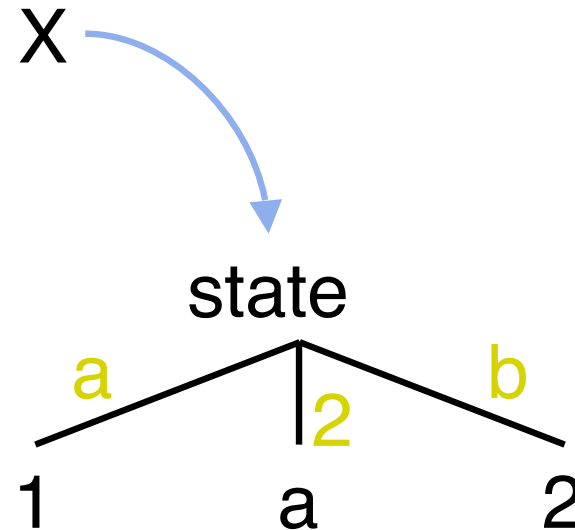
Enregistrements (“records”)

- Un enregistrement est une généralisation d’un tuple
 - Les noms des champs peuvent être des atomes
 - Les noms des champs peuvent être n’importe quel entier
 - Pas besoin de commencer avec 1
 - Pas besoin d’être consécutif
- Un enregistrement a aussi une étiquette et un largeur



Enregistrements

$X = \text{state}(a:1 \ 2:a \ b:2)$

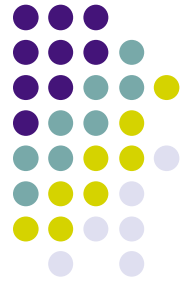


- La position n'est plus significative
 - Il y a le nom du champ qui est significatif à la place
- L'accès aux champs est comme avec les tuples
 - $X.a=1$

Opérations sur les enregistrements



- Il y a les opérations d'étiquette et de largeur
 - $\{\text{Label } X\}=\text{state}$
 - $\{\text{Width } X\}=3$
- Il y a le test d'égalité
 - $X==\text{state}(a:1 \ b:2 \ 2:a)$
- Il y a une nouvelle opération: l'arité
 - La liste des noms de champs
 - $\{\text{Arity } X\}=[2 \ a \ b]$ (en ordre lexicographique)
 - L'arité marche aussi pour les tuples et les listes (!)



Un tuple est un enregistrement

- L'enregistrement

$X = \text{state}(1:a 2:b 3:c)$

est équivalent à

$X = \text{state}(a b c)$

- Dans un **tuple**, tous les champs sont numérotés consécutivement à partir de 1
- Qu'est-ce qui se passe si on fait

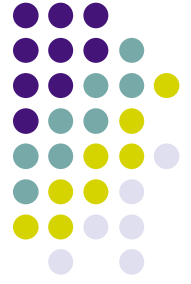
$X = \text{state}(a 2:b 3:c)$

ou

$X = \text{state}(2:b 3:c a)$

- Dans un **enregistrement**, tous les champs sans nom sont numérotés consécutivement à partir de 1

Il n'y a que des enregistrements



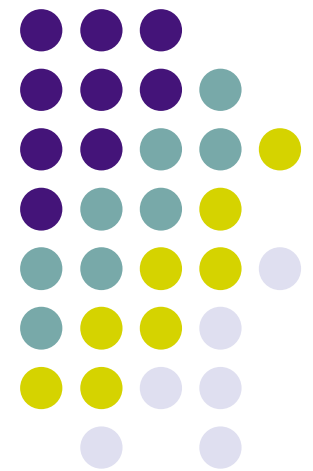
- Dans le langage noyau il n'y a que des enregistrements
 - Un atome est un enregistrement dont le largeur est 0
 - Un tuple est un enregistrement dont les noms des champs sont des entiers successifs à partir de 1
 - Si on ne satisfait pas à cette condition, c'est un enregistrement
 - Une liste est construit avec les tuples nil et '|'(X Y)
- C'est la simplicité

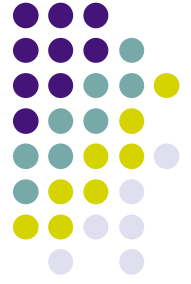


Quelques exemples

- Pour les enregistrements suivants: s'agit-il d'un tuple ou une liste?
 - $A = a(1:a\ 2:b\ 3:c)$
 - $B = a(1:a\ 2:b\ 4:c)$
 - $C = a(0:a\ 1:b\ 2:c)$
 - $D = a(1:a\ 2:b\ 3:c\ d)$
 - $E = a(a\ 2:b\ 3:c\ 4:d)$
 - $F = a(2:b\ 3:c\ 4:d\ a)$
 - $G = a(1:a\ 2:b\ 3:c\ \text{foo}:d)$
 - $H = '|'(1:a\ 2:'|')(1:b\ 2:\text{nil})$
 - $I = '|'(1:a\ 2:'|')(1:b\ 3:\text{nil})$

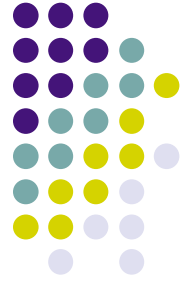
Les arbres





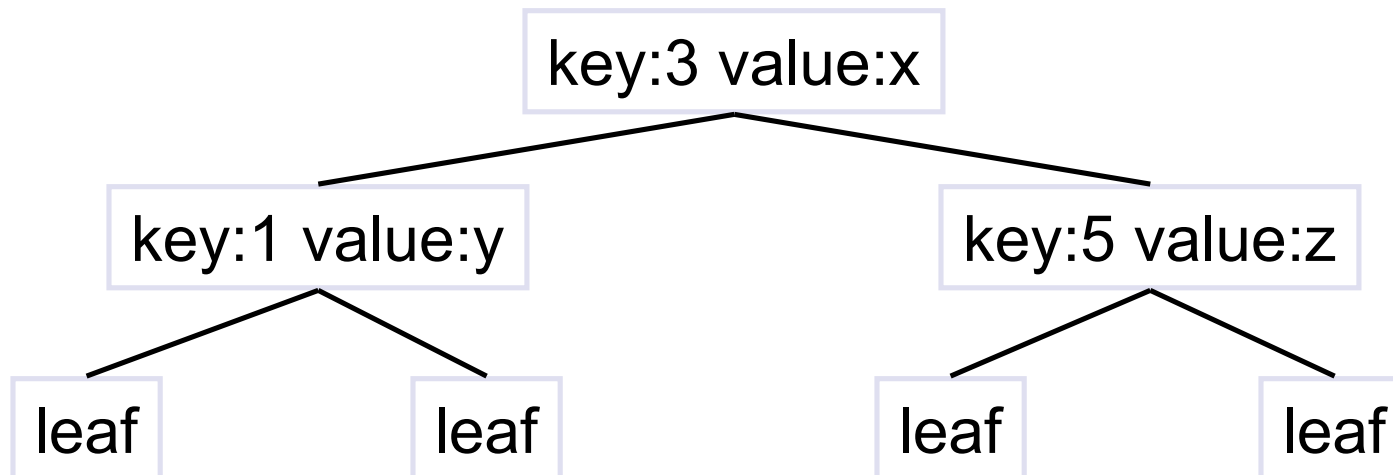
Les arbres

- A part les listes, les arbres sont les structures de données inductives les plus importantes
- Un **arbre** est une feuille (« leaf ») ou une séquence de zéro ou plusieurs arbres
- Une liste a une structure linéaire, mais un arbre a une structure bifurcante
- Il y a énormément de différentes sortes d'arbres. Nous allons regarder une seule sorte d'arbre, **les arbres binaires ordonnés**.



Arbres binaires ordonnés

- $\langle \text{btree } T \rangle ::= \text{tree}(\text{key}:T \text{ value}:<\text{value}> \langle \text{btree } T \rangle \langle \text{btree } T \rangle)$
| leaf
- **Binaire**: chaque noeud qui n'est pas une feuille a deux sous-arbres
- **Ordonné**: les clés du sous-arbre gauche $<$ clé du noeud $<$ clés du sous-arbre droite

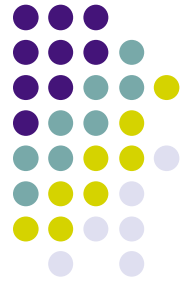


Arbres de recherche



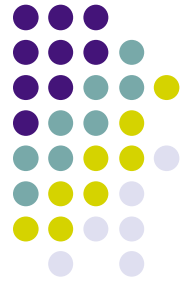
- **Arbre de recherche**: Un arbre qui est utilisé pour rechercher des informations, insérer des informations, ou enlever des informations
- Définissons trois opérations:
 - **{Lookup X T}**: renvoie la valeur qui correspond à la clé X
 - **{Insert X V T}**: renvoie un nouvel arbre qui contient (X,V)
 - **{Delete X T}**: renvoie un nouvel arbre qui ne contient pas X

Rechercher des informations



- Il y a quatre cas de figure:
- X n'est pas trouvé
- X est trouvé
- X peut être dans le sous-arbre gauche
- X peut être dans le sous-arbre droite

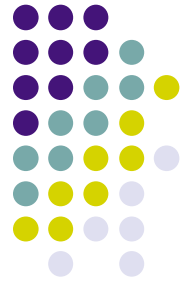
```
fun {Lookup X T}
  case T
  of leaf then notfound
  [] tree(key:Y value:V T1 T2) andthen X==Y then
    found(V)
  [] tree(key:Y value:V T1 T2) andthen X<Y then
    {Lookup X T1}
  [] tree(key:Y value:V T1 T2) andthen X>Y then
    {Lookup X T2}
  end
end
```



Insérer des informations

- Il y a quatre cas de figure:
- (X,V) est inséré tout de suite
- (X,V) remplace un nœud existant avec la même clé
- (X,V) est inséré dans le sous-arbre gauche
- (X,V) est inséré dans le sous-arbre droite

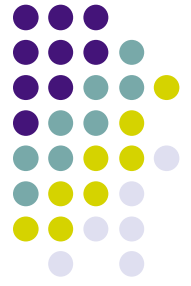
```
fun {Insert X V T}  
  case T  
  of leaf then tree(key:X value:V leaf leaf)  
  [] tree(key:Y value:W T1 T2) andthen X==Y then  
    tree(key:X value:V T1 T2)  
  [] tree(key:Y value:W T1 T2) andthen X<Y then  
    tree(key:Y value:W {Insert X V T1} T2)  
  [] tree(key:Y value:W T1 T2) andthen X>Y then  
    tree(key:Y value:W T1 {Insert X V T2})  
  end  
end
```



Enlever des informations

- Il y a quatre cas de figure:
- (X, V) n'est pas dans l'arbre
- (X, V) est enlevé tout de suite
- (X, V) est enlevé du sous-arbre gauche
- (X, V) est enlevé du sous-arbre droite
- Vrai?

```
fun {Delete X T}  
  case T  
  of leaf then leaf  
  [] tree(key:Y value:W T1 T2) andthen X==Y then  
    leaf  
  [] tree(key:Y value:W T1 T2) andthen X<Y then  
    tree(key:Y value:W {Delete X T1} T2)  
  [] tree(key:Y value:W T1 T2) andthen X>Y then  
    tree(key:Y value:W T1 {Delete X T2})  
  end  
end
```

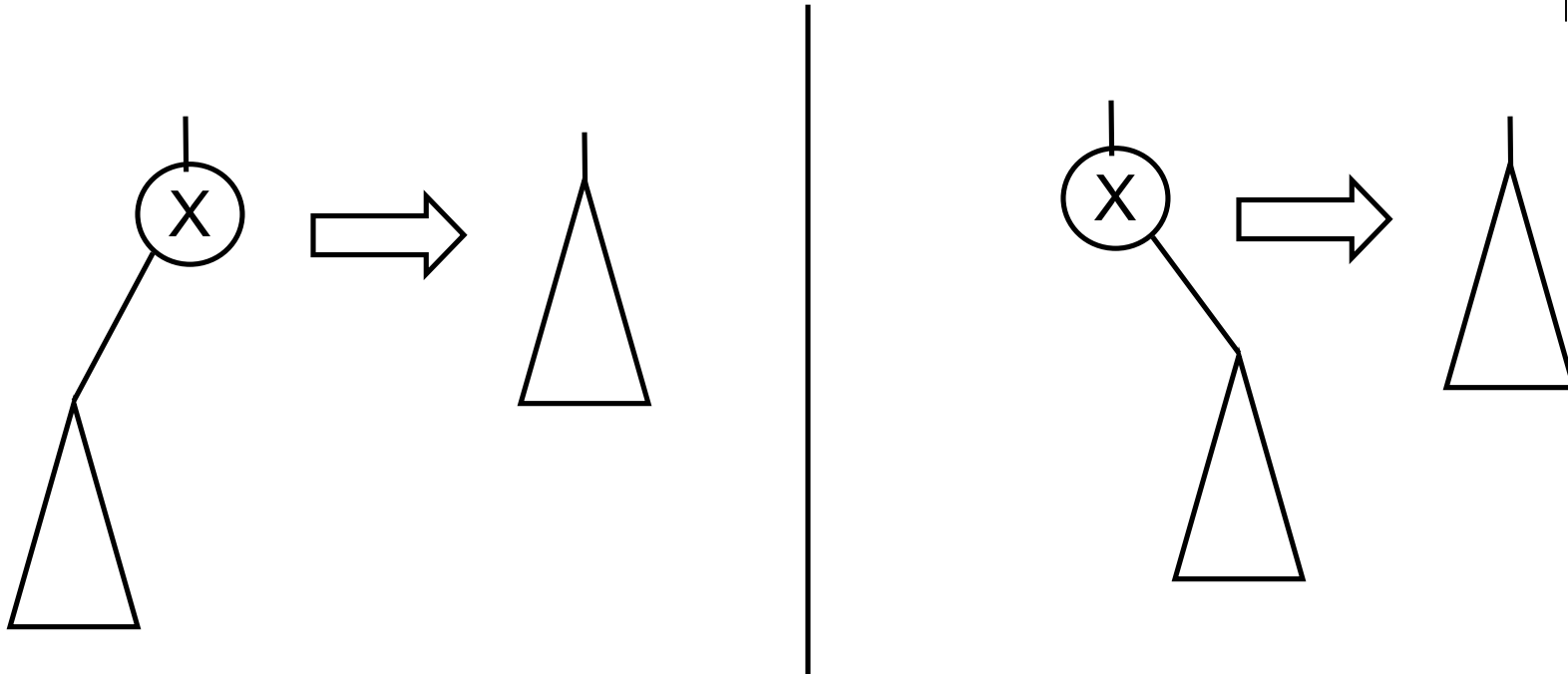
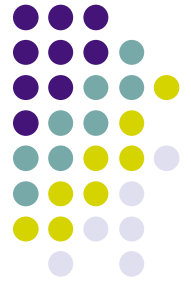


Enlever des informations

- Il y a quatre cas de figure:
- (X, V) n'est pas dans l'arbre
- (X, V) est enlevé tout de suite
- (X, V) est enlevé du sous-arbre gauche
- (X, V) est enlevé du sous-arbre droite
- Vrai? **FAUX!**

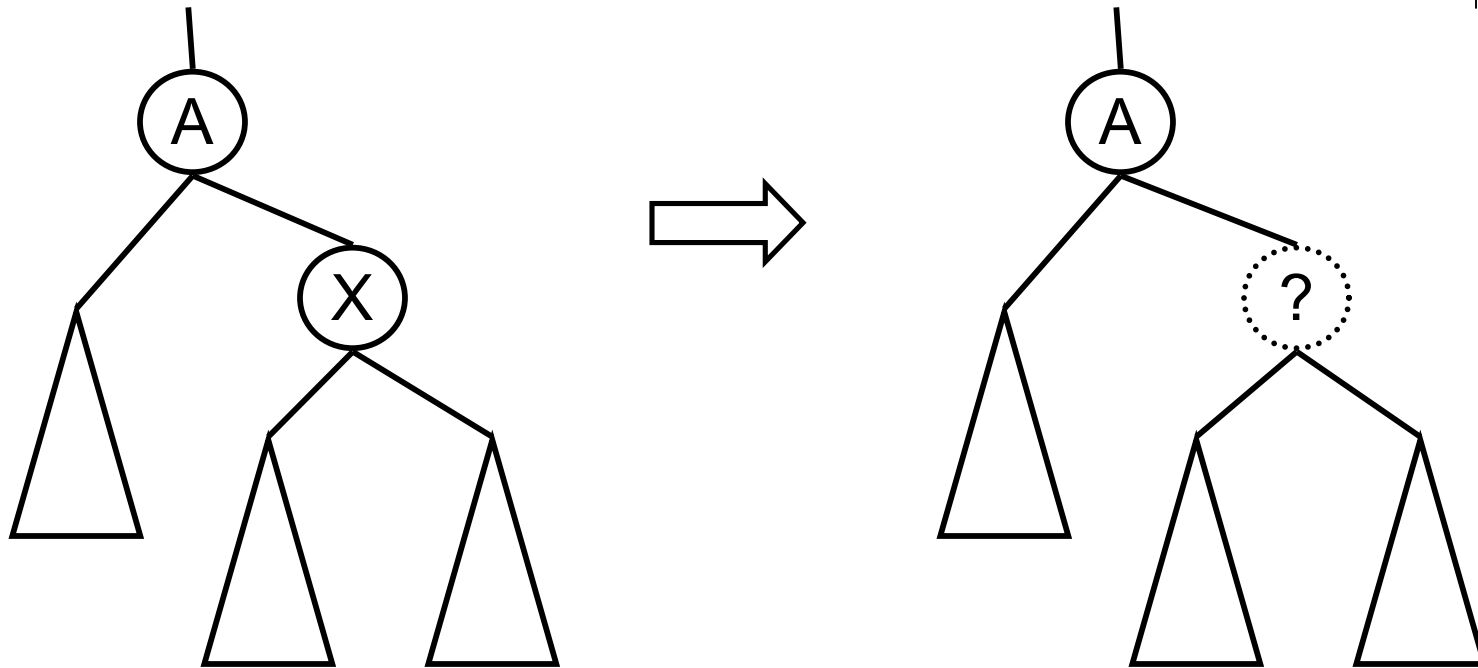
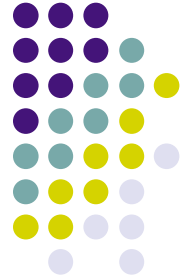
```
fun {Delete X T}
  case T
  of leaf then leaf
  [] tree(key:Y value:W T1 T2) andthen X==Y then
    leaf
  [] tree(key:Y value:W T1 T2) andthen X<Y then
    tree(key:Y value:W {Delete X T1} T2)
  [] tree(key:Y value:W T1 T2) andthen X>Y then
    tree(key:Y value:W T1 {Delete X T2})
  end
end
```

Enlever un élément d'un arbre binaire



Enlever X d'un arbre binaire. Le cas est simple quand un des sous-arbres est vide. Ce n'est pas toujours vrai!

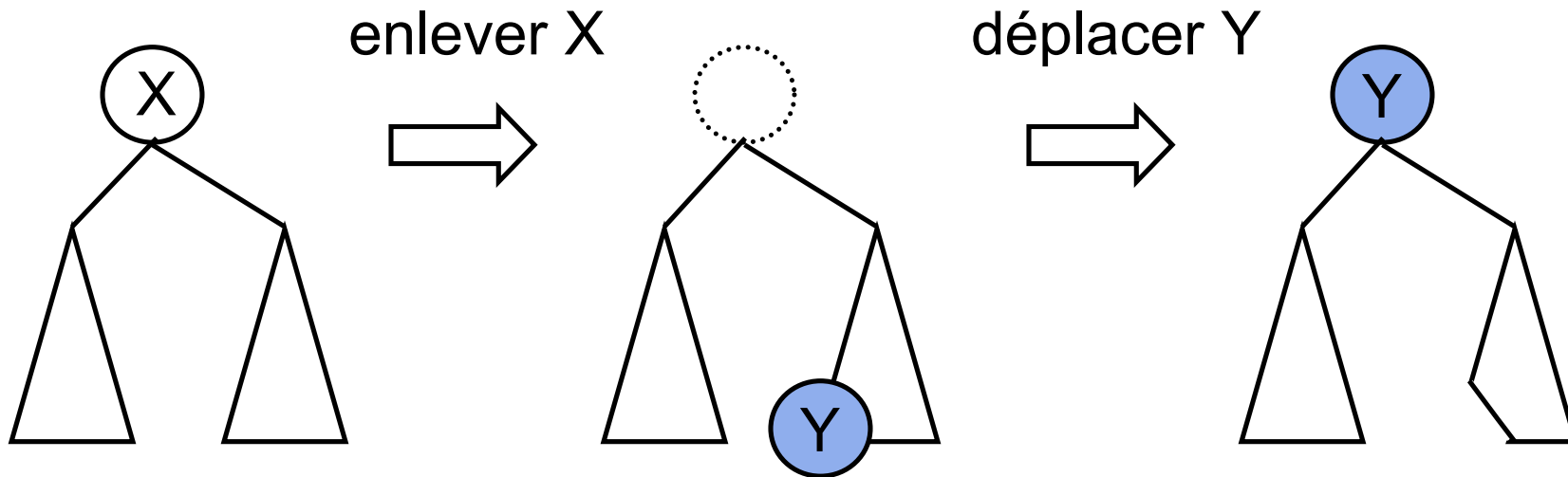
Enlever un élément d'un arbre binaire



Enlever X d'un arbre binaire. Le problème est de **réparer l'arbre** après la disparition de X.



Enlever un élément (1)



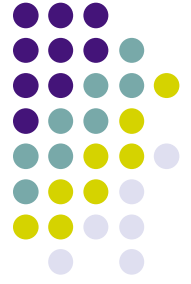
Remplir le "trou" après l'enlèvement de X

Enlever un élément (2)



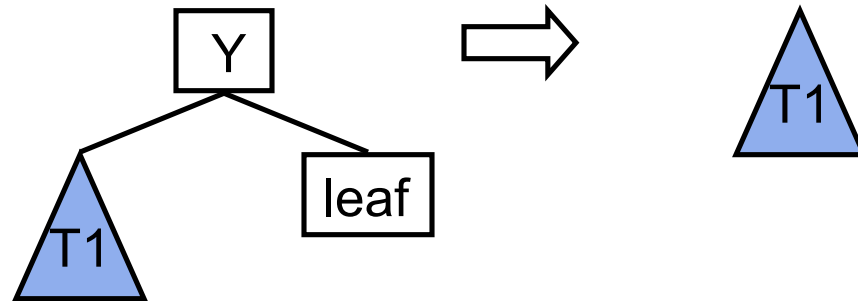
- Le problème avec l'autre programme est **qu'il n'enlève pas correctement un nœud non-feuille**
- Pour le faire correctement, l'arbre doit être **reorganisé**:
 - Un nouvel élément doit remplacer l'élément enlevé
 - L'élément peut être le plus petit du sous-arbre droit ou le plus grand du sous-arbre gauche

```
fun {Delete X T}
  case T
  of leaf then leaf
  [] tree(key:Y value:W T1 T2) andthen X==Y then
    case {RemoveSmallest T2}
    of none then T1
    [] Yp#Wp#Tp then
      tree(key:Yp value:Wp T1 Tp)
    end
  end
end
```



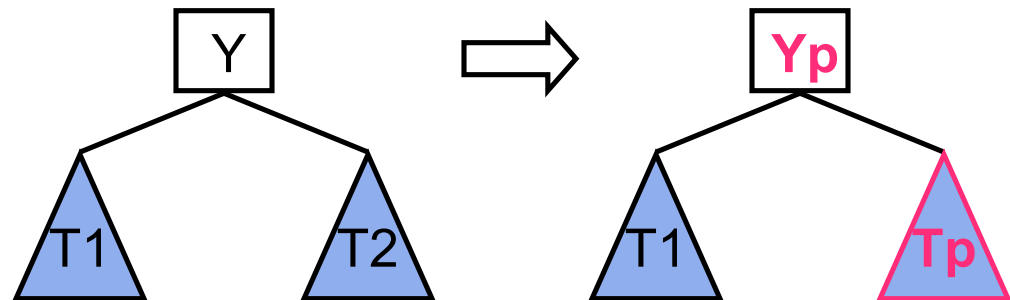
Enlever un élément (3)

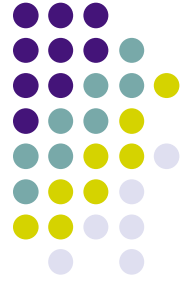
- Pour enlever une racine Y , il y a deux possibilités:



- Un sous-arbre est une feuille. On prend l'autre.

- Aucun sous-arbre n'est une feuille. On enlève un élément d'un des sous-arbres.

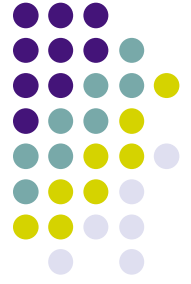




Enlever un élément (4)

- La fonction `{RemoveSmallest T}` enlève le plus petit élément de T et renvoie le triplet `Xp#Vp#Tp`, où `(Xp,Vp)` est le plus petit élément et `Tp` est l'arbre restant

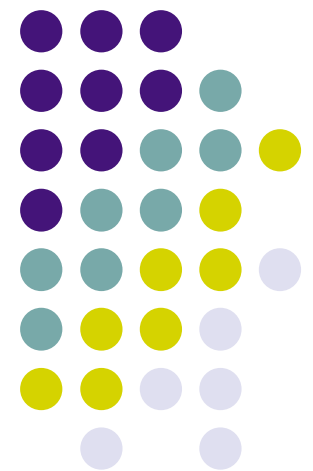
```
fun {RemoveSmallest T}
  case T
  of leaf then none
  [] tree(key:X value:V T1 T2) then
    case {RemoveSmallest T1}
    of none then X#V#T2
    [] Xp#Vp#Tp then
      Xp#Vp#tree(key:X value:V Tp T2)
    end
  end
end
```



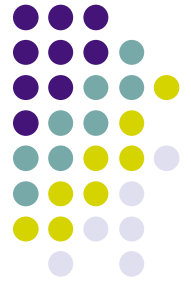
Enlever un élément (5)

- Pourquoi l'opération d'enlèvement est-elle compliquée?
- C'est parce que l'arbre satisfait une **condition globale**, d'être ordonné
- L'opération d'enlèvement doit travailler pour maintenir cette condition
- Beaucoup d'algorithmes sur des arbres dépendent des conditions globales et doivent travailler dur pour les maintenir
- Le bon côté de la condition globale est qu'elle donne à l'arbre **une étincelle de magie**: l'arbre se comporte un peu comme un être vivant (« **comportement orienté but** » ou « **goal-oriented** »)
 - Les êtres vivants sont typiquement très dirigés vers des buts

Introduction à la sémantique

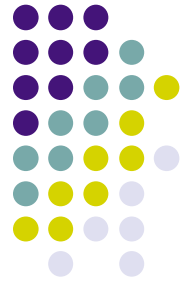


Du langage pratique vers le langage noyau

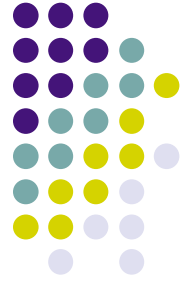


- Un langage pratique contient beaucoup de concepts qui sont là pour le confort du programmeur
- Comment on peut donner un sens exact à tout cela?
 - En exprimant tout dans un langage simple, le langage noyau
 - Nous allons définir la sémantique par rapport au langage noyau

Votre programme est-il correct?

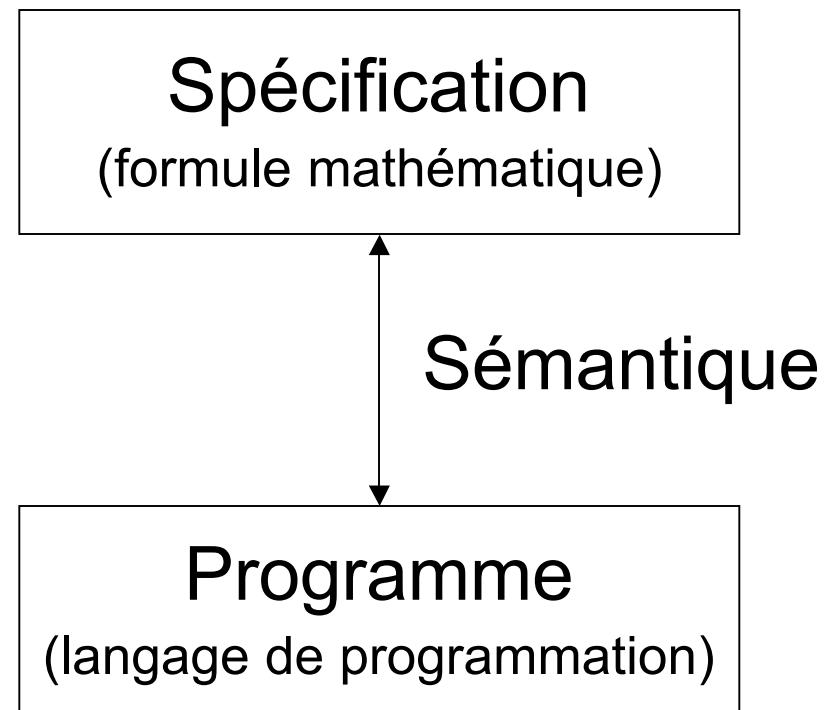


- "Un programme est correct quand il fait ce qu'on veut qu'il fasse"
- Comment se rassurer de cela?
- Il y a deux points de départ :
 - **La spécification du programme**: une définition du résultat du programme en termes de l'entrée (typiquement une fonction ou relation mathématique)
 - **La sémantique 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



Les trois piliers

- La spécification:
ce qu'on veut
- Le programme:
ce qu'on a
- La sémantique permet de faire le lien entre les deux: de prouver que ce qu'on a marche comme on veut!

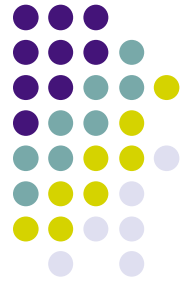




Induction mathématique

- 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 $H|T$

Exemple: exactitude de la factorielle



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

$$0! = 1$$

$$n! = n * ((n-1)!) \text{ si } n > 0$$

- Le **programme** (en langage de programmation)

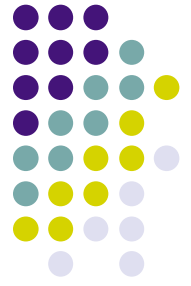
```
fun {Fact N}
```

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

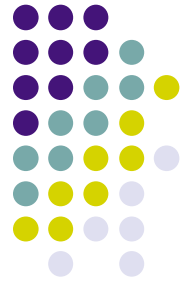
```
end
```

- Où est la **sémantique** du langage?
 - Nous la verrons la semaine prochaine!
 - Aujourd'hui le raisonnement sera intuitif

Raisonnement pour la factorielle



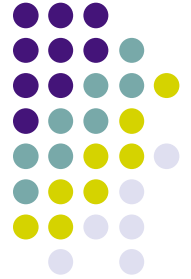
- Il faut démontrer que l'exécution de {Fact N} donne $n!$ pour tout n
- **Cas de base:** $n=0$
 - La spécification montre $0!=1$
 - L'exécution de {Fact 0} avec la sémantique montre {Fact 0}=1
- **Cas inductif:** $(n-1) \rightarrow n$
 - L'exécution de {Fact N}, selon la sémantique, montre que {Fact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



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

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 $(\langle s \rangle, E)$
 - Une instruction avec son environnement
- Pile sémantique $ST = [(\langle s \rangle_1, E_1), \dots, (\langle s \rangle_n, E_n)]$
 - Une pile d'instructions sémantiques
- Exécution $(ST_1, \sigma_1) \rightarrow (ST_2, \sigma_2) \rightarrow (ST_3, \sigma_3) \rightarrow \dots$
 - Une séquence d'états d'exécution (pile + mémoire)



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
 - $E = \{X \rightarrow x, Y \rightarrow y\}$
 - $E(X) = x$
 - $E(Y) = y$

L'environnement pendant l'exécution

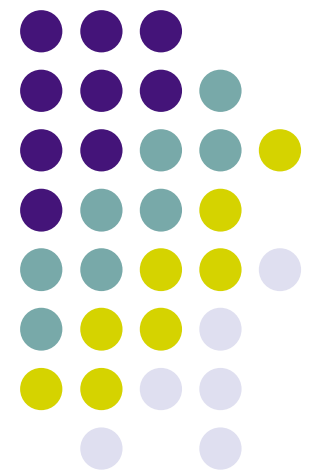


- Prenons une instruction:

```
(E0) local X Y in (E1)  
      X=2 Y=3  
      local X in (E2)  
          X=Y*Y  
          {Browse X}  
      end (E3)  
end
```

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

Résumé





Résumé

- Les structures de données
 - Tuples et enregistrements
 - Il n'a que les enregistrements dans le langage noyau
- Les arbres
 - Opérations sur les arbres binaires ordonnés
 - L'importance et la difficulté de maintenir les conditions globales
- Introduction à la sémantique