

# FSAB1402: Informatique 2

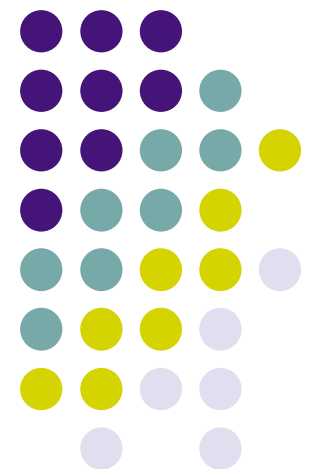
## Sémantique Formelle

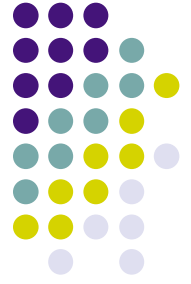


Département d'Ingénierie Informatique, UCL

**Peter Van Roy**

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



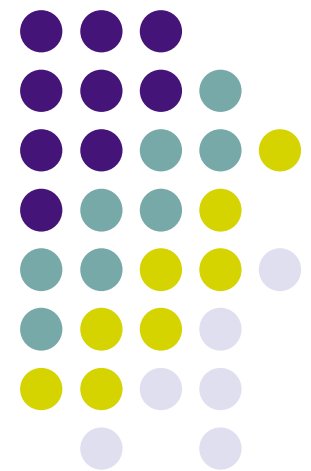


# Lecture pour ce cours

- Chapitre 1 (section 1.6):
  - L'exactitude
- Chapitre 2 (sections 2.2, 2.3, 2.4, 2.6):
  - La mémoire à affectation unique
  - Le langage noyau déclaratif
  - La sémantique du langage noyau
  - Du langage noyau au langage pratique

# Résumé du dernier cours

---

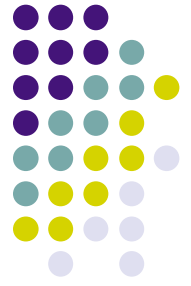




# Tuples et enregistrements

- Un **enregistrement** a une étiquette, des champs et des noms de champs
  - L'étiquette et les noms des champs sont des atomes ou des entiers
  - L'opération Arity donne une liste des noms des champs en ordre lexicographique
- Un **tuple** est un enregistrement où des noms des champs sont des entiers consécutifs à partir de 1
- Une **liste** est un tuple:
  - Une liste est l'atome 'nil' ou le tuple ']'(H T) où T est une liste
  - Il y a du soutien syntaxique pour les listes

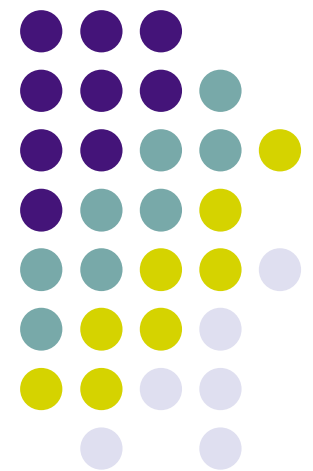
# Arbres de recherche

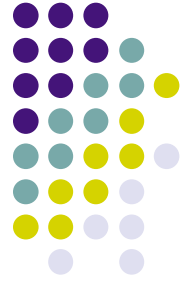


- Un **arbre binaire ordonné**
  - Chaque noeud a une clé
  - Pour chaque racine: Les clés du sous-arbre de gauche sont plus petites que la clé de la racine, qui est plus petite que les clés du sous-arbre de droite
- Un **arbre de recherche**
  - Chaque noeud a une paire (clé,valeur); les clés sont ordonnées
  - Chercher, insérer, enlever une paire d'un arbre selon la clé
- **Conditions globales**
  - Quand un arbre est **ordonné**, c'est une condition sur **tout l'arbre**
  - Chaque fois quand on fait quelque chose, comme enlever une paire, il faut **reorganiser l'arbre** pour maintenir la condition globale
  - Le maintien d'une condition globale est un exemple d'un **calcul orienté-but** ("goal-oriented computation")

# La sémantique du langage

---





# La sémantique du langage

- La sémantique du langage donne une **explication mathématique** de l'exécution de tout programme
- Commençons avec un programme quelconque. Il y a deux pas à suivre pour avoir l'explication:
  1. La **traduction en langage noyau**
  2. L'**exécution du programme en langage noyau**
- L'exécution se fait dans une machine simplifiée appelée **machine abstraite**
  - La machine abstraite a une définition mathématique précise
- On peut **raisonner** sur l'exécution
  - Calculs de complexité, exactitude, comprendre l'exécution (pourquoi la récursion terminale marche)



# Le langage noyau

- Notation EBNF;  $\langle s \rangle$  désigne une instruction

```
 $\langle s \rangle$  ::= skip  
      |  $\langle X \rangle_1 = \langle X \rangle_2$   
      |  $\langle X \rangle = \langle V \rangle$   
      | local  $\langle X \rangle$  in  $\langle s \rangle$  end  
      | if  $\langle X \rangle$  then  $\langle s \rangle_1$  else  $\langle s \rangle_2$  end  
      | {  $\langle X \rangle$   $\langle X \rangle_1$  ...  $\langle X \rangle_n$  }  
      | case  $\langle X \rangle$  of  $\langle p \rangle$  then  $\langle s \rangle_1$  else  $\langle s \rangle_2$  end  
  
 $\langle v \rangle$  ::= ...  
 $\langle p \rangle$  ::= ...
```



# Valeurs dans le langage noyau

- Notation EBNF;  $\langle v \rangle$  désigne une valeur,  $\langle p \rangle$  désigne une forme (pattern)

$\langle v \rangle ::= \langle \text{number} \rangle \mid \langle \text{record} \rangle \mid \langle \text{procedure} \rangle$

$\langle \text{number} \rangle ::= \langle \text{int} \rangle \mid \langle \text{float} \rangle$

$\langle \text{record} \rangle, \langle p \rangle ::= \langle \text{literal} \rangle$

$\mid \langle \text{literal} \rangle (\langle \text{feature} \rangle_1 : \langle x \rangle_1 \dots \langle \text{feature} \rangle_n : \langle x \rangle_n)$

$\langle \text{procedure} \rangle ::= \text{proc } \{ \$ \langle x \rangle_1 \dots \langle x \rangle_n \} \langle s \rangle \text{end}$

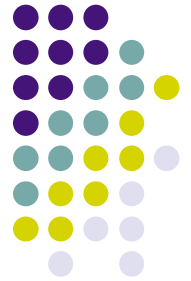
$\langle \text{literal} \rangle ::= \langle \text{atom} \rangle \mid \langle \text{bool} \rangle$

$\langle \text{feature} \rangle ::= \langle \text{atom} \rangle \mid \langle \text{bool} \rangle \mid \langle \text{int} \rangle$

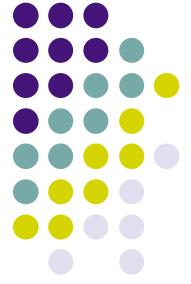
$\langle \text{bool} \rangle ::= \text{true} \mid \text{false}$

- Souvenez-vous qu'un tuple est un enregistrement (record) et qu'une liste est un tuple! Les enregistrements suffisent donc.

# L'exécution d'une instruction en langage noyau



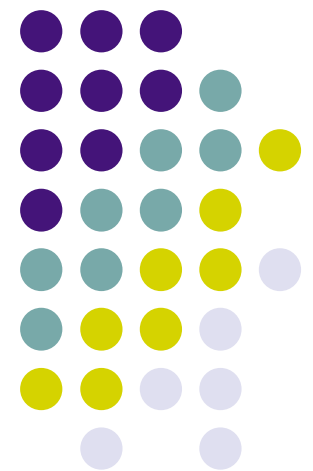
- Pour exécuter une instruction, il faut:
  - Un **environnement E** pour faire le lien avec la mémoire
  - Une **mémoire  $\sigma$**  qui contient les variables et les valeurs
- Instruction sémantique ( **$\langle s \rangle$ , E**)
  - Remarquez: chaque instruction a son propre environnement!
    -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



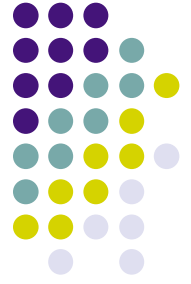
# Début de l'exécution

- Etat initial
  - $([(\langle s \rangle, \emptyset)], \emptyset)$ 
    - Instruction  $\langle s \rangle$  avec environnement vide  $\emptyset$  (pas d'identificateurs libres):  $(\langle s \rangle, \emptyset)$
    - Pile contient une instruction sémantique:  $[(\langle s \rangle, \emptyset)]$
    - Mémoire vide:  $\emptyset$  (pas encore de variables)
- A chaque pas
  - Enlevez l'instruction du sommet de la pile
  - Exécutez l'instruction
- Quand la pile est vide, l'exécution s'arrête

# Exemple d'une exécution

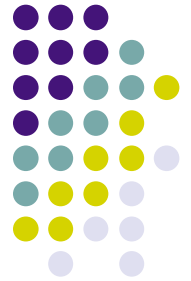


# L'instruction en langage noyau



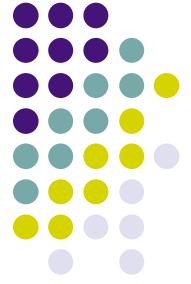
```
local X in
  local B in
    B=true
    if B then X=1 else skip end
  end
end
```

# Début de l'exécution: 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



# 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



# 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

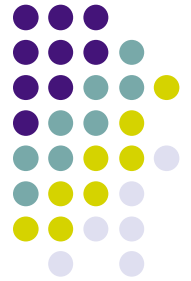




# Affectation de B

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

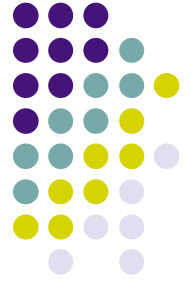
- Affectez  $b$  à **true**



# Instruction “if”

$([(X=1, \{B \rightarrow b, X \rightarrow x\})],$   
 $\{b=\text{true}, x\})$

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



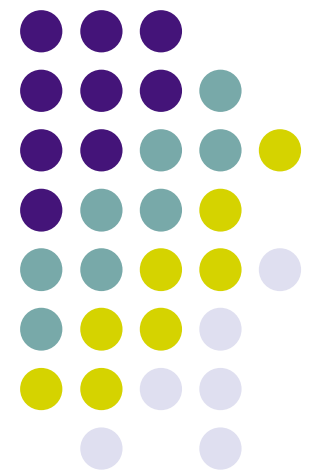
# Affectation de X

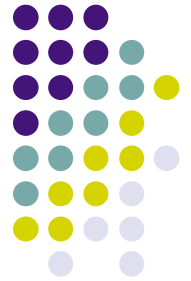
([],  
{*b*=**true**, *x*=1})

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

# Calculs avec des environnements

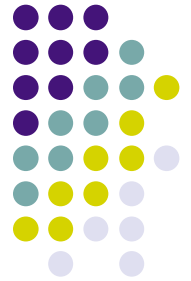
---





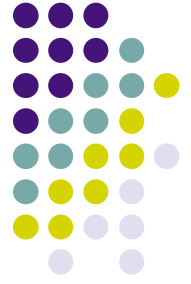
# Calculs avec l'environnement

- 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**  $\langle x \rangle$  **in**  $\langle s \rangle$  **end**
- **Restriction:**  $CE = E_{\{X, Y, Z\}}$ 
  - Enlever des liens d'un environnement
  - Pour le calcul d'un environnement contextuel



# L'adjonction

- Pour une instruction **local**  
**local** X **in** ( $E_1$ )  
    X=1  
    **local** X **in** ( $E_2$ )  
        X=2  
        {Browse X}  
    **end**  
**end**
- $E_1 = \{\text{Browse} \rightarrow b, X \rightarrow x\}$
- $E_2 = E_1 + \{X \rightarrow y\}$
- $E_2 = \{\text{Browse} \rightarrow b, X \rightarrow y\}$



# La restriction

- Pour une définition de procédure

**local** A B C **in**

A=1 B=2 C=3 (E)

**fun** {AddB X}

X+B

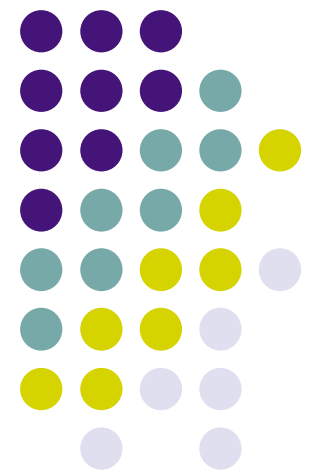
**end**

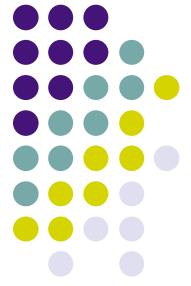
**end**

- $E = \{A \rightarrow a, B \rightarrow b, C \rightarrow c\}$   $\sigma = \{a=1, b=2, c=3\}$
- $CE = E|_{\{B\}} = \{B \rightarrow b\}$

# Sémantique de chaque instruction

---

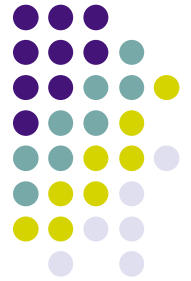




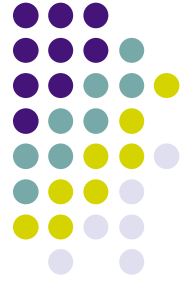
# Sémantique des instructions

- 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**  $\langle x \rangle$  **in**  $\langle s \rangle$  **end**
- Vous pouvez regarder les autres pendant les travaux pratiques

# Les instructions du langage noyau

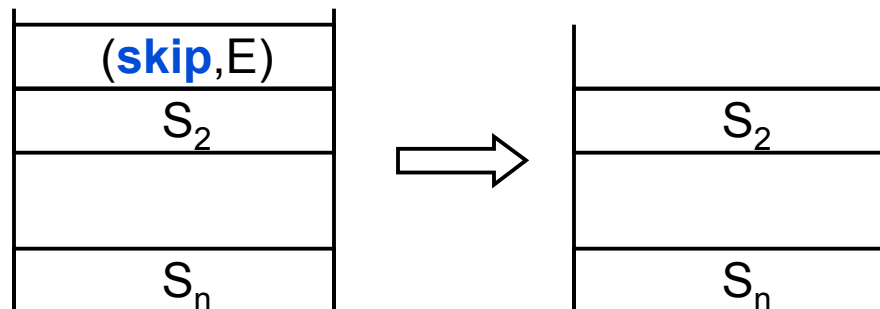


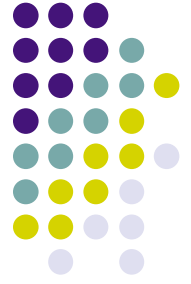
- Les instructions du langage noyau sont:
  - **skip**
  - $\langle s \rangle_1 \langle s \rangle_2$  (composition séquentielle)
  - **local**  $\langle x \rangle$  **in**  $\langle s \rangle$  **end**
  - $\langle x \rangle = \langle v \rangle$  (affectation)
  - **if**  $\langle x \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end** (conditionnel)
  - Création de nombre et d'enregistrement
  - Définition de procédure
  - Appel de procédure
  - **case**  $\langle x \rangle$  **of**  $\langle p \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end**  
(correspondance de formes)



# skip

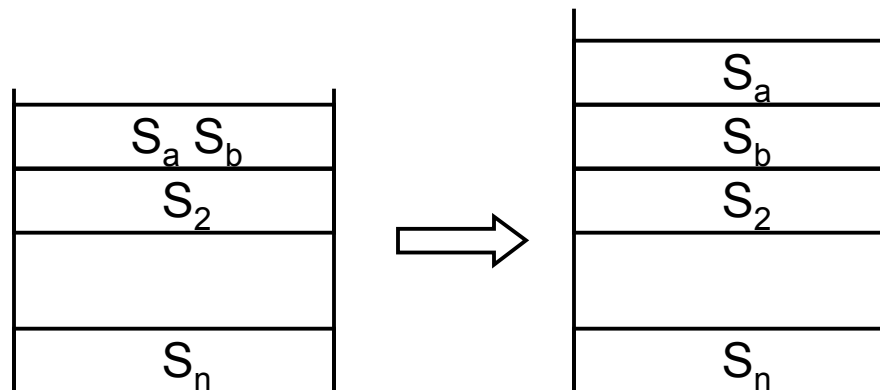
- L'instruction la plus simple
- Elle ne fait "rien"!
- Etat de l'entrée:  $([(\text{skip}, E), S_2, \dots, S_n], \sigma)$
- Etat de la sortie:  $([S_2, \dots, S_n], \sigma)$
- C'est tout!





# $\langle S \rangle_1 \langle S \rangle_2$ (composition séquentielle)

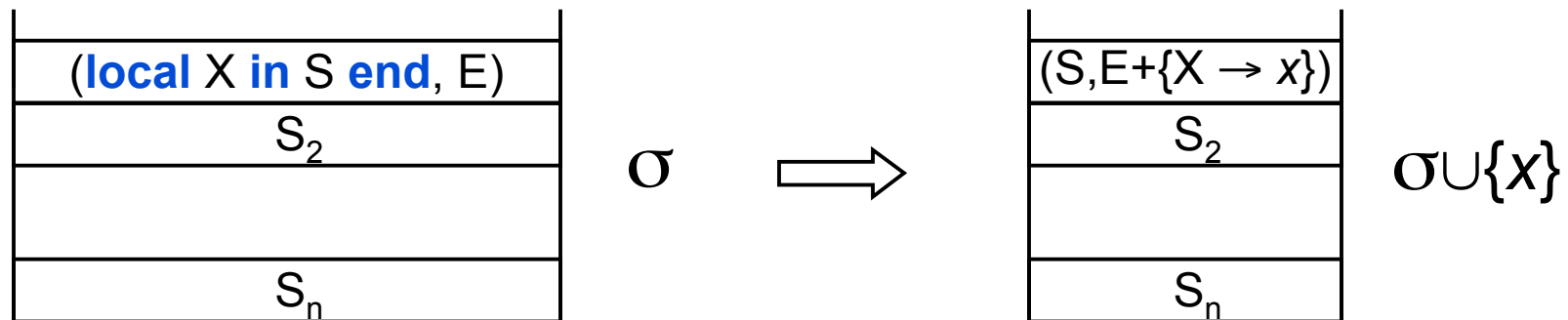
- Presque aussi simple
- L'instruction enlève le sommet de la pile et en ajoute deux nouveaux éléments





# local <x> in <s> end

- Créez la nouvelle variable  $x$
- Ajoutez la correspondance  $\{X \rightarrow x\}$  à l'environnement  $E$  (l'adjonction)

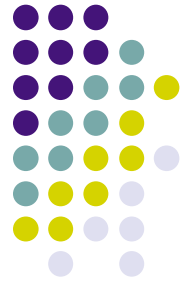




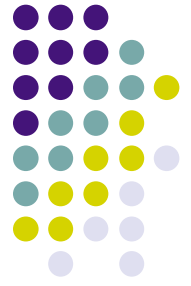
# Les autres instructions

- Elles ne sont pas vraiment compliquées
- A vous de regarder tranquillement chez vous
- $\langle x \rangle = \langle v \rangle$  (l'affectation)
  - **Attention**: quand  $\langle v \rangle$  est une procédure, il faut créer l'environnement contextuel!
- **if**  $\langle x \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end** (le conditionnel)
  - **Attention**: si  $\langle x \rangle$  n'a pas encore de valeur, l'instruction attend ("bloque") jusqu'à ce que  $\langle x \rangle$  est true ou false
  - La **condition d'activation**: " $\langle x \rangle$  est lié à une valeur"
- **case**  $\langle x \rangle$  **of**  $\langle p \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end**
  - **Attention**: les **case** plus compliqués sont construits à partir de ce cas simple

# 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



# L'environnement contextuel

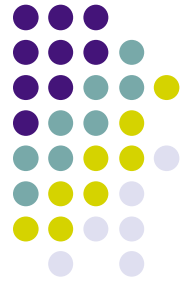
**local Z in**

**Z=1**

**proc {P X Y} Y=X+Z end**

**end**

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

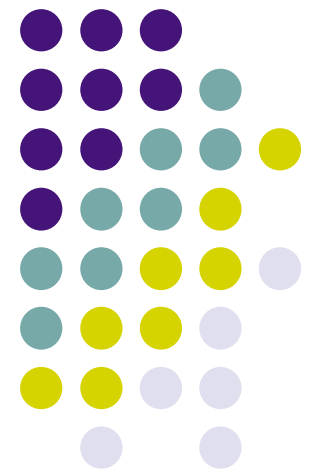


# Définition d'une procédure

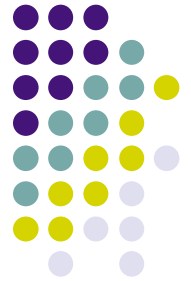
- L'instruction sémantique de la définition est  
 $(\langle x \rangle = \text{proc } \{ \$ \langle x \rangle_1 \dots \langle x \rangle_n \} \langle s \rangle \text{ end}, E)$
- Arguments formels:  
 $\langle x \rangle_1, \dots, \langle x \rangle_n$
- Identificateurs libres dans  $\langle s \rangle$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

# Exemple avec un appel de procédure

---



# L'instruction en langage noyau



```
local P in local Y in local Z in
```

```
  Z=1
```

```
  proc {P X} Y=X end
```

```
  {P Z}
```

```
end end end
```

# L'instruction en langage noyau



```
local P Y Z in
```

```
Z=1
```

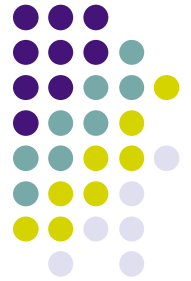
```
proc {P X} Y=X end
```

```
{P Z}
```

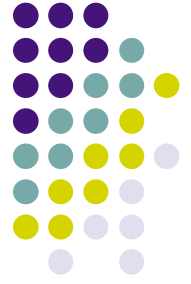
```
end
```

- Nous allons tricher un peu: nous allons faire toutes les déclarations en même temps
  - Ce genre de petite “tricherie” est très utile et on peut la justifier en ajoutant des règles de sémantique

# Petit exercice pour vous: un “local” plus pratique



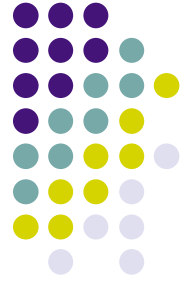
- On peut définir une version de **local** qui sait faire plusieurs déclarations à la fois
- Il faut modifier la sémantique
- Attention aux détails: **local** X X **in** ... **end**



# Début de l'exécution

```
((local P Y Z in  
  Z=1  
  proc {P X} Y=X end  
  {P Z}  
end,  $\emptyset$ ),  
 $\emptyset$ )
```

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



# Exemple avec procédure

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

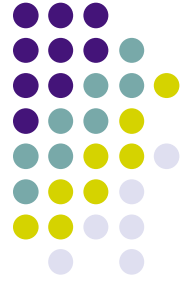
- Première instruction



# Début de l'exécution

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

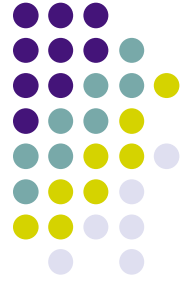
- Environnement initial (vide)



# Début de l'exécution

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

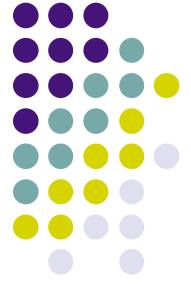
- Instruction sémantique



# Début de l'exécution

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

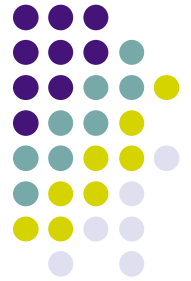
- Pile sémantique initiale (qui contient une instruction sémantique)



# Début de l'exécution

```
((local P Y Z in  
  Z=1  
  proc {P X} Y=X end  
  {P Z}  
end,  $\emptyset$ ),  
 $\emptyset$ )
```

- Mémoire initiale (vide)



# L'instruction "local"

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

- Créez les nouvelles variables en mémoire
- Étendez l'environnement



# L'instruction "local"

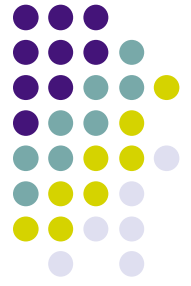
((Z=1

**proc** {P X} Y=X **end**

{P Z}, {P → p, Y → y, Z → z}),

{p, y, z})

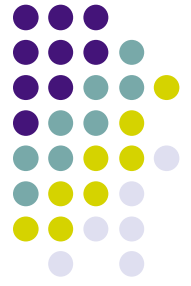
- Créez les nouvelles variables en mémoire
- Étendez l'environnement



# Composition séquentielle

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

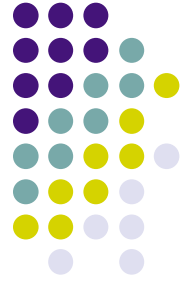
- Faites la première composition séquentielle



# Composition séquentielle

$((Z=1, \quad \{P \rightarrow p, Y \rightarrow y, Z \rightarrow z\}),$   
 $(\text{proc } \{P \ X\} Y=X \text{ end}$   
 $\{P \ Z\}, \quad \{P \rightarrow p, Y \rightarrow y, Z \rightarrow z\}),$   
 $\{p, y, z\})$

- Faites la première composition séquentielle

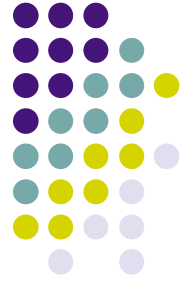


# Affectation de Z

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

- Liez la variable z en mémoire

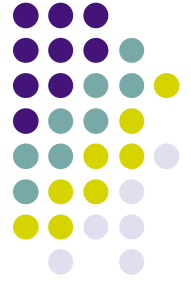




# Définition de la procédure

$((\text{proc } \{P \ X\} \ Y=X \ \text{end},$   
 $\{P \rightarrow p, Y \rightarrow y, Z \rightarrow z\}),$   
 $(\{P \ Z\}, \{P \rightarrow p, Y \rightarrow y, Z \rightarrow z\}),$   
 $\{p, y, z=1\})$

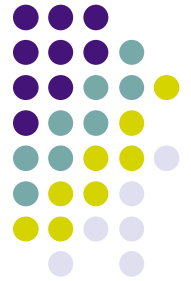
- Définition de la procédure
  - Identificateur libre  $Y$
  - Argument formel  $X$
- L'environnement contextuel est  $\{Y \rightarrow y\}$ 
  - Restriction de  $\{P \rightarrow p, Y \rightarrow y, Z \rightarrow z\}$
- Mettez ensuite la valeur procédurale dans la mémoire



# Appel de la procédure

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

- 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  $\{Y \rightarrow y\}$
  - Adjonction de  $\{X \rightarrow z\}$



# Appel de la procédure

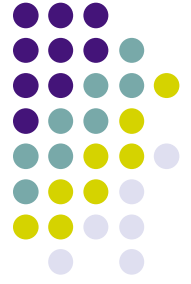
$([(Y=X, \quad \{Y \rightarrow y, X \rightarrow z\})],$   
 $\{ p = (\text{proc } \{ \$ X \} Y=X \text{ end}, \{Y \rightarrow y\}),$   
 $y, z=1\})$



# 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:  $z$



# Fin de l'exemple

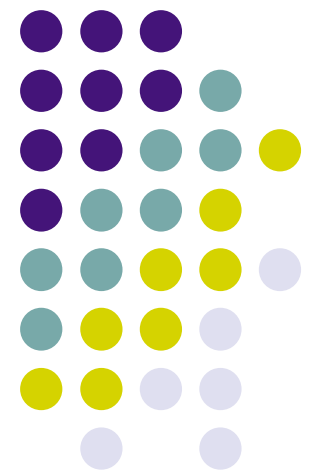
([],  
{  $p = (\text{proc } \{ \$ X \} Y=X \text{ end}, \{ Y \rightarrow y \}),$   
   $y=1, z=1 \}$ )

Voilà!

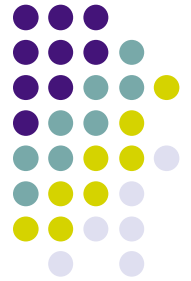
Qu'en pensez-vous?

# Les procédures en mémoire

---

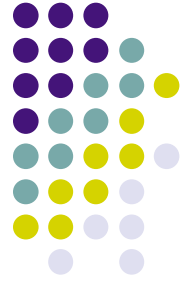


# Une procédure en mémoire est une valeur!

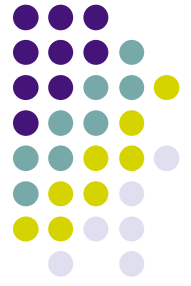


- Une procédure est stockée en mémoire, tout comme n'importe quelle valeur
    - Rappelez-vous que le langage noyau ne connaît que trois types de valeurs: nombres, enregistrements, procédures
- ```
proc {Inc X Y} Y=X+1 end  
Inc=proc {$ X Y} Y=X+1 end
```
- La valeur est “**proc {\$ X Y} Y=X+1 end**”
    - L'identificateur Inc, le “nom” de la procédure, ne fait pas partie de la valeur
    - Comment comprendre cela? Par analogie avec les entiers:  
**declare N=23**
    - La valeur est 23, l'identificateur N ne fait pas partie de la valeur
    - C'est pareil avec les valeurs procédurales

# Importance de la valeur procédurale

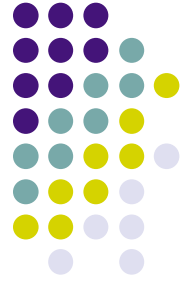


- Une valeur procédurale contient **deux choses**
  - Le code de la procédure et l'environnement contextuel
  - L'appel de la procédure construit l'environnement de l'appel: on commence avec l'environnement contextuel, en on y ajoute les arguments formels
- La valeur procédurale est **un des concepts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



# Appel de procédure (1)

- L'instruction sémantique est  
 $(\{\langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n\}, E)$
- 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



## Appel de procédure (2)

- L'instruction sémantique est

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

avec

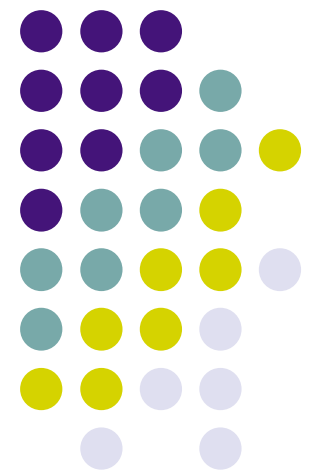
$$E(\langle x \rangle) = (\mathbf{proc} \{\$ \langle z \rangle_1 \dots \langle z \rangle_n\} \langle s \rangle \mathbf{end}, CE)$$

- Alors empilez

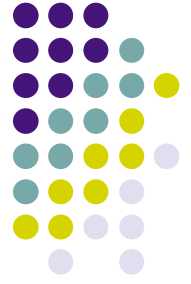
$$(\langle s \rangle, CE + \{\langle z \rangle_1 \rightarrow E(\langle y \rangle_1), \dots, \langle z \rangle_n \rightarrow E(\langle y \rangle_n)\})$$

# Résumé

---

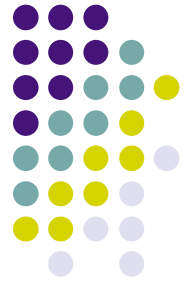


# Concepts de la machine abstraite

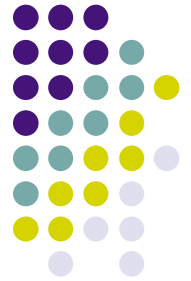


- **Mémoire à affectation unique**  $\sigma = \{x_1=10, x_2, x_3=20\}$ 
  - Variables et leurs valeurs
- **Environnement**  $E = \{X \rightarrow x, Y \rightarrow y\}$ 
  - Lien entre identificateurs et variables en mémoire
- **Instruction sémantique**  $(\langle s \rangle, E)$ 
  - Une instruction avec 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)

# Résumé



- Enregistrements, tuples, listes et arbres
  - Une liste est un tuple, un tuple est un enregistrement
  - Calcul orienté-but dans un arbre de recherche
- 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 des environnements
  - Adjonction, restriction
- Exemple d'exécution avec une procédure
  - Une procédure en mémoire est une valeur!
  - Une valeur procédurale contient **deux choses**: code + environnement contextuel
  - Définition et appel de procédure



# Il faut faire des exercices!

- La sémantique est la partie la plus formelle du cours
  - Pour bien la comprendre il faut faire beaucoup d'exercices
  - Il faut faire marcher le mécanisme
- Bien comprendre les exemples du livre:
  - Section 2.4.5: Exemples de base
  - Section 2.5.1: Pourquoi la récursion terminale marche
  - Section 3.3.1: Pourquoi la pile grandit avec la factorielle naïve
- J'attends de vous que vous puissiez faire marcher le mécanisme
  - Faire des pas de sémantique sur papier
  - Attention aux définitions et appels de procédure