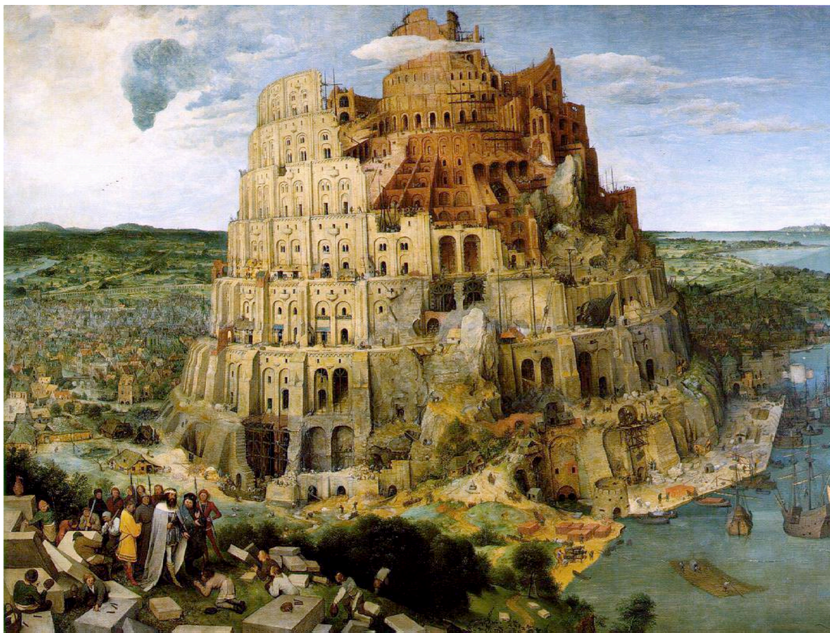
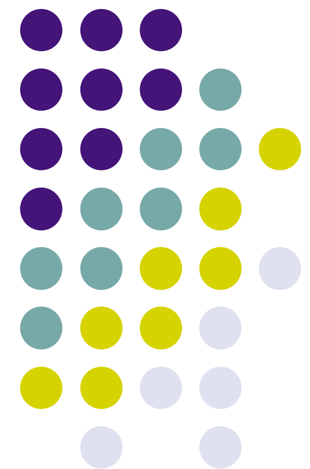
Les principaux paradigmes de programmation



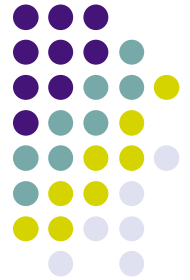
11 janvier 2008

UPMC

Peter Van Roy
Université catholique de Louvain
Louvain-la-Neuve, Belgium

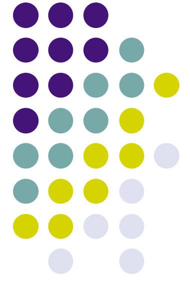


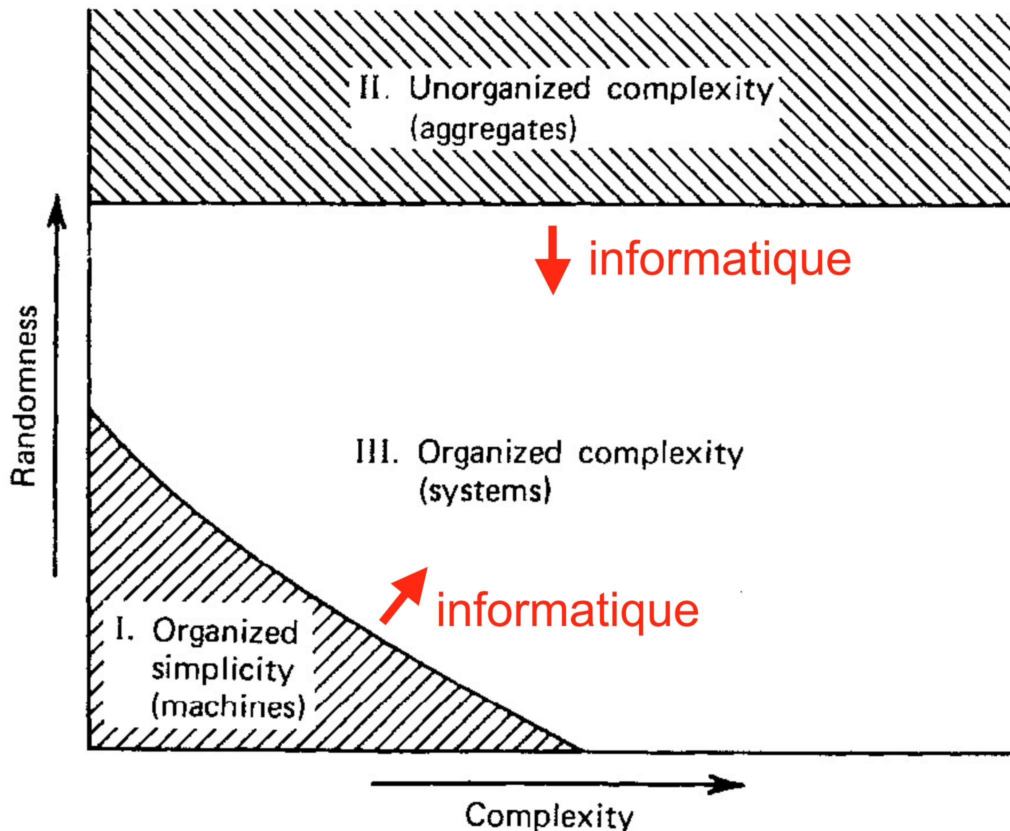
Les paradigmes de programmation



- Un paradigme est une manière de programmer un ordinateur basé sur un ensemble de principes ou une théorie
  - Différentes théories sur le calcul donnent différents paradigmes (λ calcul, π calcul, logique de premier ordre, science cognitive, …)
  - Aucune théorie “pré-existante” couvre tous les concepts de programmation!
  - La programmation est vraiment un domaine nouveau par rapport aux théories mathématiques classiques
- Nous allons explorer le monde des paradigmes
  - Nous allons donner un aperçu de chaque paradigme

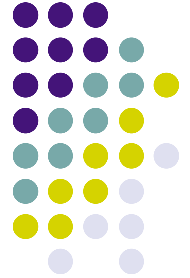
La programmation et la maîtrise des systèmes

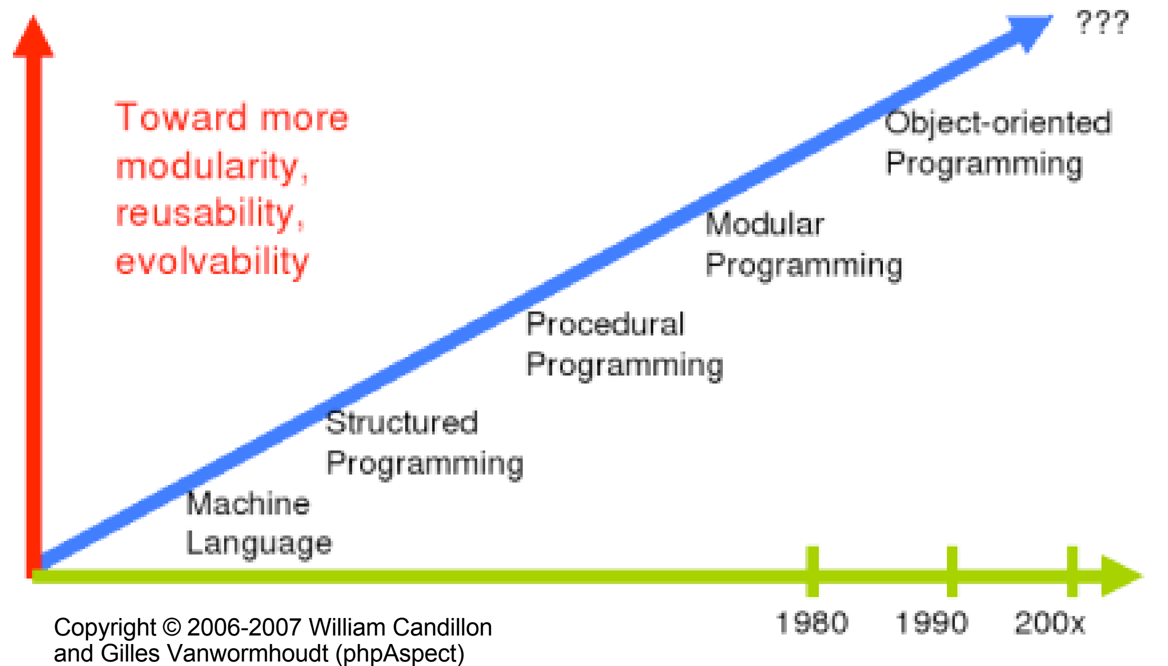




- Voici une vue générale du contexte du développement de l'informatique
- Ce diagramme vient de [Weinberg 1977] *An Introduction to General Systems Thinking*
- Les concepts de programmation permettent de construire des systèmes plus complexes
- Les paradigmes de programmation sont à l'avant-garde de la théorie des systèmes

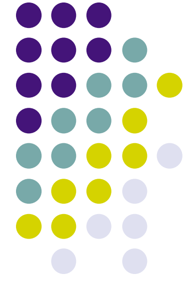
Le développement de la programmation (1)

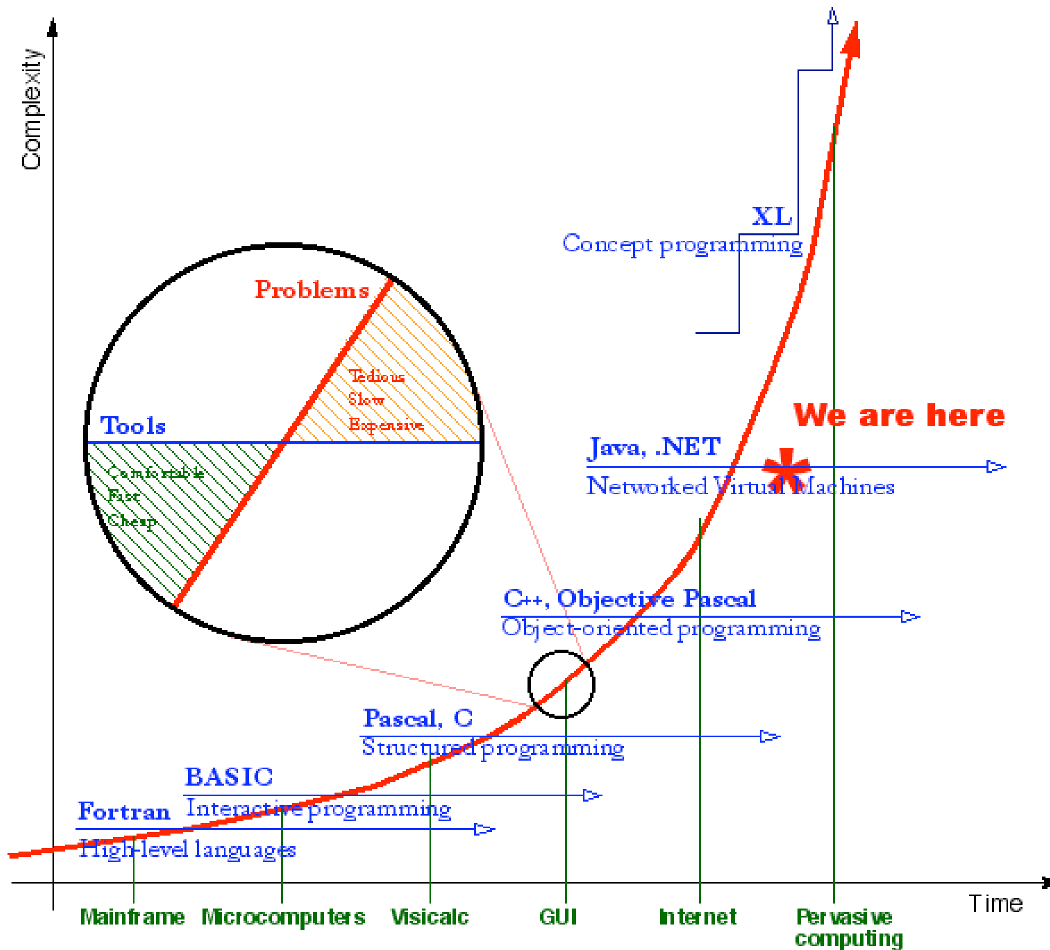




- Voici un diagramme typique qui illustre ce développement comme il est généralement vu
- Ce genre de diagramme ne prend en compte qu'une partie minime du développement des paradigmes, viz. le soutien pour l'encapsulation

Le développement de la programmation (2)

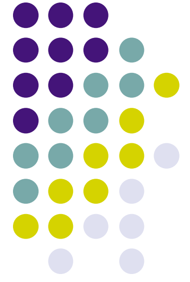




Copyright © 2004 Christophe de Dinechin (SourceForge.net)

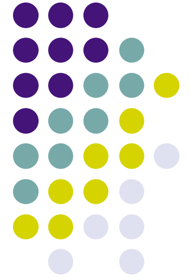
- Voici un autre diagramme un peu plus réaliste
- Chaque “paradigme” permet la programmation facile d’un certain niveau de interactivité
- Chaque nouveau paradigme augmente le niveau d’interactvitié qui est facile
- De nouveau, ce diagramme ne prend pas en compte qu’une petite partie du développement des paradigmes
  - Il y a au moins la programmation en réseau qui est mentionné, qui fait partie du paradigme de la programmation concurrente

Paradigmes et concepts

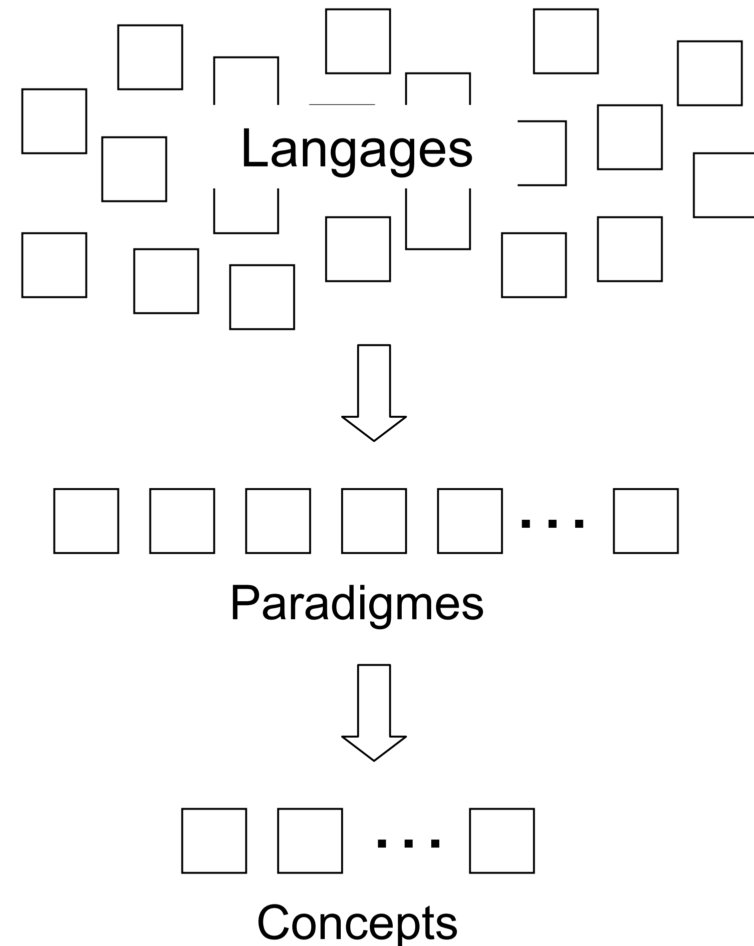


- Nous allons explorer le développement de la programmation dans le contexte des paradigmes
  - Il y a beaucoup moins de paradigmes que de langages de programmation
  - Mais il reste quand même beaucoup de paradigmes (il y en a au moins 29 effectivement utilisés)
- Heureusement, les paradigmes ne sont pas des îlots; ils ont beaucoup en commun
  - Chaque paradigme est défini par un ensemble de concepts de programmation
  - Souvent un seul concept peut faire un monde de différence
- Nous allons donc nous concentrer sur les concepts et comment ils influencent les paradigmes

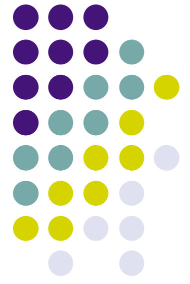
Étudier les concepts pour comprendre les paradigmes



- Au lieu de penser en termes de paradigmes, il est mieux de penser en termes de concepts de programmation
- Un paradigme = un ensemble de concepts
- Avec n concepts, on peut théoriquement construire 2^n paradigmes
  - n est beaucoup plus petit que 2^n
  - Nous allons donc regarder les concepts
- Nous allons organiser les concepts selon le principe d'extension créatrice



Il y a beaucoup de paradigmes



Programmation à continuations

Programmation impérative avec recherche

Programmation logique déterministe

Programmation transactionnelle

Programmation avec objets actifs

Programmation fonctionnelle de premier ordre

Programmation fonctionnelle paresseuse

Programmation fonctionnelle réactive (FRP)

Programmation relationnelle

Programmation fonctionnelle

Programmation impérative

Programmation déclarative descriptive

Programmation par contraintes concurrentes

Programmation dataflow multi-agent

Programmation par boucle d'événements

Programmation par contraintes

Programmation multi-agent

Programmation dataflow monotone

Programmation orientée objet séquentielle

Programmation orientée objet concurrente

Programmation dataflow non-monotone

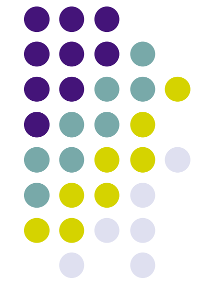
Programmation dataflow paresseuse

Programmation par contraintes paresseuses

Programmation fonctionnelle encapsulée (ADT)

Programmation par capacité objet

Taxonomie des paradigmes



Voici comment on peut organiser les paradigmes avec leurs concepts

enregistrement

Programmation déclarative descriptive

+ procédure

Programmation fonctionnelle de premier ordre

+ cellule

Programmation impérative

+ recherche

Programmation impérative avec recherche

+ fermeture

Programmation fonctionnelle

+ unification

Programmation logique déterministe

+ recherche

Programmation relationnelle

+ résolveur

Programmation par contraintes

+ fil

Programmation par contraintes concurrentes

+ synch besoin

Programmation par contraintes paresseuses

Programmation à continuations

+ synch. besoin

Programmation fonctionnelle paresseuse

+ fil
+ affect. unique

Programmation dataflow monotone

+ fil
+ affect. unique
+ synch. besoin

Programmation dataflow paresseuse

+ port

Programmation dataflow multi-agent

+ choix

Programmation dataflow non-monotone

+ synch. term.

Programmation fonctionnelle réactive (FRP)

+ nom

Programmation fonctionnelle encapsulée (ADT)

+ port

Programmation par boucle d'événements

+ fil

Programmation multi-agent

+ cellule locale

Programmation avec objets actifs

Programmation par capacité objet

+ cellule
+ fermeture

Programmation orientée objet séquentielle

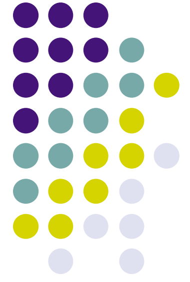
+ fil

Programmation orientée objet concurrente

+ journal

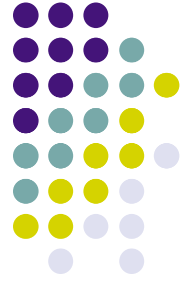
Programmation transactionnelle

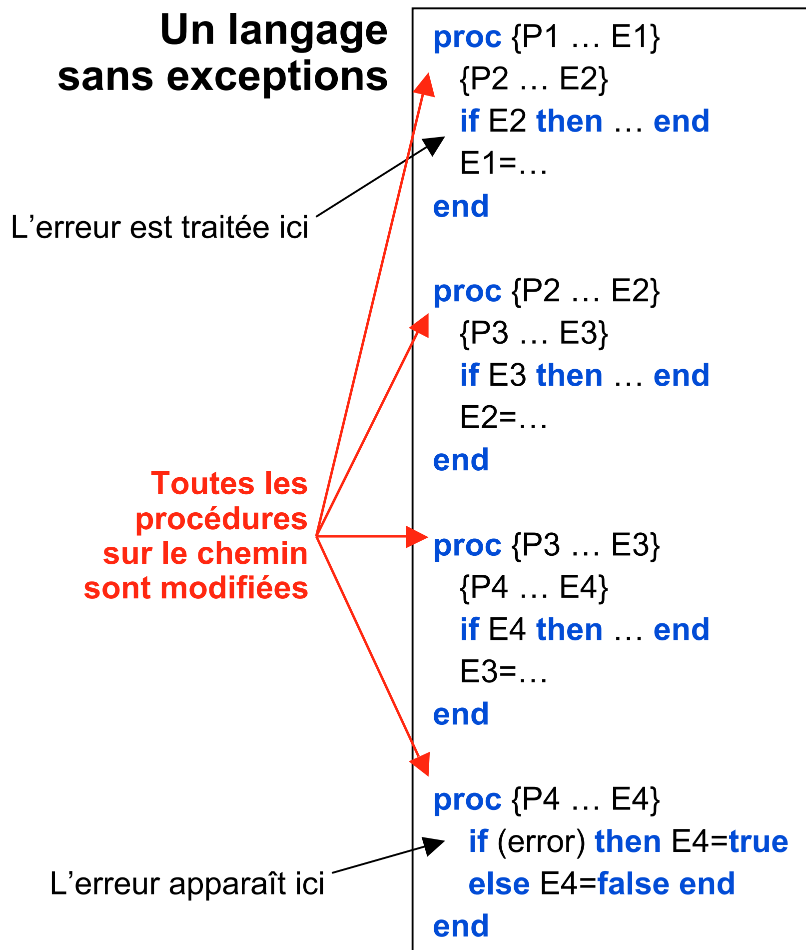
Le principe d'extension créatrice

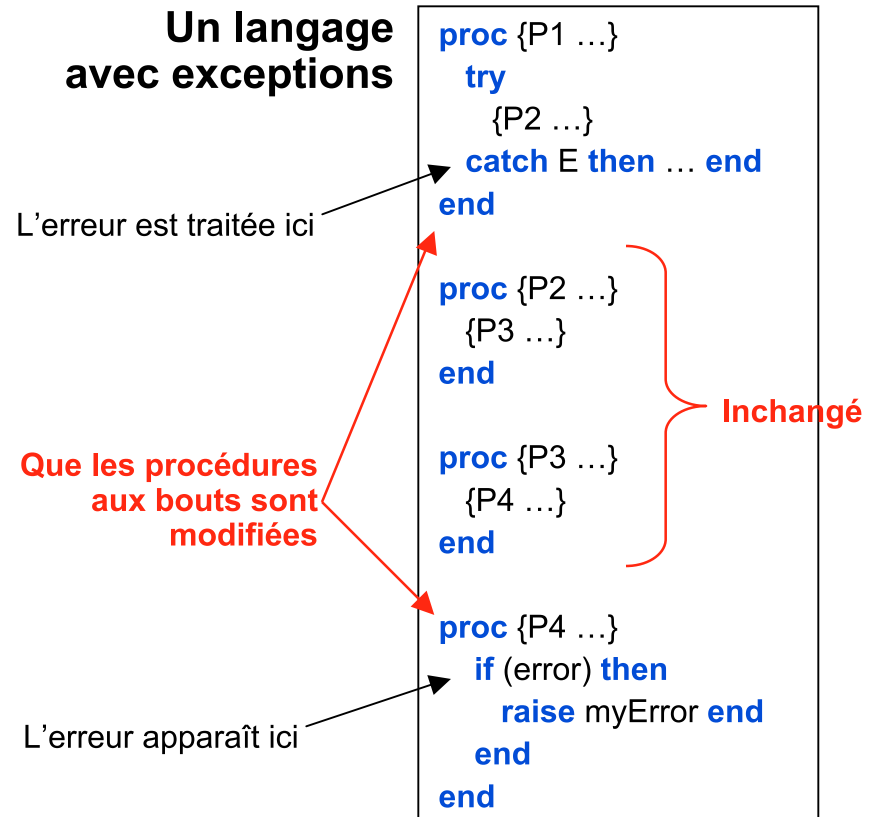


- Cette taxonomie est organisée selon le principe d'extension créatrice
  - Un principe général pour étendre un paradigme
  - Cela nous donne un plan pour explorer la jungle des paradigmes
- Dans un paradigme donné, quand les programmes deviennent compliqués pour des raisons techniques qui n'ont pas de relation avec le problème que l'on résoud (des transformations non-locales sont nécessaires), alors il y a un nouveau concept de programmation qui attend à être découvert
  - L'ajout de ce concept au paradigme récupère la simplicité (transformations locales)
- Un exemple typique est le concept des exceptions
  - Si le paradigme ne les a pas, toutes les routines sur le chemin de l'appel doivent tester et renvoyer des codes d'erreur (changements non-locaux)
  - Avec les exceptions, il ne faut changer que les bouts (changements locaux)
- Nous avons redécouvert ce principe dans nos recherches
  - Déjà défini par (Felleisen 1990)

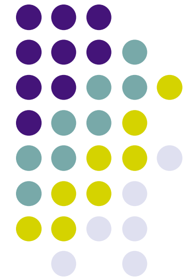
Un exemple du principe d'extension créatrice







Tous les concepts (jusqu’ici)



<s> ::=

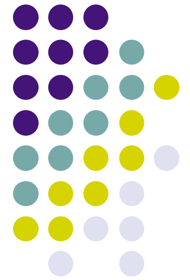
Syntaxe	Concept
skip	*Instruction vide*
<x>$_1$=<x>$_2$	*Lien de variable*
<x>=<record> | <number> | <procedure>	*Création de valeur*
<s>$_1$ <s>$_2$	*Composition séquentielle*
local <x> **in** <s> **end**	*Création de variable*
if <x> **then** <s>$_1$ **else** <s>$_2$ **end**	*Conditionnel*
case <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**	*Correspondance de formes*
{<x> <x>$_1$ … <x>$_n$}	*Invocation de procédure*
thread <s> **end**	*Création de fil*
{WaitNeeded <x>}	*Synchronisation par besoin*
{NewName <x>}	*Création de nom*
<x>$_1$= !!<x>$_2$	*Variable morte*
try <s>$_1$ **catch** <x> **then** <s>$_2$ **end**	*Contexte d’exception*
raise <x> **end**	*Lever une exception*
{NewPort <x>$_1$ <x>$_2$}	*Création de port*
{Send <x>$_1$ <x>$_2$}	*Envoi sur port*
<space>	*Contraintes*

Déclaratif descriptif (skip … local)

Déclaratif (if … WaitNeeded)

De moins en moins déclaratif

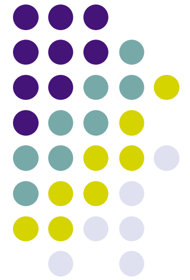
Tous les concepts (jusqu'ici)

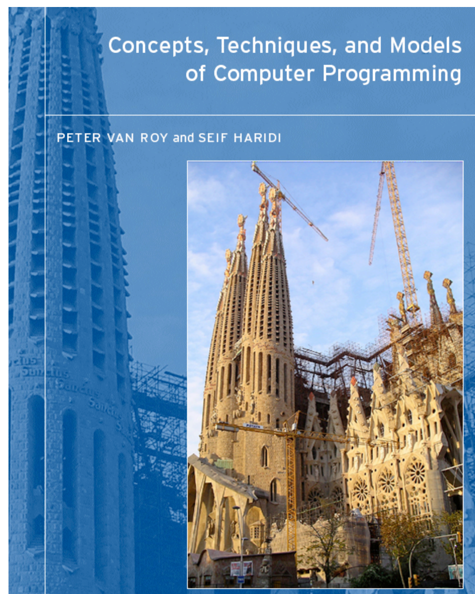


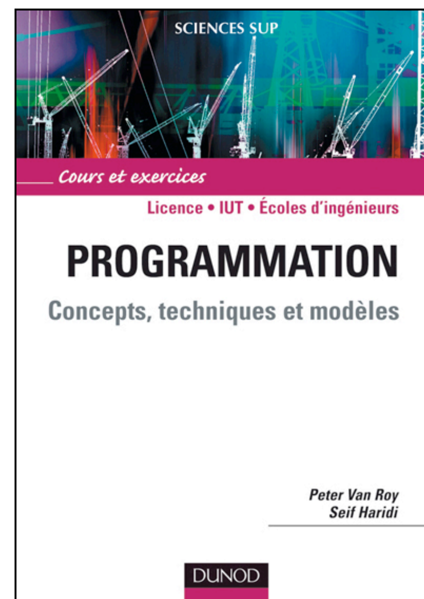
<s> ::=

skip	*Instruction vide*
<x>$_1$=<x>$_2$	*Lien de variable*
<x>=<record> | <number> | <procedure>	*Création de valeur*
<s>$_1$ <s>$_2$	*Composition séquentielle*
local <x> **in** <s> **end**	*Création de variable*
if <x> **then** <s>$_1$ **else** <s>$_2$ **end**	*Conditionnel*
case <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**	*Correspondance de formes*
{<x> <x>$_1$ … <x>$_n$}	*Invocation de procédure*
thread <s> **end**	*Création de fil*
{WaitNeeded <x>}	*Synchronisation par besoin*
{NewName <x>}	*Création de nom*
<x>$_1$= !!<x>$_2$	*Variable morte*
try <s>$_1$ **catch** <x> **then** <s>$_2$ **end**	*Contexte d'exception*
raise <x> **end**	*Lever une exception*
{NewCell <x>$_1$ <x>$_2$}	***Création de cellule*** } **Alternative**
{Exchange <x>$_1$ <x>$_2$ <x>$_3$}	***Échange de cellule*** } **Alternative**
<space>	*Contraintes*

Pour en comprendre plus…

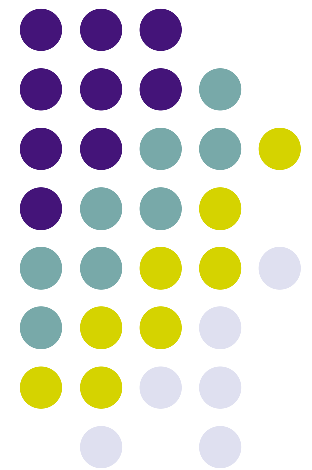




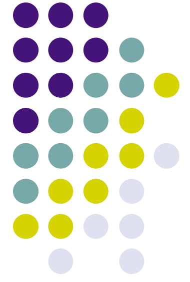


- Cette organisation des paradigmes est à la base du livre "Concepts, Techniques, and Models of Computer Programming", MIT Press, 2004
- Pour comprendre les détails, nous vous conseillons de consulter ce livre
- Une traduction française d'une partie du livre est sortie chez Dunod en 2007
- La version française est complémentée par un logiciel pédagogique, le Labo Interactif, édité par ScienceActive
- La version française et son logiciel sont à la base de mon cours de programmation de deuxième année à l'UCL

L'organisation d'un langage et de ces programmes

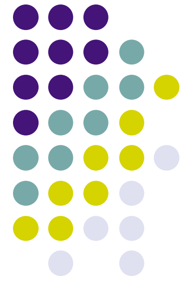


Quel paradigme choisir?



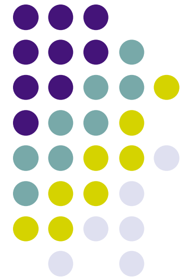
- Les frontières conventionnelles entre les paradigmes sont totalement artificielles
  - Elles sont là purement pour des raisons historiques
- Un bon programme a presque toujours besoin de plusieurs paradigmes
  - Chaque paradigme est bon pour certains problèmes
- Un bon langage doit donc soutenir plusieurs paradigmes!

Comment organiser un langage



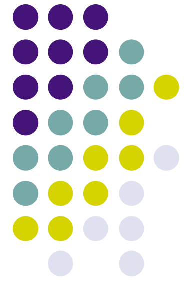
- Il faut pouvoir utiliser les bons concepts quand on en a besoin dans un programme
  - Il faut pouvoir utiliser plusieurs paradigmes dans un programme
- Comment organiser le langage pour permettre cela?
  - Quatre projets de recherche nous montrent le chemin
  - Chaque projet a conçu un langage pour résoudre un problème important
  - La surprise est que les structures des quatre langages sont semblables: une structure avec 3 ou 4 couches
- Une possibilité est donc d'utiliser une structure en couches…

Quatre projets de conception de langage



- La programmation de **systèmes à haute disponibilité** pour les télécommunications
  - Par Joe Armstrong *et al* au Ericsson Computer Science Laboratory
  - Conception du langage **Erlang** et son utilisation (e.g., AXD 301 ATM switch)
- La programmation des **systèmes répartis sécurisés** avec multiples utilisateurs dans multiples domaines de sécurité
  - Par Doug Barnes, Mark Miller et la communauté E
  - Conception du langage **E** ; modèle Actor de Hewitt et capacités de Dennis & Van Horn
- La programmation répartie **avec réseau transparent**
  - Par Seif Haridi, Peter Van Roy, Per Brand, Gert Smolka et leurs collègues
  - Conception du langage **Distributed Oz** et du système Mozart
- L'enseignement de la **programmation comme discipline unifiée** qui couvre tous les paradigmes de programmation populaires
  - Par Peter Van Roy et Seif Haridi
  - "Reconstruction" de **Oz** et un livre qui organise la programmation selon les concepts des paradigmes

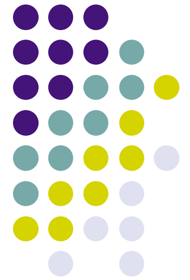
Comment organiser plusieurs paradigmes dans un langage



- Le langage a une structure en couches avec un noyau fonctionnel
- Cela permet d'utiliser chaque paradigme quand on en a besoin
- Le nombre de couches peut varier; voici quelques exemples à 3 ou 4 couches

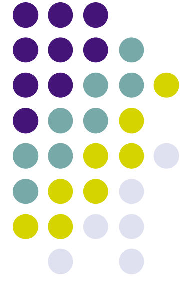
Erlang	E	Oz (répartition)	Oz (enseignement)	
Base de données pour la tolérance aux pannes (Mnesia)	—	État avec cohérence globale; transactions pour latence et tolérance aux pannes	État pour la modularité	**Concurrence par état partagé (var. affectable)**
Tolérance aux pannes par isolation; gestion de pannes	Messages entre objets dans "futs" différents, sécurité par isolation	Messages asynchrones pour cacher la latence entre processus; aucun état global	Facile à programmer et a enseigner	**Concurrence par envoi de messages (canal de commun.)**
—	"Boucle d'événements" dans un fut (processus): tous objets partagent un fil (pas d'entrelacement)	Concurrence dataflow avec protocole efficace d'unification répartie	La concurrence garde un raisonnement fonctionnel et permet des programmes multi-agents sans courses	**Concurrence déterministe**
Processus léger défini par une fonction, mise à jour "à chaud"	L'objet est une fonction récursive avec état local	Fonctions, classes et composants sont valeurs avec protocoles efficaces	Fermeture à portée lexicale à la base	**Fonctionnelle**

Des langages qui soutiennent *deux* paradigmes



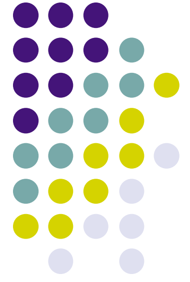
- Il y a beaucoup de langages qui soutiennent avec succès deux paradigmes
  - L'un pour faire les calculs, l'autre pour structurer le problème
  - Une version simple de la structure en couches
- Voici quelques exemples:
  - Prolog: le cœur est la programmation logique, l'autre est la programmation impérative
    - Prolog est un vieux langage; les développement récents sont les langages de modélisation basé sur algorithmes de recherche
  - Langages de modélisation (Comet, Numerica, …): le cœur est le moteur du modèle (par ex., contraintes, algorithmes de recherche locale,…), l'autre est la programmation orientée objet
  - SQL: le cœur est la programmation logique/relationnelle, l'autre est la programmation transactionnelle

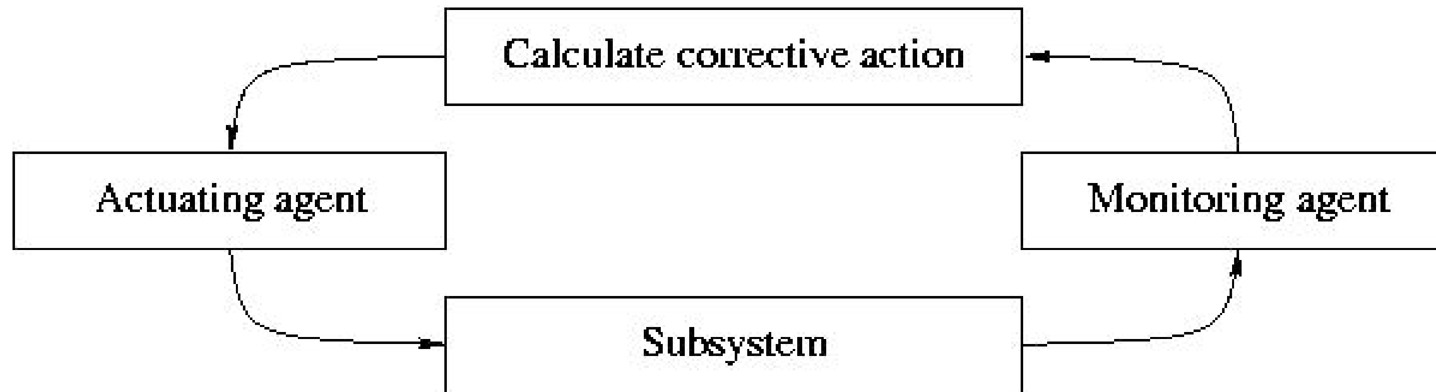
L'architecture des programmes à grande échelle



- Quelle est l'organisation des programmes à grande échelle?
  - Nous considérons qu'un programme doit pouvoir faire tout; aucune intervention humaine n'est nécessaire ni pour la maintenance ni pour adapter le système au monde externe; le programme a une durée de vie illimitée et permet du développement dans son sein (comme un système d'exploitation)
  - Comment est-ce qu'un tel système doit être organisé?
- Des composants concurrents qui communiquent par envoi de messages
- Les composants sont des entités de première classe qui permettent la programmation d'ordre supérieur
  - Important pour la reconfiguration et la maintenance du système
- Les composants sont organisés dans des boucles de rétroaction, pour permettre au programme de s'adapter à l'environnement externe
- Les composants sont organisés de façon décentralisée; le nombre de points centraux est minimisé

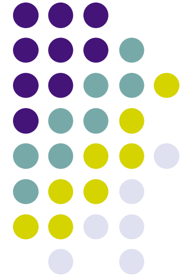
Organiser des composants en boucles de rétroaction





- Une boucle de rétroaction contient trois éléments qui interagissent avec un sous-système: un agent moniteur, un agent correcteur et un agent actuateur
- Des boucles de rétroaction peuvent interagir de deux manières:
  - Deux boucles qui influencent le même sous-système (stigmergie)
  - Une boucle qui contrôle une autre directement (gestion)

Le système respiratoire humain



Trigger unconsciousness when O2 falls to threshold

Render unconscious (and reduce CO2 threshold to base level)

Conscious control of body and breathing

Other inputs

Increase or decrease breathing rate and change CO2 threshold (maximum is breath–hold breakpoint)

Trigger breathing reflex when CO2 increases to threshold

Trigger laryngospasm temporarily when sufficient obstruction in airways

Breathing reflex

Laryngospasm (seal air tube)

Detect obstruction in airways

Measure CO2 in blood

Monitor breathing

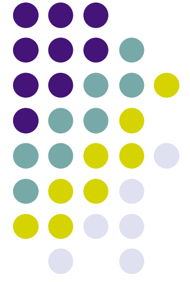
Measure O2 in blood

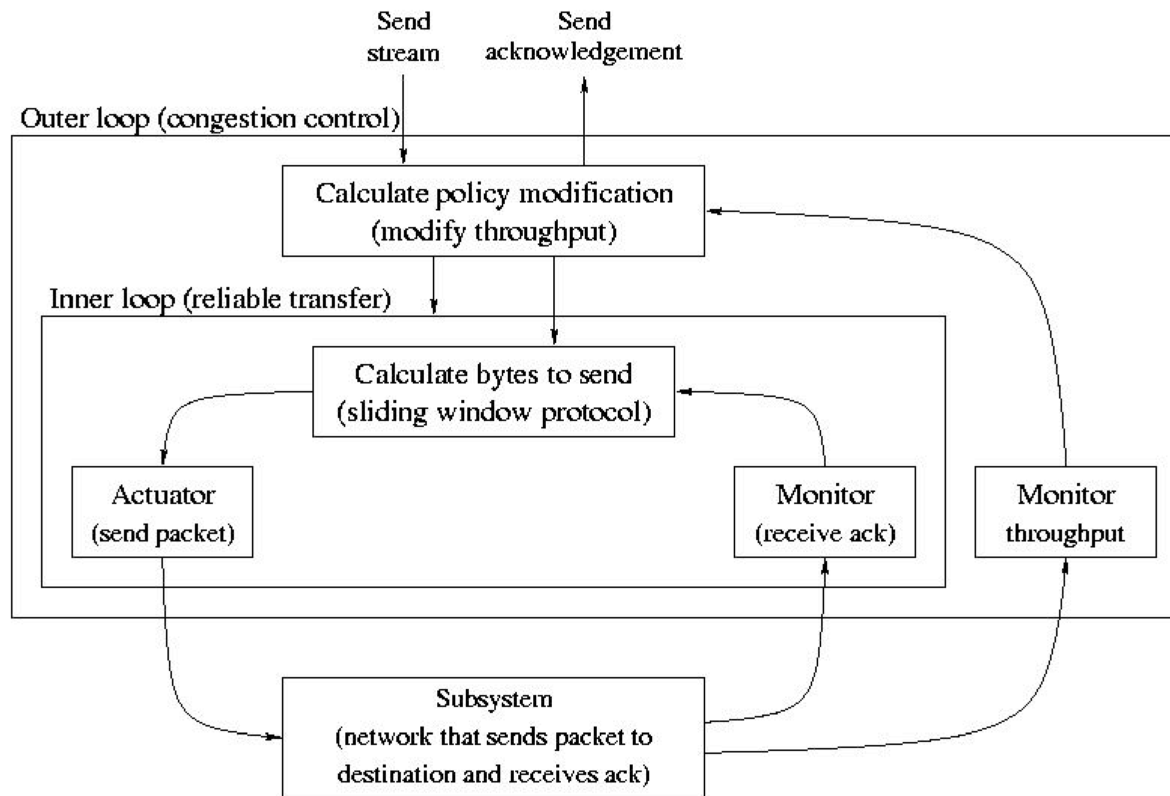
Breathing apparatus in human body

Actuating agents

Monitoring agents

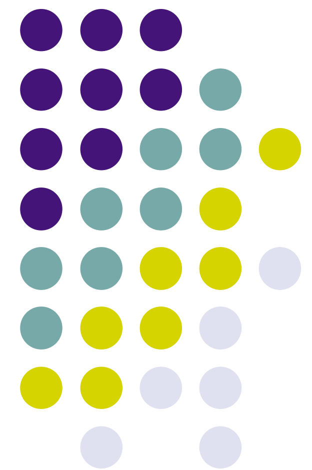
Conception de programmes avec boucles de rétroaction



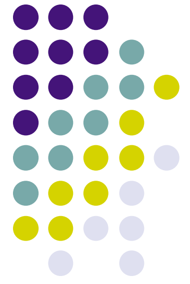


- Le style de conception de systèmes illustré par le système respiratoire peut être appliqué à la programmation
- La programmation fera des hiérarchies des boucles de rétroaction interagissantes.
- Cet exemple montre un protocole de flot d'octets avec contrôle de désengorgement (une variante de TCP)
- La boucle de contrôle de désengorgement gère la boucle de transfert fiable
  - En changeant la taille de la fenêtre glissante

Quelques concepts

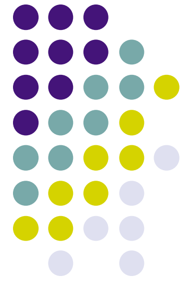


Quelques concepts et paradigmes intéressants

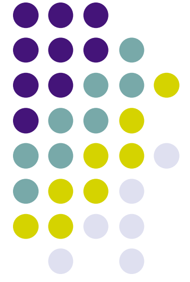


- Nous avons donné une vue générale des paradigmes et comment les organiser
- Maintenant nous allons descendre sur terre pour explorer quelques concepts et paradigmes interessants
- Pour créer un paradigme, à vous de prendre les concepts qui vous intéressent!
  - Un paradigme est simplement un ensemble de concepts qui sont cohérents pour la résolution d'un certain genre de problèmes

Les concepts majeurs



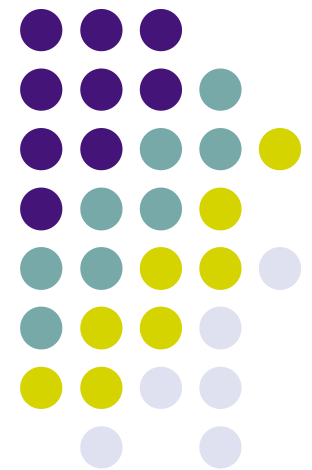
- L'enregistrement
- La fermeture
  - La continuation
  - L'exception
- L'indépendance
  - La concurrence
  - L'interaction: affectation unique, choix non-déterministe ou état partagé
- L'état
  - La variable affectable (1ère forme de l'état)
  - Le canal de communication (2ème forme de l'état)
- L'abstraction de données
  - L'encapsulation
  - Le polymorphisme
  - L'héritage
- La programmation orientée but
  - La programmation paresseuse
  - La programmation par contraintes



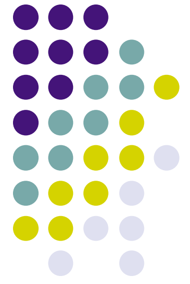
L’enregistrement

- Un enregistrement est un regroupement de données avec un accès direct à chaque donnée
  - R=chanson(nom:”Erlkönig” artiste:”Dietrich Fischer-Dieskau” compositeur:”Schubert”)
  - R.nom est égal à “Erlkönig”
- L’enregistrement est à la base de la programmation symbolique
  - Il faut pouvoir calculer avec des enregistrements: les créer, les décomposer, les examiner, tout pendant l’exécution
  - D’autres structures comme les listes, les chaînes et les arbres sont dérivées des enregistrements
  - Les enregistrements sont nécessaires pour soutenir d’autres techniques comme la programmation orientée objet, les interfaces graphiques et la programmation par composants
- Nous n’allons pas en dire plus à cause d’un manque de temps
  - Nous avons beaucoup d’autres concepts à explorer!

La fermeture

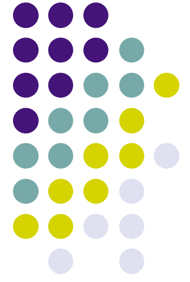


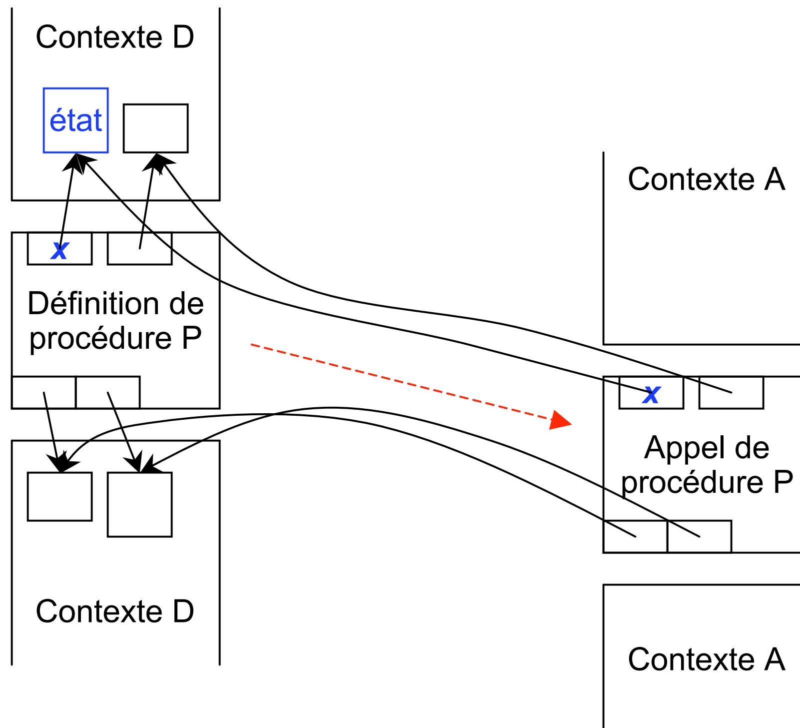
La fermeture



- S’il existe un concept qui est au cœur de la programmation, c’est certainement la fermeture à portée lexicale (“lexically scoped closure”)
  - Dans la taxonomie, on voit que la programmation fonctionnelle, qui en dépend, est un paradigme clé
- Du point de vue de l’implémentation, une fermeture regroupe une procédure avec ses références externes
- Du point de vue de l’utilisateur, une fermeture est un “paquet de travail”: on peut créer un paquet comme une valeur (une constante) à un endroit du programme, le donner à un autre endroit et décider de l’exécuter à cet endroit. Le résultat de son exécution est le même comme s’il on l’exécutait directement.

Une fermeture se souvient de l'endroit de sa création



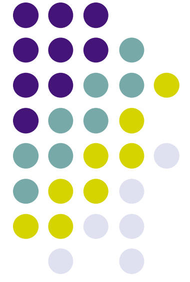


1. Définition

2. Appel

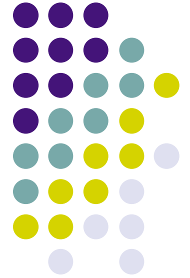
- La définition de P créé une fermeture: elle se souvient du contexte D de sa définition
- À l'appel de P, la fermeture utilise le contexte D
- Un objet est un exemple d'une fermeture: ***x*** fait référence à l'état de l'objet (un de ses attributs)

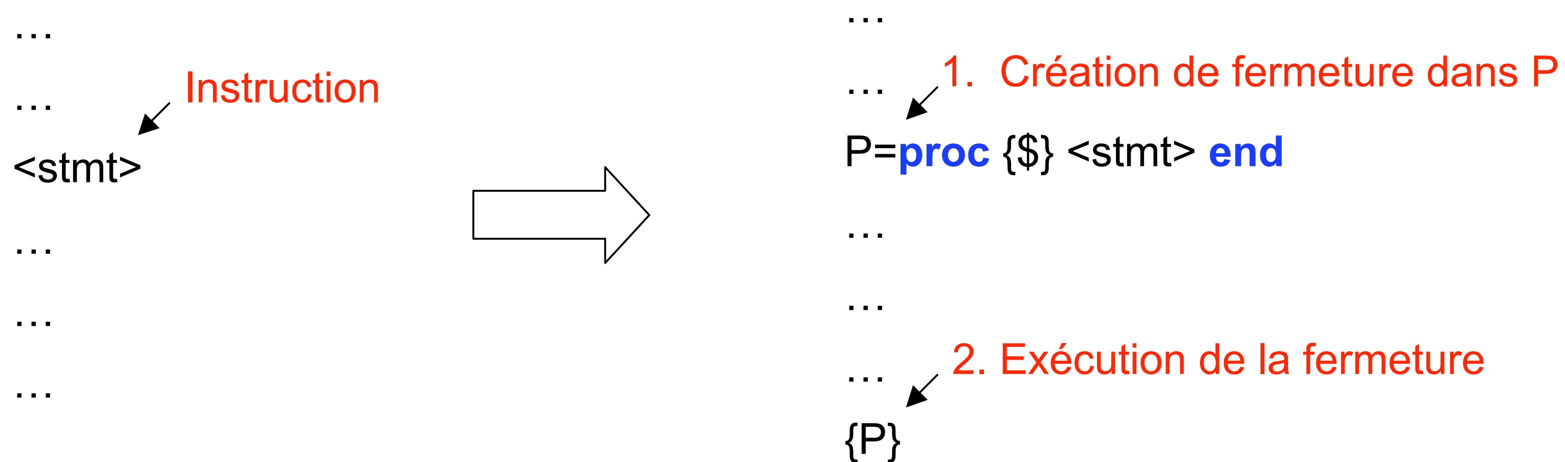
Les fermetures sont partout!



- Presque tous les langages de programmation (sauf quelques vénérables ancêtres comme le Pascal et le C) utilisent les fermetures
  - Les objets sont des fermetures
  - Les classes sont des fermetures
  - Les composants sont des fermetures
  - Les fonctions sont des fermetures
  - …
- Souvent, cette utilisation est cachée à l'intérieur et n'est pas disponible au programmeur
  - C'est dommage

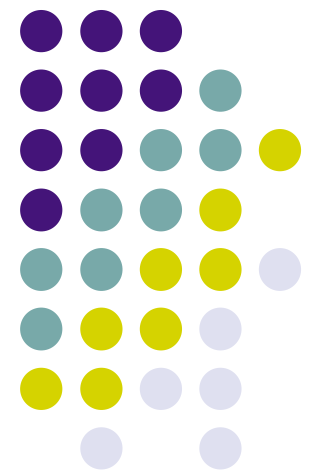
Les fermetures et les structures de contrôle



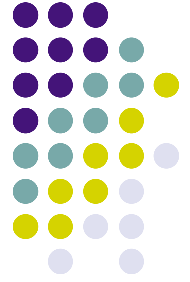


- On peut mettre une instruction dans une fermeture et décider plus tard de son exécution
- Cela permet de programmer toute structure de contrôle: boucles, conditionnels, etc.
- Un langage avec fermetures n'a pas besoin de structures de contrôle; on peut tout programmer avec les fermetures

L’indépendance

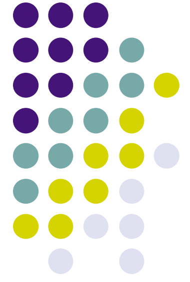


L’indépendance



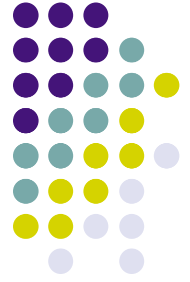
- Un autre concept clé est l’indépendance: comment construire un programme en parties indépendantes
- L’indépendance est à la base de plusieurs autres concepts dérivés:
  - La concurrence: les activités n’ont pas d’interaction
  - Le choix non-déterministe: les activités interagissent indirectement
  - L’état: les activités interagissent directement

La concurrence



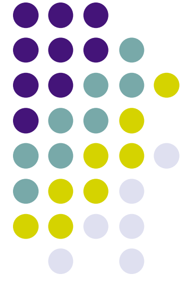
- Le monde réel est concurrent
  - Il est fait d'activités qui évoluent de façon indépendante
- Le monde informatique est concurrent aussi
  - Système réparti: ordinateurs liés par un réseau
    - Une activité concurrente s'appelle un **ordinateur**
  - Système d'exploitation d'un ordinateur
    - Une activité concurrente s'appelle un **processus**
    - Les processus ont des mémoires indépendantes
  - Plusieurs activités dans un processus
    - Typiquement, dans les browsers Web chaque fenêtre correspond à une activité!
    - Une activité concurrente s'appelle un **fil**
    - Les fils partagent la même mémoire

La concurrence dans un programme



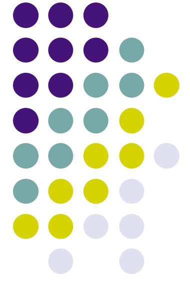
- La concurrence est naturelle
  - Deux activités qui sont indépendantes sont concurrentes!
    - Lien fort entre indépendance et concurrence
  - Qu'est-ce qu'on fait si on veut faire un programme qui fait deux activités indépendantes?
  - La concurrence doit être soutenue par les langages de programmation
- Un programme concurrent
  - Plusieurs activités s'exécutent simultanément
  - Les activités peuvent communiquer et synchroniser
    - Communiquer: les informations passent d'une activité à une autre
    - Synchroniser: une activité attend une autre

Quatre formes de programmation concurrente



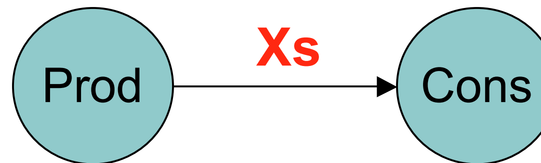
- À la base, deux activités concurrentes sont indépendantes
- La concurrence doit être associée avec un autre concept si les activités doivent interagir
- Cela donne plusieurs formes de programmation concurrente selon le choix de l'autre concept:
  - L'affectation unique: dataflow monotone
  - Le choix non-déterministe: dataflow non-monotone
  - Le canal de communication (une forme d'état): concurrence par envoi de messages
  - L'affectation multiple (une autre forme d'état): concurrence par état partagé

Dataflow monotone



- Producteur/consommateur

```
fun {Prod N Max}
  if N<Max then
    N|{Prod N+1 Max}
  else nil end
end
```

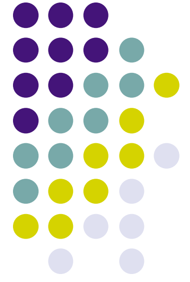



```
proc {Cons Xs}
  case Xs of X|Xr then
    {Display X}
    {Cons Xr}
  [] nil then skip end
end
```

```
local Xs in
  thread Xs={Prod 0 1000} end
  thread {Cons Xs} end
end
```

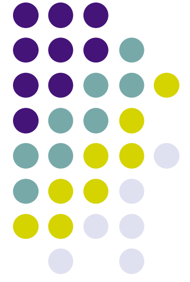
- Les fils de Prod et Cons se partagent la liste dataflow **Xs**
- Le comportement dataflow de l'instruction **case** (attendre la disponibilité des données) donne une communication par flot
- Aucun autre contrôle n'est nécessaire

Le choix non-déterministe

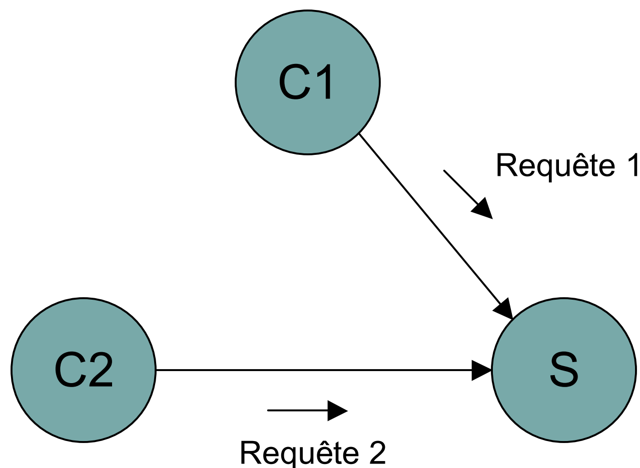


- Le non-déterminisme est nécessaire quand deux entités indépendantes interagissent avec une troisième entité
- Chacune des deux entités doit pouvoir interagir avec la troisième sans influencer ou être influencée de l'autre
- La troisième doit pouvoir recevoir des messages des deux autres indépendamment
  - La troisième fait un choix

Dataflow non-monotone



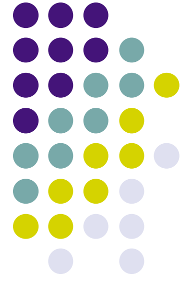
- Voici l'exemple typique du non-déterminisme
- Supposons que l'on a deux clients indépendants qui interagissent avec le même serveur



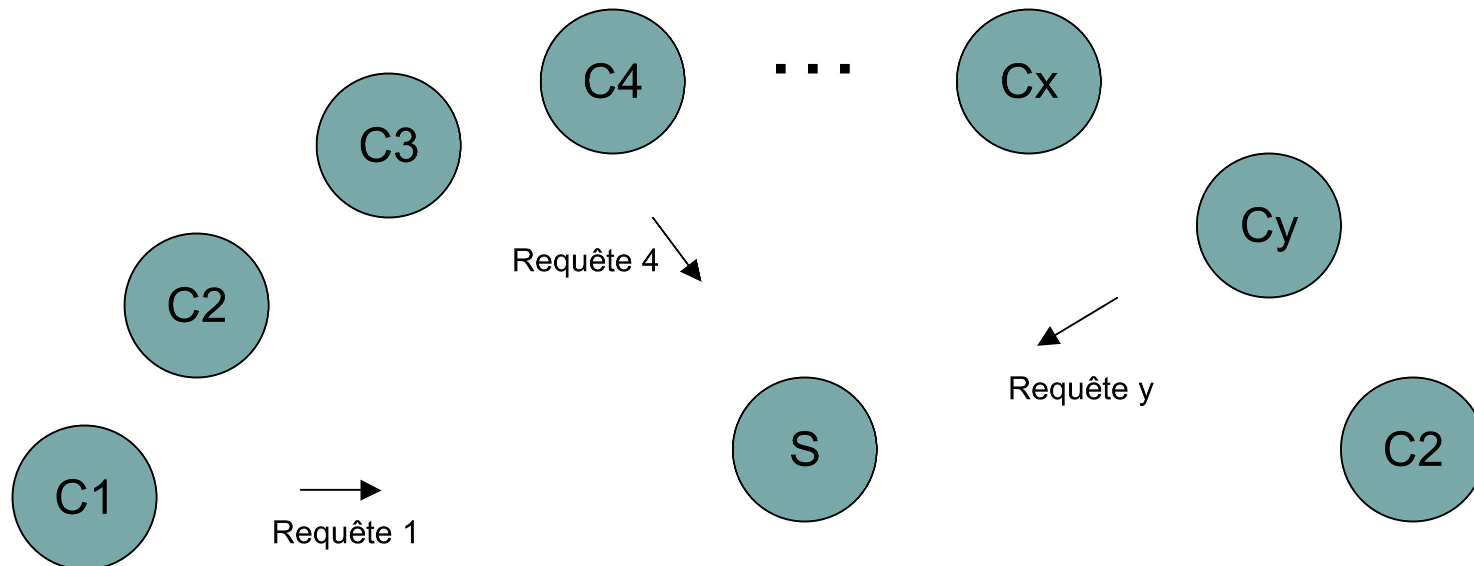
C1 et C2 sont indépendants; les requêtes peuvent donc arriver à S dans n'importe quel ordre. S connaît C1 et C2. Si deux requêtes arrivent en même temps, S doit pouvoir choisir un des deux.

(Dans le cas du dataflow monotone, S connaît aussi l'ordre d'arrivée des messages.)

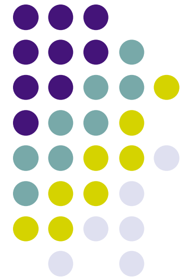
Concurrence par envoi de messages



- C'est comme le cas précédent, sauf que S ne connaît pas tous les clients
- N'importe quel client peut envoyer un message à S à n'importe quel moment, sans que S connaisse l'existence du client

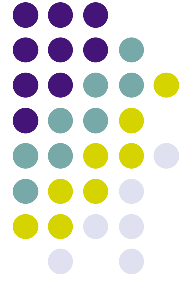


Propriétés



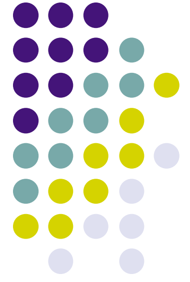
- Dataflow monotone: toute entité sait toujours d'où arrivera le prochain message
  - C'est aussi agréable que la programmation fonctionnelle: pas de courses ("race conditions"), une exécution déterministe comme un programme fonctionnel mais avec des résultats qui arrivent de façon incrémentale
  - C'est le paradigme préféré pour la programmation concurrente!
- Dataflow non-monotone: toute entité connaît toujours les entités qui peuvent lui envoyer les messages (l'entité doit pouvoir choisir: non-déterminisme)
  - Moins utile que les deux autres, sauf dans le cas de la Programmation Fonctionnelle Réactive qui récupère le comportement déterministe
- Concurrence par envoi de messages: une entité ne sait pas en général d'où arrivera le prochain message
  - Une entité peut ne pas connaître l'existence de l'expéditeur avant l'arrivée du message
  - C'est le paradigme de la programmation multi-agent
  - On essaie toujours d'utiliser le dataflow monotone le plus possible, et l'envoi de messages uniquement quand c'est nécessaire

Deux formes d'état



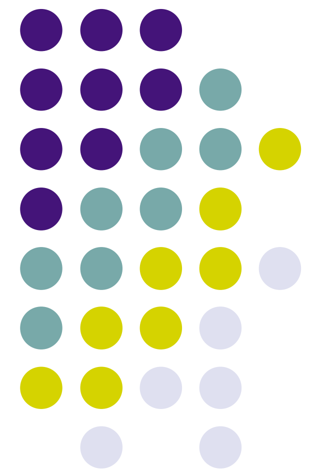
- Il y a deux formes principales de l'état
  - La variable affectable (comme en Java)
  - Le canal de communication (comme en Erlang)
- Le canal de communication est une forme d'état
  - Variables affectables et canaux de communication sont théoriquement équivalents en expressivité (chacun peut être codé avec l'autre), *mais…*
  - Le choix lequel fait partie du noyau (lequel est utilisé par défaut) a un énorme effet pour la facilité de programmer certaines choses
- Nous les considérons séparément
  - Le canal de communication donne la concurrence par envoi de messages
  - La variable affectable donne la concurrence par état partagé

Concurrence par état partagé

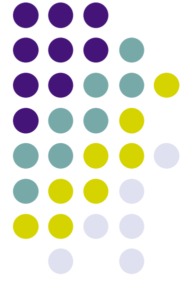


- On n'en a pas encore parlé, mais on ne l'a pas oublié
- C'est la forme la plus répandue de la concurrence
  - La plupart des langages courants ("mainstream") soutiennent cette forme de concurrence
  - Par exemple, en Java un objet peut être synchronisé: l'objet peut alors contrôler quel fil l'exécute
    - L'objet est un moniteur ("monitor")
- Cette form de concurrence n'est pas recommandée
  - Les moniteurs sont difficiles à utiliser (éviter l'interblocage est difficile, pas compositionnelle)
  - Il faut plutôt utiliser les transactions

L'état

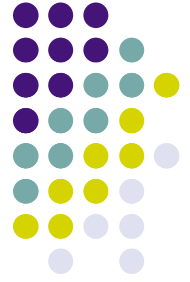


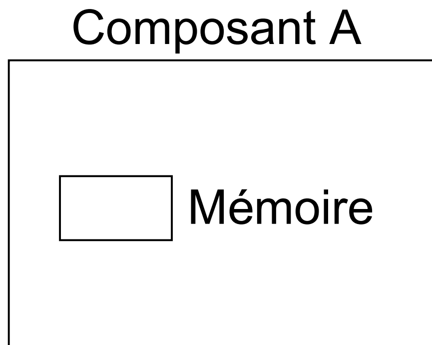
La notion de temps

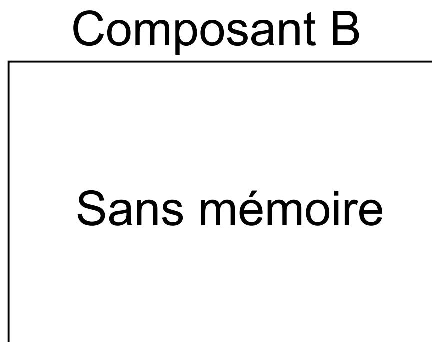


- Dans la programmation fonctionnelle, il n'y a pas de temps
  - Les fonctions sont des fonctions mathématiques, qui ne changent jamais
- Dans le monde réel, il y a le temps et le changement
  - Les organismes changent leur comportement avec le temps, ils grandissent et ils apprennent
  - Comment est-ce qu'on peut modéliser ce changement dans un programme?
- On peut ajouter une notion de temps abstrait dans les programmes
  - Temps abstrait = une séquence de valeurs
  - Un état = un temps abstrait = une séquence de valeurs

L’état: un concept à double tranchant

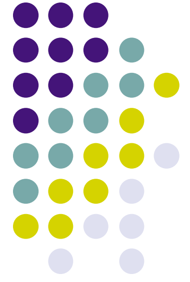






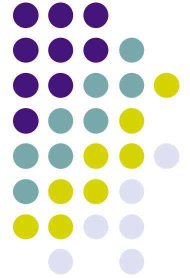
- L’état est une mémoire interne à un composant
- Sans état, le composant gardera toujours le même comportement
  - Comme une personne cérébrolésée sans nouveaux souvenirs à long terme; elle vit dans un éternel “présent”
  - Un composant correct le restera pour toujours
  - Un composant ne peut pas être mis à jour
- Avec un état, le composant peut changer son comportement
  - Un composant peut s’adapter à son environnement et grandir
  - Un composant peut devenir fautif

L’état est bénéfique pour la modularité



- On dit qu’un système (ou programme) est modulaire si des mises à jour dans une partie du système n’obligent pas de changer le reste
  - Partie = fonction, procédure, composant, …
- Nous allons vous montrer un exemple comment l’utilisation de l’état explicite nous permet de construire un système modulaire
  - Dans un paradigme sans état ce n’est pas possible

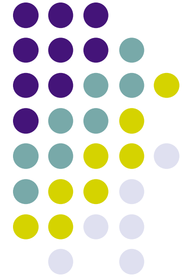
Scénario de développement (1)



- Il y a trois personnes, P, U1 et U2
- P a développé le module M qui offre deux fonctions F et G
- U1 et U2 sont des développeurs qui ont besoin du module M

```
fun {MF}
   fun {F ...}
      ⟨Définition de F⟩
   end
   fun {G ...}
      ⟨Définition de G⟩
   end
in 'export'(f:F  g:G)
end
M = {MF}
```

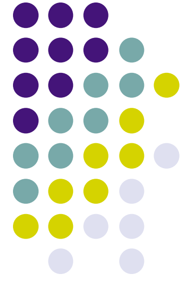
Scénario de développement (2)



- Développeur U2 a une application très coûteuse en temps de calcul
- Il veut étendre le module M pour compter le nombre de fois que la fonction F est appelée par son application
- Il va voir P et lui demande de faire cela sans changer l'interface de M

```
fun {MF}
   fun {F ...}
      ⟨Définition de F⟩
   end
   fun {G ...}
      ⟨Définition de G⟩
   end
in 'export'(f:F  g:G)
end
M = {MF}
```

Dilemme!

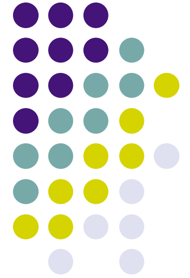


- Ceci est impossible dans un paradigme sans état parce que F ne se souvient pas de ses appels précédents!
- La seule solution est de changer l'interface de F en ajoutant deux arguments Fin et Fout:

  fun {F … Fin Fout} Fout=Fin+1 … **end**

- Le reste du programme doit s'assurer que la sortie Fout d'un appel de F soit l'entrée Fin de l'appel suivant de F
  - Des changements sont nécessaires partout
- Mais l'interface de M a changé
  - Tous les utilisateurs de M, même U1, doivent changer leur programme

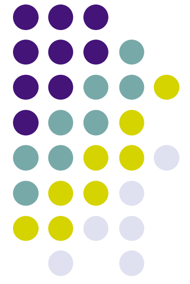
Solution avec l'état explicite



- Créez une cellule quand MF est appelé
  - Une cellule = une variable affectable
- A cause de la portée lexicale, la cellule X est cachée du programme: elle n'est visible que dans le module M
- M.f n'a pas changé
- Une nouvelle fonction M.c est disponible

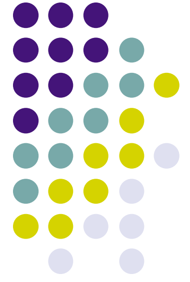
```
fun {MF}
   X = {NewCell 0}
   fun {F ...}
      X:=@X+1
      ⟨Définition de F⟩
   end
   fun {G ...} ⟨Définition de G⟩
   end
   fun {Count} @X end
in 'export'(f:F  g:G c:Count)
end
M = {MF}
```

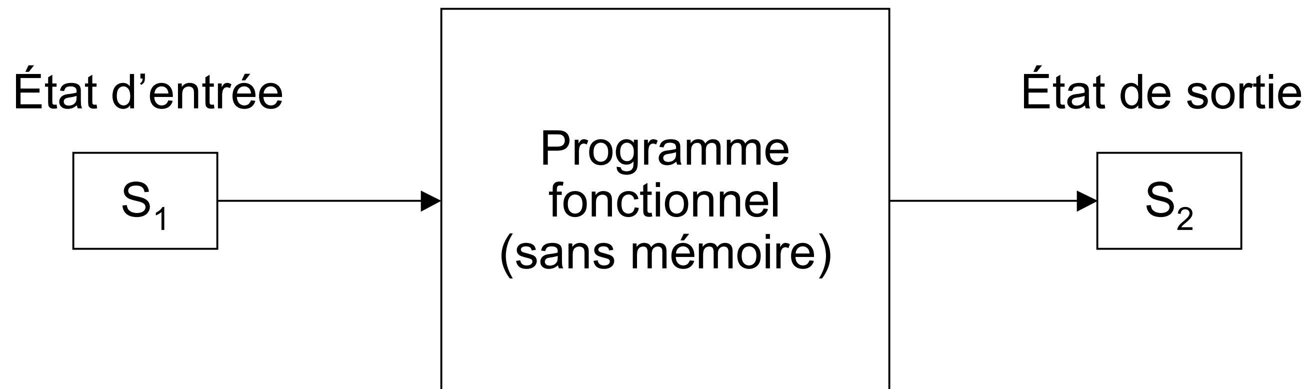
État et modularité



- En résumé, l'avantage principal d'avoir un état est de rendre le programme modulaire
  - Un programme est modulaire si on peut changer une partie du programme sans devoir changer le reste
- L'inconvénient principal est qu'un programme peut se planter facilement
  - L'état d'une partie du programme peut devenir erroné
  - C'est extrêmement difficile à tracer et à corriger
- Deux solutions
  - Concentrer l'état dans une partie du programme seulement et programmer le reste sans état (une machine à état)
  - Garder d'anciens états et utiliser un comportement transactionnel

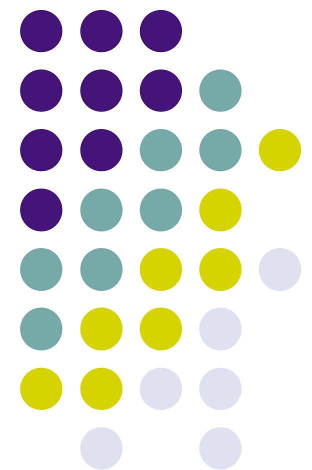
Maîtriser l'état



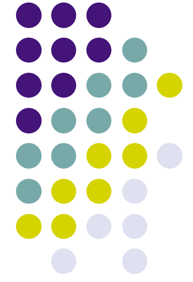


- Un programme est un transformateur d'état
- L'état est concentré dans un petit nombre d'endroits

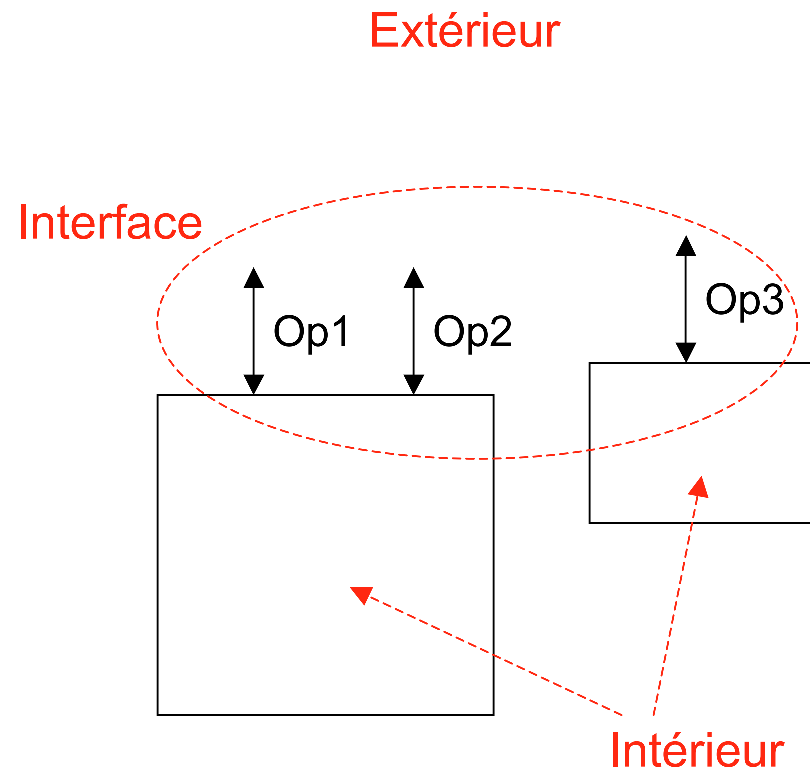
L’abstraction de données



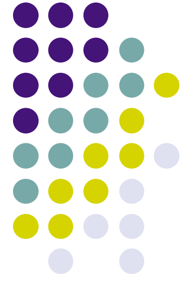
L'abstraction de données



- Une abstraction de données est une manière d'organiser l'utilisation des données selon des règles précises
- Une abstraction de données a un intérieur, un extérieur et une interface entre les deux
- L'intérieur est caché de l'extérieur
  - Toute opération sur l'intérieur doit passer par l'interface
- Cette encapsulation peut avoir un soutien du langage
  - Sans soutien est parfois bon pour de petits programmes
  - Nous allons voir comment le langage peut soutenir l'encapsulation, c'est-à-dire faire respecter l'encapsulation

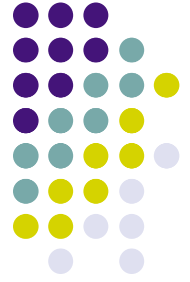


Avantages de l’encapsulation



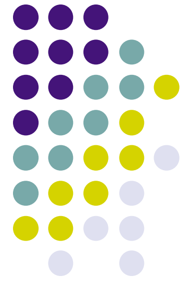
- La garantie que l’abstraction marchera toujours
  - L’interface est bien définie (les opérations autorisées)
- La réduction de complexité
  - L’utilisateur de l’abstraction ne doit pas comprendre comment l’abstraction est réalisée
  - Le programme peut être partitionné en beaucoup d’abstractions réalisées de façon indépendante, ce qui simplifie de beaucoup la complexité pour le programmeur
- Le développement de grands programmes devient possible
  - Chaque abstraction a un responsable: la personne qui l’implémente et qui garantie son comportement
  - On peut donc faire des grands programmes en équipe
  - Il suffit que chaque responsable connaît bien les interfaces des abstractions qu’il utilise

Organiser les abstractions de données



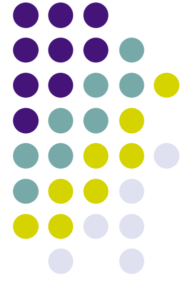
- Une bonne manière d'organiser un programme est comme un ensemble d'abstractions de données qui interagissent
- Il y a au moins quatre manières pour organiser une abstraction de données
  - Selon deux axes: l'agrégation et l'état
- L'agrégation
  - Opérations et valeurs fusionnées en une entité
  - Opérations et valeurs comme entités séparées
- L'état
  - Avec état
  - Sans état

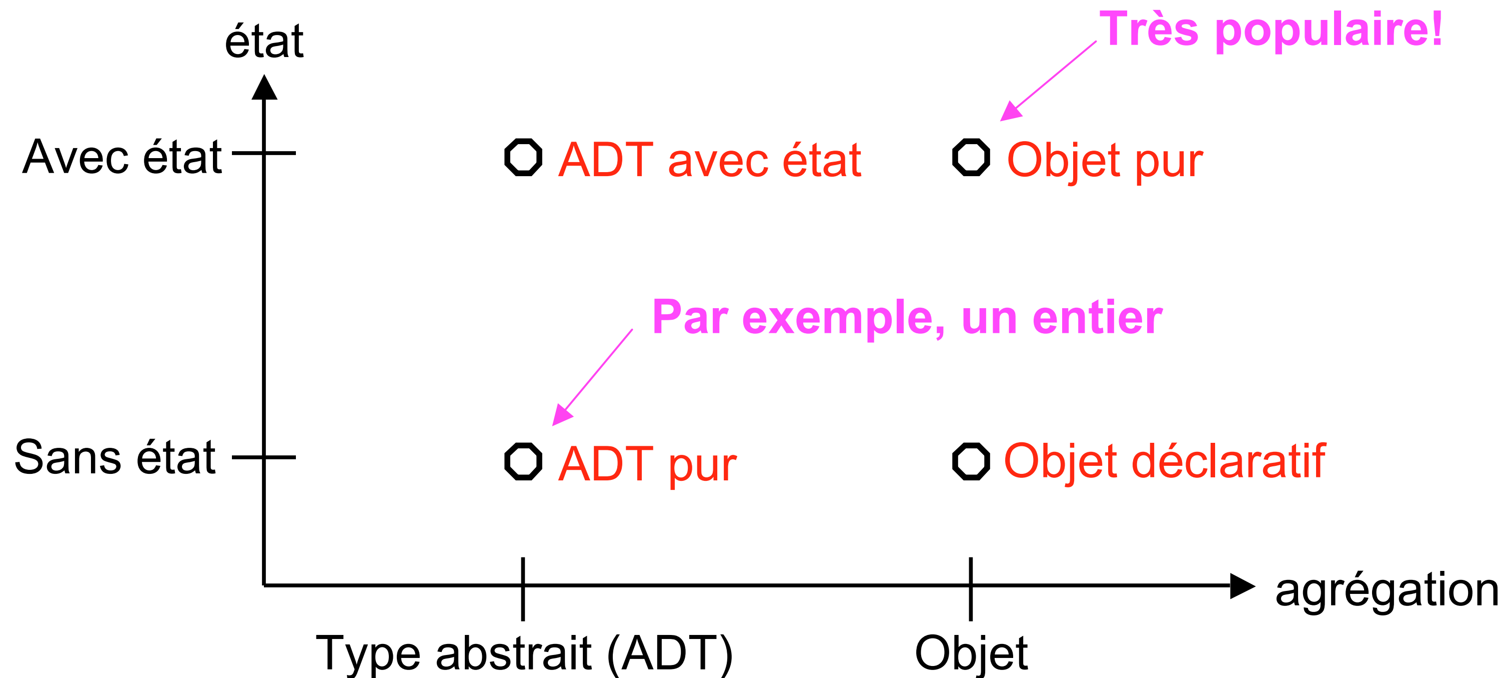
Les objets et les types abstraits



- Le premier axe est l'agrégation
- Un type abstrait (ADT) fait une séparation entre valeurs et opérations
  - Par exemple: les entiers (valeurs: 1, 2, 3, …; opérations: +, -, *, div, …)
  - Déjà investigué dans les années 1970 (le langage CLU par Barbara Liskov et al)
- Un objet fait la fusion des valeurs et opérations dans une seule entité
  - Par exemple: un objet pile (avec instances push, pop, …)
  - Déjà investigué dans les années 1960 (Simula 67) et ensuite dans Smalltalk (années 1970) et C++ (années 1980)
  - Actuellement très populaire (Java, C#, …)

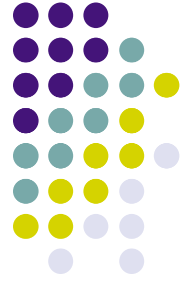
Résumé des abstractions de données





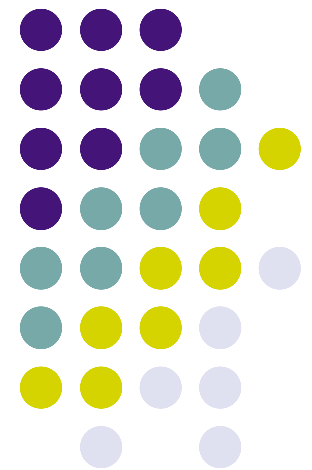
- Deux des quatre possibilités sont actuellement très populaires
- Mais il ne faut pas oublier les deux autres!

Les objets ont-ils gagné?

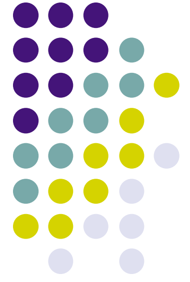


- Absolument pas! Les langages orientés objet purs font en réalité un savant mélange entre objets et ADT
  - Par exemple, en Java:
    - Les types primitifs comme les entiers sont des ADT (pas besoin de s'excuser!)
    - Les instances d'une même classe peuvent accéder à leurs attributs privés (c'est une propriété ADT)
- Il faut comprendre les différences entre objets et ADT
  - Les ADT permettent une implémentation efficace, qui n'est pas toujours possible avec d'objets purs (même Smalltalk utilise des ADT pour cette raison!)
  - Le polymorphisme et l'héritage marchent pour les deux, mais sont plus faciles avec les objets

Le polymorphisme

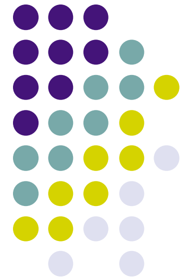


Le polymorphisme



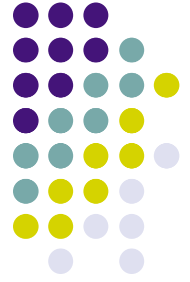
- Dans le langage de tous les jours, une entité est polymorphe si elle peut prendre des formes différentes
- Dans le contexte de l'informatique, une opération est polymorphe si elle peut prendre des arguments de types différents
- Cette possibilité est importante pour que les responsabilités soient bien réparties sur les différentes parties d'un programme

Le principe de la répartition des responsabilités



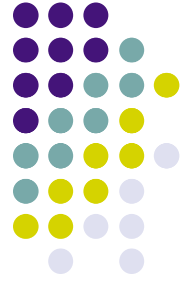
- Le polymorphisme permet d'isoler des responsabilités dans les parties du programme qui les concernent
  - En particulier, une responsabilité doit de préférence être concentrée dans une seule partie du programme
- Exemple: un patient malade va chez un médecin
  - Le patient ne devrait pas être médecin lui-même!
  - Le patient dit au médecin: "guérissez-moi"
  - Le médecin fait ce qu'il faut selon sa spécialité
- Le programme "guérir d'une maladie" est polymorphe: il marche avec toutes sortes de médecins
  - Le médecin est un argument du programme
  - Tous les médecins comprennent le message "guérissez-moi"

Réaliser le polymorphisme



- Toutes les formes d'abstraction de données soutiennent le polymorphisme
  - Les objets et les types abstraits
  - C'est particulièrement simple avec les objets
    - Une des raisons du succès des objets
- L'idée est simple: si un programme marche avec une abstraction de données, il pourrait marcher avec une autre, si l'autre a la même interface

Exemple de polymorphisme

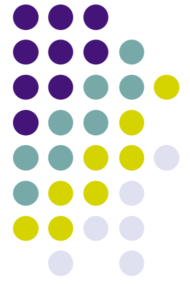


```
class Figure
   …
end

class Circle
   attr x y r
   meth draw … end
   …
end

class Line
   attr x1 y1 x2 y2
   meth draw … end
   …
end
```

```
class CompoundFigure
   attr figlist
   meth draw
      for F in @figlist do
         {F draw}
      end
   end
   …
end
```

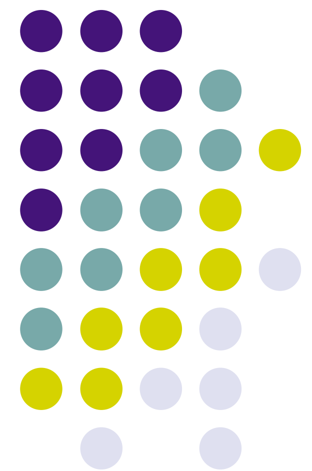
La définition de la méthode draw de CompoundFigure marche pour toutes les figures possibles: des cercles, des lignes et aussi d'autres CompoundFigures!

Exécution correcte d’un programme polymorphe

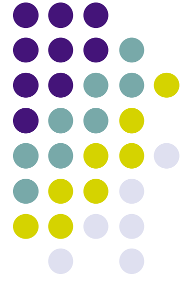


- Quand est-ce qu’un programme polymorphe est correct?
  - Pour une exécution correcte, l’abstraction doit satisfaire à certaines propriétés
  - Le programme marchera alors avec toute abstraction qui a ces propriétés
  - Pour chaque abstraction, il faut donc vérifier que sa spécification satisfait à ces propriétés
- Pour l’exemple des médecins, le programme exige que le médecin veut la guérison du patient
  - Le polymorphisme marche si chaque médecin veut la guérison du patient
  - Chaque abstraction (“médecin”) satisfait la même propriété (“veut la guérison du patient”)

L’héritage

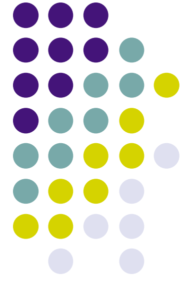


L'héritage et les classes



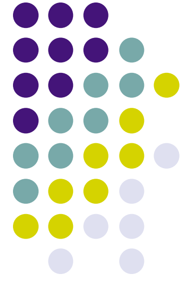
- Il peut être intéressant de définir des abstractions sans répéter les parties communes
  - Parties communes = code dupliqué
  - Si une partie est changée, toutes les parties doivent être changées
    - Source d'erreurs!
- L'héritage est une manière de définir des abstractions de façon incrémentale
  - Une définition A peut "hériter" d'une autre définition B
  - La définition A prend B comme base, avec éventuellement des modifications et des extensions
- La définition incrémentale A est appelée une classe
  - Attention: le résultat est une abstraction de données complète

Dangers de l’héritage



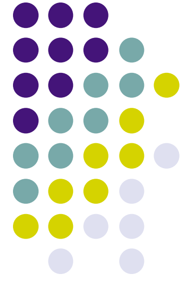
- L’héritage est parfois très utile, mais il faut l’utiliser avec beaucoup de précautions
- La possibilité d’étendre A avec l’héritage peut être vue comme une autre interface à A
  - Une autre manière d’interagir avec A
- Cette interface doit être maintenue pendant toute la vie de A
  - Une source supplémentaire d’erreurs!

Notre recommandation



- Nous recommandons d'utiliser l'héritage le moins possible
  - C'est dommage que l'héritage est considéré comme tellement important par les mandarins de la programmation orientée objet
- Quand on définit une classe, nous recommandons de la prendre comme "final" (non-extensible par l'héritage) par défaut
- Nous recommandons d'utiliser la composition de préférence sur l'héritage
  - La composition = une classe peut dans son implémentation utiliser des objets d'autres classes (comme la liste de figures dans CompoundFigure)

L’héritage versus la composition



B

hérite de

A

instance de

O_A

L’héritage

B

instance de

O_B

A

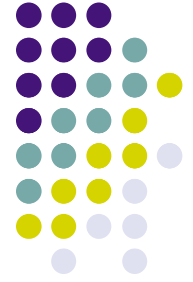
attr b: O_B

instance de

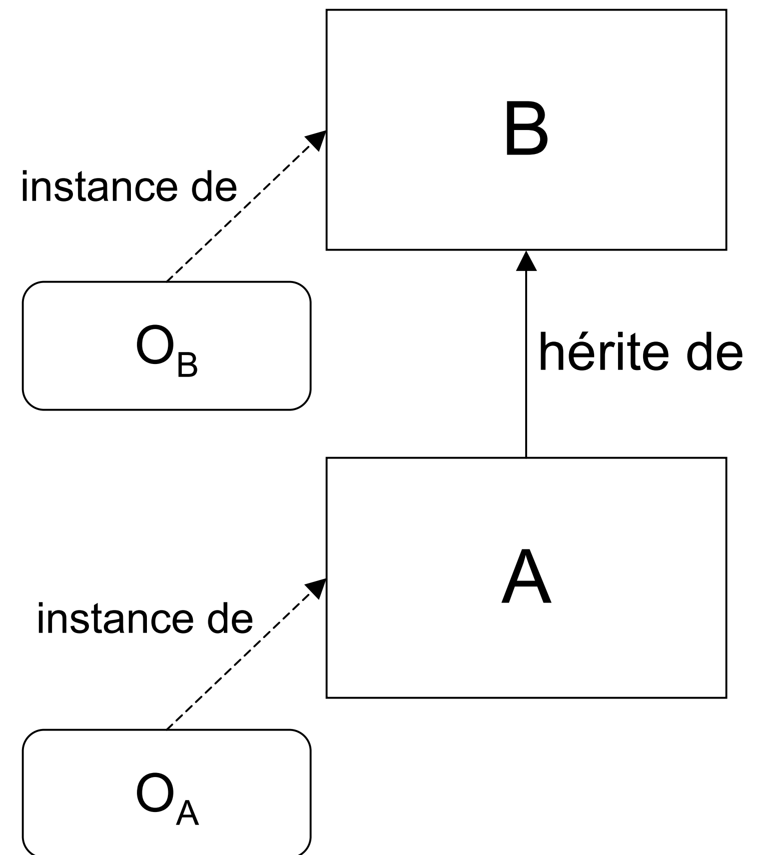
O_A

La composition

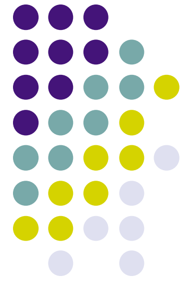
Le principe de substitution



- La bonne manière d'utiliser l'héritage
  - Supposons que la classe A hérite de B et qu'on a deux objets, O_A et O_B
- Toute procédure qui marche avec O_B doit marcher avec O_A
  - L'héritage ne doit rien casser!
  - A est une extension conservatrice de B

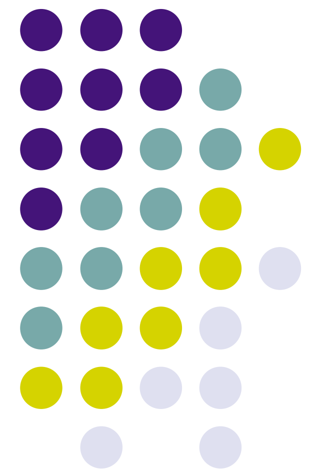


Le principe de substitution: une leçon coûteuse

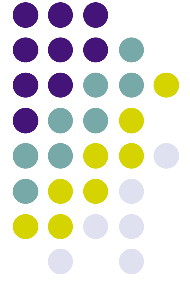


- Dans les années 1980, une grande entreprise a initié un projet ambitieux basé sur la programmation orientée objet
  - Non, ce n’est pas Microsoft!
- Malgré un budget de quelques milliards de dollars, le projet a échoué lamentablement
- Une des raisons principales était une utilisation fautive de l’héritage. Deux erreurs principales avaient été commises:
  - Violation du principe de substitution. Une procédure qui marchait avec les objets d’une classe ne marchait plus avec les objets d’une sous-classe!
  - Création de sous-classes pour masquer des problèmes, au lieu de corriger ces problèmes à leur origine. Le résultat était une hiérarchie d’une grande profondeur, complexe, lente et remplie d’erreurs.

La programmation orientée but

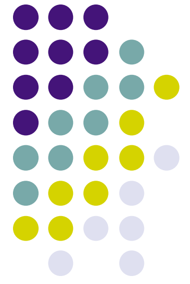


La programmation orientée but



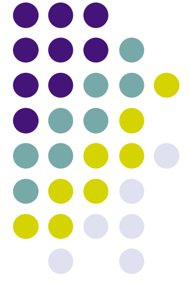
- Dans la programmation orientée but, c'est le résultat demandé qui contrôle le calcul à faire
- Cette approche est beaucoup utilisée dans l'intelligence artificielle
- Dans la programmation courante, elle apparaît sous deux formes principales:
  - La programmation paresseuse
  - La programmation par contraintes
- Il est possible d'utiliser ces deux formes ensemble

La programmation paresseuse



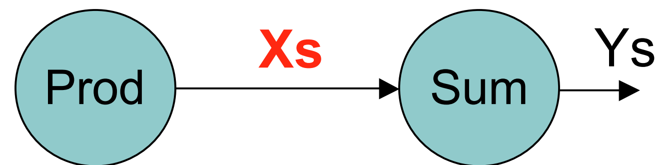
- C’est le consommateur qui décide s’il faut faire un calcul, pas le producteur
  - Dans une boucle, la condition de terminaison est dans le consommateur, pas dans le producteur
  - Le producteur peut avoir une boucle infinie!
  - Cette forme fait le minimum de calculs nécessaires pour obtenir un résultat
- Cette forme de calcul s’appelle la programmation paresseuse, par opposition à la programmation immédiate, où le producteur décide de faire un calcul même si on n’en a pas besoin
- La programmation fonctionnelle et la programmation dataflow monotone ont toutes les deux des variantes paresseuses
  - Les autres paradigmes n’en ont pas, pour des raisons fondamentales!

Dataflow monotone paresseuse



- Producteur/consommateur paresseux

```
fun lazy {Prod N}
  N|{Prod N+1}
end
```

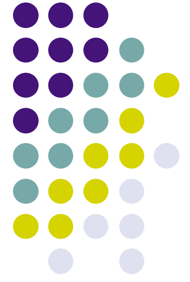



```
fun lazy {Sum S Xs}
  case Xs of X|Xr then
    S|{Sum S+X Xr}
  [] nil then nil end
end
```

```
local Xs in
  thread Xs={Prod 0} end
  thread Ys={Sum 0 Xs} end
  {Browse {Nth 1000 Ys}}
end
```

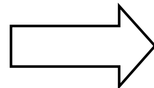
- **Différence avec la version immédiate (non-paresseuse): c'est le consommateur final qui décide combien il faut calculer, pas le producteur initial**
- Le comportement dataflow de l'instruction **case** marche comme avant
- Aucun autre contrôle n'est nécessaire

Le calcul paresseux avec WaitNeeded



- Le mot clé lazy est un raccourci pour l'utilisation de WaitNeeded, le seul nouveau concept nécessaire pour réaliser le calcul paresseux

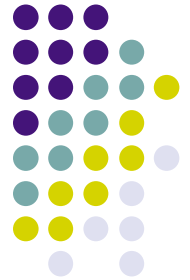
```
fun lazy {Prod N}
   N|{Prod N+1}
end
```



```
proc {Prod N Xs}
   {WaitNeeded Xs}
   local Xr in Xs=N|Xr {Prod N+1 Xr} end
end
```

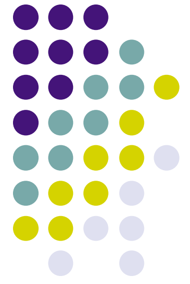
- Le résultat de la fonction Prod devient explicite: Xs
- {WaitNeeded Xs} attend jusqu'à ce qu'un autre fil a besoin de la valeur de Xs
- Cela permet la programmation dataflow paresseuse
  - Comme la programmation dataflow monotone, ce paradigme garde les bonnes propriétés de la programmation déclarative

Le chemin vers les contraintes



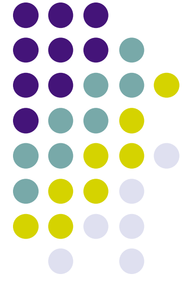
- L'affectation unique est un cas particulier d'un concept beaucoup plus puissant:
  - L'unification: l'égalité de deux structures de données
    - person(name:A age:25)=person(name:"Georges" age:B) → A="Georges" et B=25
  - Affectation unique = unification avec une seule variable
- L'unification est un cas particulier d'un concept beaucoup plus puissant:
  - La résolution des contraintes
  - $X,Y \in \{0,\ldots,9\} \wedge X+Y=10 \wedge X*Y=24 \wedge X \leq Y \rightarrow X=4 \wedge Y=6$
  - Unification = résolution d'une contrainte d'égalité
- La résolution de contraintes est vraiment puissante
  - Des contraintes plus expressives
  - Des résolveurs plus puissants

La programmation par contraintes

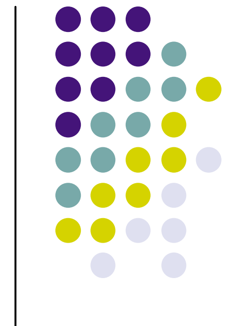


- La programmation par contraintes permet la programmation orientée but: il suffit de donner le but de la programmation comme une contrainte et l'outil trouvera un chemin vers le but
  - Un concept puissant qui nécessite un grand soutien du système
  - On peut changer le but en cours de route
  - On peut avoir plusieurs buts en même temps, qui sont en différents degrés de résolution
- En fait, il n'y a pas de magie: pour trouver le but, l'outil recherche un chemin dans un espace de possibilités
  - Mais il y a quand même un effet magique: c'est que la recherche est sacrément efficace parce que les techniques de résolution sont très avancées
  - Dans une première approximation, on peut ignorer le fait que la recherche est une technique exponentielle

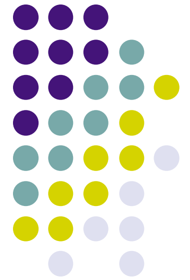
Conclusions



- Cet exposé a donné un survol rapide des paradigmes de programmation
  - Nous n'avons pas eu le temps de montrer beaucoup de code
  - C'est dommage, parce que chaque paradigme a une "âme" que l'on ne peut connaître qu'en l'utilisant
    - Chaque paradigme a une communauté active et des "champions"
- Nous vous recommandons d'explorer les différents paradigmes en faisant de la programmation
  - Avec un système multi-paradigme comme Mozart, Curry ou CIAO
  - Il y a une grande différence entre un système conçu comme multi-paradigme (comme Mozart) et un système qui contient beaucoup de concepts (comme Common Lisp)
    - Dans un système multi-paradigme, il est possible de programmer dans un paradigme sans interférence des autres paradigmes
  - Vous pouvez essayer chaque paradigme avec un langage qui le soutient bien: Haskell (fonctionnelle paresseuse), Erlang (envoi de messages), SQL (transactionnel), Oz (dataflow monotone, contraintes), etc.

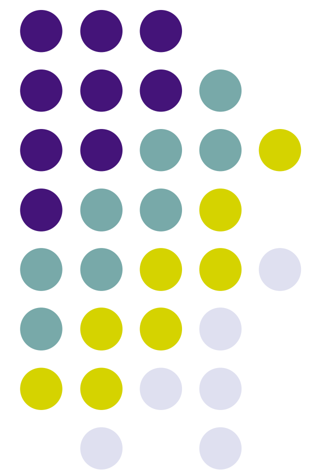


Quelques concepts pas abordés…

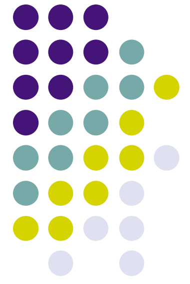


- Les transactions
  - Software transactional memory
- Les continuations

Les exceptions

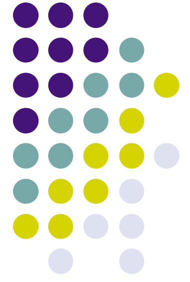


Les exceptions



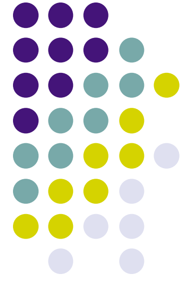
- Les exceptions sont un nouveau concept de programmation
- Comment est-ce qu'on traite les situations exceptionnelles dans un programme?
  - Par exemple: division par 0, ouvrir un fichier qui n'existe pas
  - Des erreurs de programmation mais aussi des erreurs imposées par l'environnement autour du programme
- Avec les exceptions, un programme peut gérer des situations exceptionnelles sans que le code soit encombré partout avec des bouts qui ne sont presque jamais exécutés

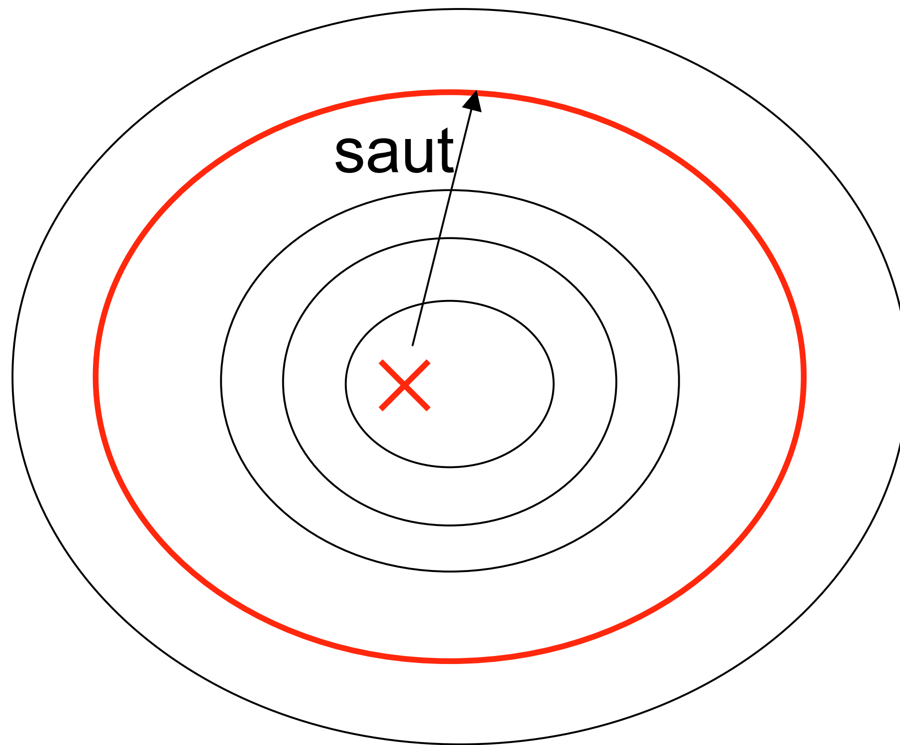
Le principe d'endiguement



- Quand il y a une erreur, on voudrait se retrouver dans un endroit du programme d'où l'on peut récupérer de l'erreur
- En plus, on voudrait que l'erreur influence la plus petite partie possible du programme
- Le principe d'endiguement:
  - Un programme est une hiérarchie de contextes d'exécution
  - Une erreur n'est visible qu'à l'intérieur d'un contexte dans cette hiérarchie
  - Une routine de récupération existe à l'interface d'un contexte d'exécution, pour que l'erreur ne se propage pas (ou se propage proprement) vers un niveau plus elevé

La gestion d'une exception (1)





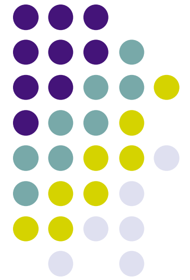
Une erreur qui lève une exception

Un contexte d'exécution

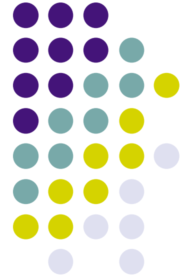
Le contexte d'exécution qui attrape l'exception

Mais c'est quoi exactement un contexte d'exécution?

La gestion d’une exception (2)



- Un programme qui rencontre une erreur doit transférer exécution vers une autre partie (le “gestionnaire d’exceptions”) et lui donner une valeur qui décrit l’erreur (l’“exception”)
- Deux nouvelles instructions
  try <stmt>$_1$ **catch** <x> **then** <stmt>$_2$ **end**
  raise <x> **end**
- Comportement:
  - Le **try** met un “marqueur” sur la pile et exécute <stmt>$_1$
  - S’il n’y a pas d’erreur, <stmt>$_1$ exécute normalement
  - S’il y a une erreur, le **raise** est exécuté, qui vide la pile jusqu’au marqueur (l’exécution du restant de <stmt>$_1$ est annulée)
    - Alors, <stmt>$_2$ est exécutée et l’exception est accessible par <x>
    - La portée de <x> couvre exactement <stmt>$_2$



La gestion d’une exception (3)

Pas d’erreur

mark

Exécution continue

Marqueur enlevé

<stmt>$_1$

mark

Pile sémantique

Création d’un contexte d’exécution (**try**)

Erreur (**raise**)

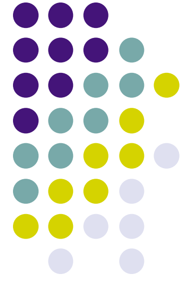
mark

Annuler l’exécution

<stmt>$_2$

Récupérati
on

Un contexte d’exécution



- Maintenant on peut définir exactement ce qu’est un contexte d’exécution
- Un contexte d’exécution est une partie de la pile sémantique qui commence avec un marqueur et qui va jusqu’au sommet de la pile
    - S’il y a plusieurs instructions **try** imbriquées, alors il y aura plusieurs contextes d’exécution imbriqués!
- Un contexte d’exécution est à l’intérieur d’une seule pile sémantique
    - Avec l’exécution concurrente il y aura plusieurs piles sémantiques (une par thread): voir plus loin dans le cours!
    - En général, c’est une bonne idée d’installer un nouveau contexte d’exécution quand on traverse l’interface d’un composant