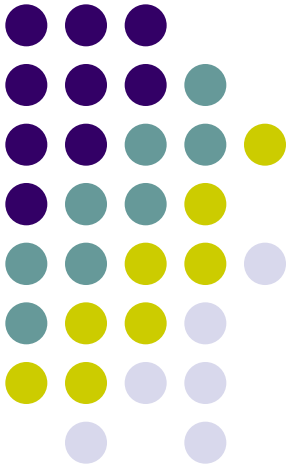
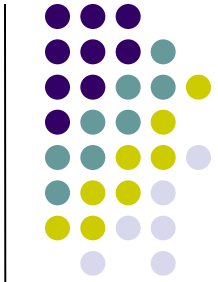
Introduction et Concepts de Base

Inspiré du cours de Peter Van Roy

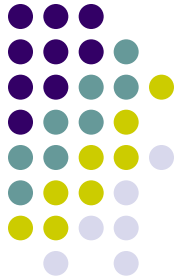


Il y a beaucoup de manières de programmer un ordinateur!



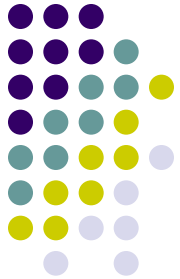
- Programmation déclarative
  - Programmation fonctionnelle ou programmation logique
- Programmation concurrente
  - Par envoi de messages ou par données partagées
- Programmation avec état
- Programmation orientée objet
- Programmation par composants logiciels

Modèles de programmation (“paradigmes”)



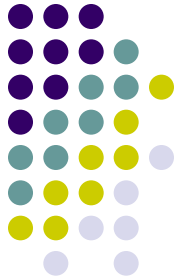
- Mettre ensemble
  - Des types de données et leurs opérations
  - Un langage pour écrire des programmes
- Chaque modèle/paradigme permet d’autres techniques de programmation
  - Les paradigmes sont complémentaires
- Le terme “paradigme de programmation”
  - Très utilisé dans le monde commercial; attention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

Langages de programmation



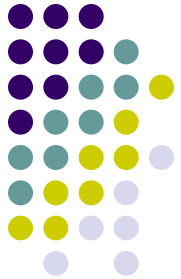
- Différents langages soutiennent différents modèles/paradigmes
  - Java: programmation orientée objet
  - Haskell: programmation fonctionnelle
  - Erlang: programmation concurrente pour systèmes fiables
  - Prolog: programmation logique
  - …
- Nous voudrions étudier plusieurs paradigmes !
- Est-ce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

La solution pragmatique…



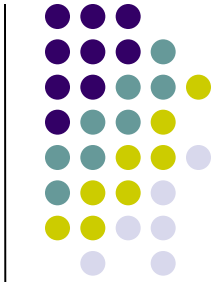
- Un seul langage de programmation
  - Qui soutient plusieurs modèles de programmation
  - Parfois appelé un langage “multi-paradigme”
- Notre choix est Oz
  - Soutient ce qu’on voit dans le cours, et plus encore
- L’accent sera mis sur
  - Les modèles de programmation !
  - Les techniques et les concepts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

Comment présenter plusieurs modèles de programmation ?



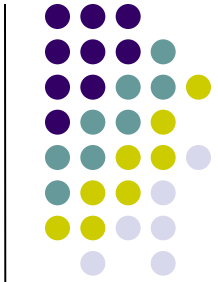
- Basé sur un langage noyau
  - Un langage simple
  - Un petit nombre de concepts **significatifs**
  - Buts: simple, minimaliste
- Langage plus riche au dessus du langage noyau
  - Exprimé en le traduisant vers le langage noyau
  - But : soutenir la programmation pratique

Approche incrémentale



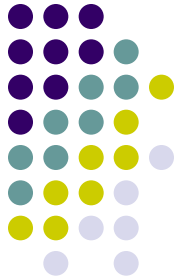
- Commencer par un langage noyau simple
  - Programmation fonctionnelle
- Ajouter des concepts
  - Pour obtenir les autres modèles de programmation
  - Très peu de concepts !
  - Très peu à comprendre !
- Nous ne verrons que quelques modèles, à cause de la taille limitée du cours
  - Trois modèles principaux

Les modèles que vous connaissez !



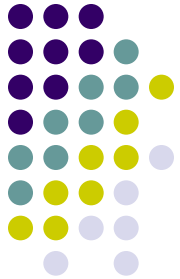
- Vous connaissez tous le langage Java, qui soutient
  - La programmation avec état
  - La programmation orientée objet
- C’est clair que ces deux modèles sont importants !

Pourquoi les autres modèles ?



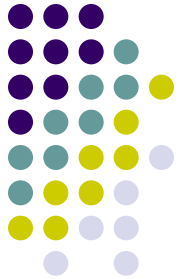
- Deux nouveaux modèles qu'on verra dans ce cours
  - Programmation déclarative (fonctionnelle)
  - Programmation concurrente (multi-agent) avec dataflow
- D'autres modèles pas vu dans ce cours
  - Programmation concurrente par envoi de messages ou par données partagées
  - Programmation logique (déterministe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

Notre premier modèle



- La programmation **déclarative**
  - Deux formes: programmation fonctionnelle et programmation logique
  - On peut considérer la programmation fonctionnelle comme la base de la plupart des autres modèles
    - On regardera uniquement la programmation fonctionnelle
- Approche
  - Introduction informelle aux concepts et techniques importants, avec exemples interactifs
  - Introduction au langage noyau
  - Sémantique formelle basée sur le langage noyau
  - Étude des techniques de programmation, surtout la récursion (sur entiers et sur listes) et la complexité calculatoire

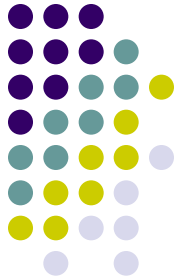
La programmation déclarative



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

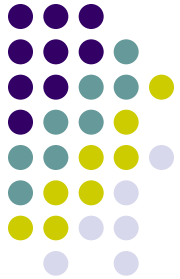
- De façon plus pragmatique
  - Demandez plus de soutien de l'ordinateur
  - Libérez le programmeur d'une partie du travail

Propriétés du modèle déclaratif



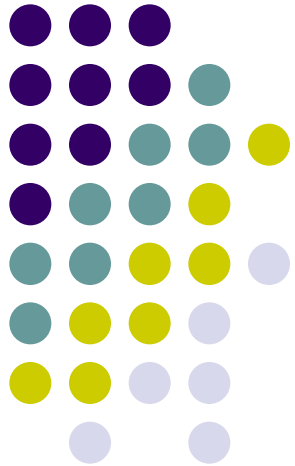
- Un programme est une fonction ou une relation au sens mathématique
  - Un calcul est l'évaluation d'une fonction ou une relation sur des arguments qui sont des structures de données
- Très utilisé
  - Langages fonctionnels: LISP, Scheme, ML, Haskell, …
  - Langages logiques (relationnels): Prolog,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

L’utilité du modèle déclaratif

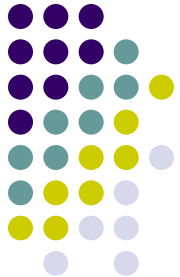


- Propriété clé: “Un programme qui marche aujourd’hui, marchera demain”
  - Les fonctions ne changent pas de comportement, les variables ne changent pas d’affectation
- Un style de programmation qui est à encourager dans tous les langages
  - “Stateless server” dans un contexte client/server
  - “Stateless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

Introduction aux Concepts de Base



Un langage de programmation



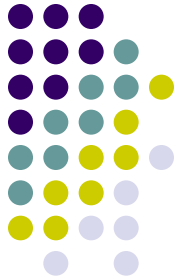
- Réalise un modèle de programmation
- Peut décrire des programmes
  - Avec des **instructions**
  - Pour calculer avec des **valeurs**
- Regardons de plus près
  - instructions
  - valeurs

Système interactif

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

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

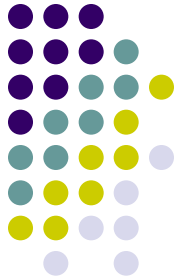
Système interactif : Qu'est-ce qui se passe ?



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

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

Variables



- Des raccourcis pour des valeurs
- Peuvent être affectées une fois au plus
  - Variables mathématiques
  - (Note: l'affectation multiple est un autre concept; on la verra plus loin dans le cours sous le nom de "cellule")
- Sont dynamiquement typées (type connu **pendant** l'exécution)
  - En Java elles sont statiquement typées: type connu **avant** l'exécution (à la compilation)
- Attention: il y a **deux** concepts cachés ici !
  - Premier concept : l'identificateur: le nom que vous tapez sur le clavier, c'est une chaine de caractères qui commence avec une majuscule

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

Déclaration d'une variable

```
declare
X = ...
```

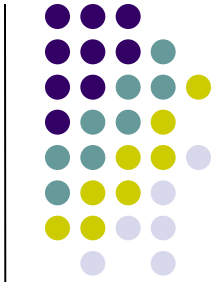
- `declare` est une instruction ("statement")
  - Crée une nouvelle variable en mémoire
  - Fait le lien entre l'identificateur `X` et la variable en mémoire
- Troisième concept : l'environnement
  - Une fonction des identificateurs vers les variables
  - Fait la correspondance entre chaque identificateur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

Affectation

```
declare
X = 42
```

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

Affectation unique



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

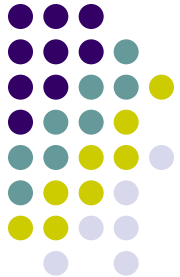
- Affectation incompatible : signale une erreur

```
X = 43
```

- Affectation compatible : ne rien faire

```
X = 42
```

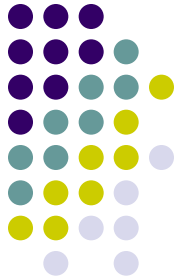
La redéclaration d'une variable (en fait : d'un identificateur !)



```
declare
X = 42
declare
X = 11
```

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

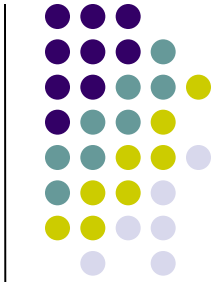
La portée d'une occurrence d'un identificateur



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

- L'instruction
  local X **in** <stmt> **end**
  déclare X qui existera entre **in** et **end**
- La portée d'une occurrence d'un identificateur est la partie d'un programme pour laquelle cet identificateur correspond à la même variable en mémoire.
- (Si la portée est déterminée par une inspection du code d'un programme, elle s'appelle portée lexicale ou portée statique.)
- Pourquoi n'y a-t-il pas de conflit entre X=42 et X=11 ?
- Le troisième Browse affichera quoi ?

Structures de données (valeurs)



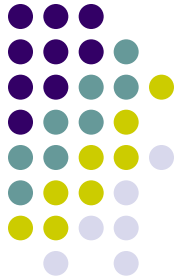
- Structures de données simples
  - Entiers `42, ~1, 0`

    `Notez:` “~” pour entier négatif (!)
  - Virgule flottante `1.01, 3.14, 0.5, ~3.2`
  - Atomes (constantes) `foo, 'Paul', nil`

- Structures de données composées
  - Listes
  - Tuples, enregistrements (“records”)

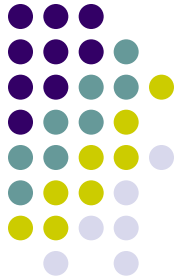
- Dans ce cours on utilisera principalement les entiers et les listes

Fonctions



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

Notre première fonction

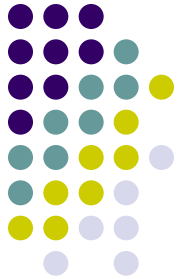


- Pour calculer la négation d'un entier
  - Prend un argument : l'entier
  - Rend une valeur : la négation de l'entier

- En notation mathématique:

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

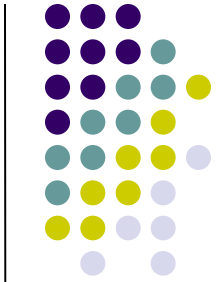
La définition de la fonction Minus



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

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

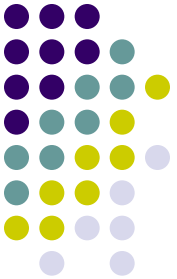
L'application de la fonction Minus



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

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

Syntaxe



- Définition d'une fonction

  fun {*Identificateur Arguments*}

  Corps de la fonction

  end

  `Le corps de la fonction est une expression qui calcule le résultat de la fonction`

- Appel d'une fonction

  `X =` {*Identificateur Arguments*}

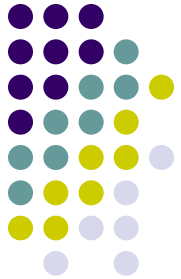
Fonction de maximum

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

- En notation mathématique:

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

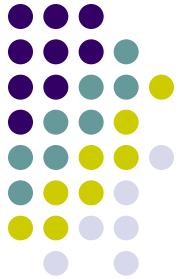
Définition de la fonction Max



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

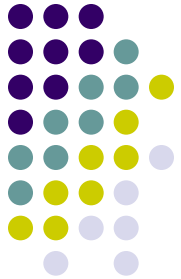
- Nouvelle instruction : conditionnel (if-then-else)
- Le conditionnel renvoie un résultat

Application de la fonction Max



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

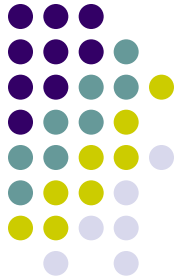
Maintenant le minimum



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

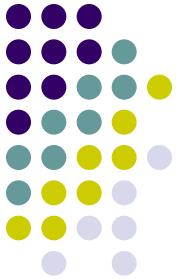
$$\min(n,m) = \sim\max(\sim n,\sim m)$$

Définition de la fonction Min



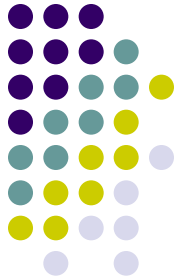
```
declare
fun {Min X Y}
   {Minus  {Max  {Minus X}
                 {Minus Y}}}
end
```

Définition de la fonction Min (avec ~)



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

Définition inductive d'une fonction



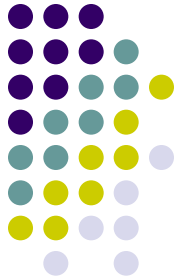
- Fonction de factorielle $n!$
  - La définition inductive est

    $0! = 1$

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

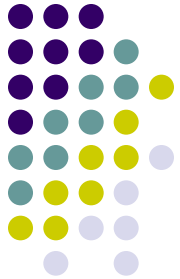
- Comment procéder ?
  - Utiliser l'application **récursive** de Fact !

Définition de la fonction Fact



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

Récursion

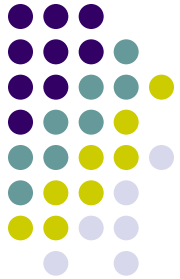


- Structure générale
  - Cas de base
  - Cas récursif

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

- On verra d'autres programmes récursifs la semaine prochaine !
  - La récursion est la manière de faire une boucle en modèle déclaratif

Une fonction est un cas particulier d'une procédure

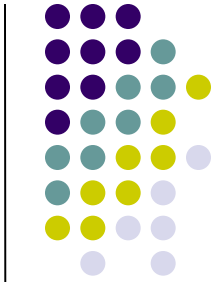


- Le vrai concept de base est la **procédure**
- Une fonction est une procédure avec un argument en plus, qui est utilisée pour renvoyer le résultat
- Z={Max X Y} *(fonction Max)*

  est équivalent à:

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

Résumé



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