`OCL : un langage d'expressions

Types : types de base, types énumérés, types construits

Expressions, Opérations

Opérations sur les collections

Types

- Types de bases
  - Integer
  - Real
  - Boolean
  - String
- Types énumérés
- Types construits
  - TupleType(x : T, y : T …)
  - Set(T)
  - OrderedSet(T)
  - Sequence(T)
  - Bag(T)
  - Collection(T)

- Types provenant d 'UML
  - classes
  - associations
  - …
- Méta-types
  - OclType
  - OclAny
  - OclState
  - OclExpression

Syntaxe des expressions

- OCL est un langage simple d 'expressions
  - constantes
  - identificateur
  - **self**
  - expr op expr
  - exprobjet . propobjet
  - exprobjet . propobjet (parametres)
  - exprcollection -> propcollection (parametres)
  - package::package::element
  - **if** cond **then** expr **else** expr **endif**
  - **let** var : type = expr **in** expr

A retenir

- . permet d 'accéder à une propriété d 'un objet
- -> permet d 'accéder à une propriété d 'une collection
- :: permet d 'accéder à un élément d 'un paquetage, d'une classe, …
- Des régles permettent de mixer collections et objets

```
self.impôts(1998) / self.enfants->size()
self.salaire() - 100
self.enfants->select( sexe = Sexe::masculin )
self.enfants->isEmpty()
self.enfants->forall(age>20)
self.enfants->collect(salaire)->sum()
self.enfants.salaire->sum()
self.enfants->union(self.parents)->collect(age)
```

Integer et Real

- **Integer**
  - valeurs : 1, -5, 34, 24343, ...
  - opérations : +, -, *, div, mod, abs, max, min
- **Real**
  - valeurs : 1.5, 1.34, ...
  - opérations : +, -, *, /, floor, round, max, min
- Le type **Integer** est « conforme » au type **Real**

Boolean

- **Boolean**
  - valeur : **true**, **false**
  - opérations : **not**, **and**, **or**, **xor**, **implies**, **if-then-else-endif**
- L 'évaluation des opérateurs or, and, if est partielle
  - true **or** x est toujours vrai, même si x est indéfini
  - false **and** x est toujours faux, même si x est indéfini

(age<40 **implies** salaire>1000) **and** (age>=40 **implies** salaire>2000)

if age<40 **then** salaire > 1000 **else** salaire > 2000 **endif**

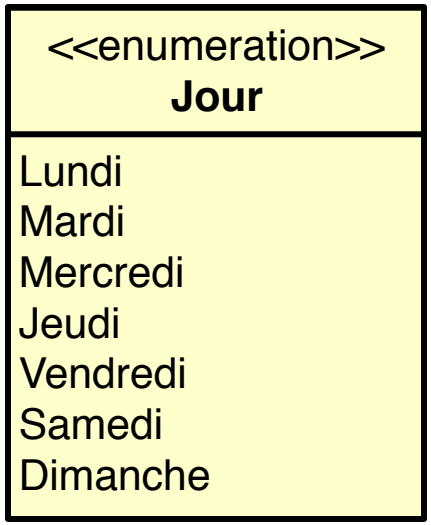
salaire > (if age<40 **then** 1000 **else** 2000 **endif**)

String

- **String**
  - valeur : ' ', ' une phrase '
  - opérations :
    - =
    - s.size(),
    - s1.concat(s2),
    - s1.substring(i1,i2)
    - s.toUpper(),
    - s.toLower(),
  - nom= nom.**substring**(1,1).**toUpper**().**concat**(
    nom.**substring**(2,nom.**size**()).**toLower**())
  - Les chaînes ne sont pas des séquences de caractères :-(
    String <> Sequence(character) , le type character n'existe pas

Enumération

- Utilisation d 'une valeur d 'un type énuméré
  Jour::Mardi (noté #Mardi avant UML2.0)
- Opérations
  =, <>
- Pas de relation d'ordre
- épouse->**notEmpty() implies** épouse.sexe = Sexe::Feminin



Element vs. singleton

- Dans tout langage typé il faut distinguer
  - un élément e
  - du singleton contenant cet élément Set{e}

- Pour simplifier la navigation OCL une conversion implicite est faite lorsqu 'une opération sur une collection est appliquée à un élement isolé

elem -> prop est équivalent à **Set**{elem}->prop

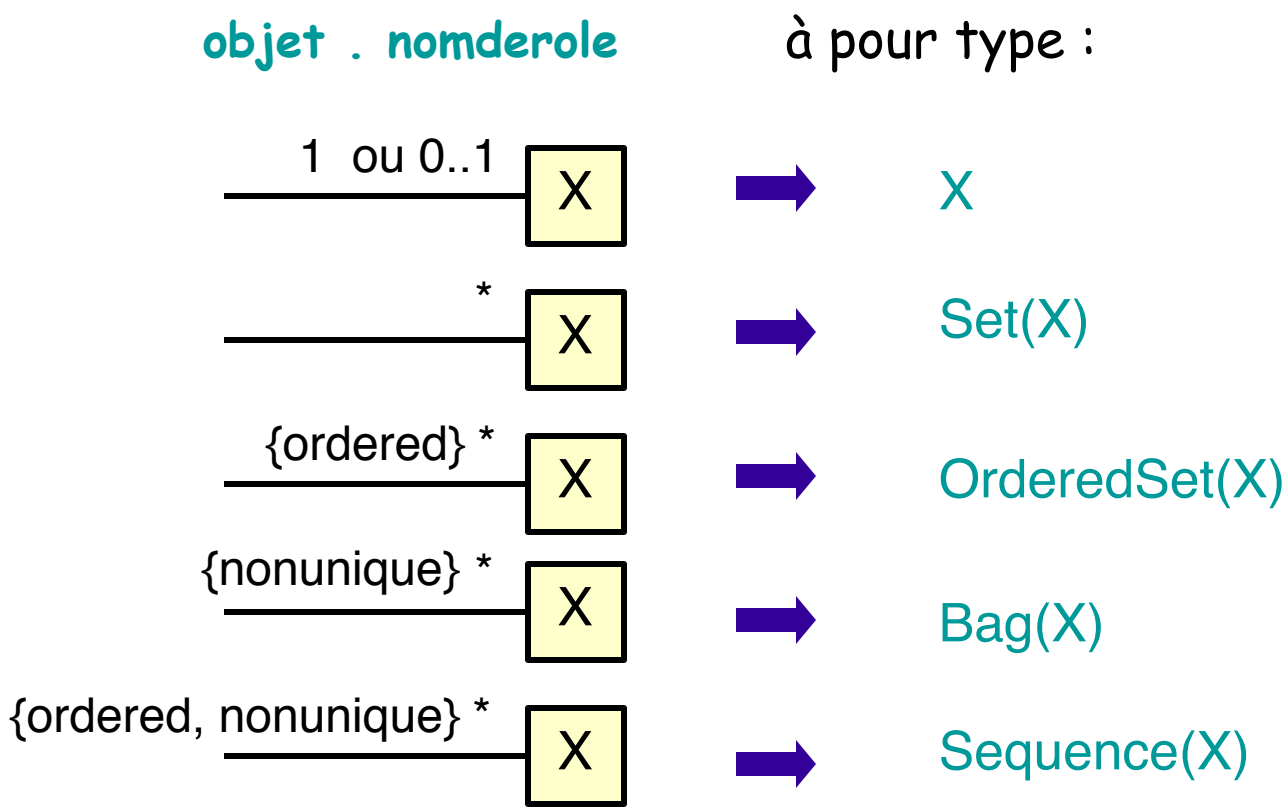
self->**size()** = 1

Collections

- Set — Set(T)
  - Unique, pas d'ordre
- OrderedSet — OrderedSet(T)
  - Unique, ordre
- Bag — Bag(T)
  - Répétition, pas d'ordre
- Sequence — Sequence(T)
  - Répétition, ordre

- Collection est le type de base — Collection(T)

Utilisation des collections lors de la naviagation



Expressions de type collection

- Exemples
  - **Set** { 'lundi', 'mercredi', 'mardi' }
  - **Bag** { 'lundi', 'lundi', 'mardi', 'lundi' }
  - **OrderedSet** { 10, 20, 5 }
  - **Sequence** { 'lundi', 'lundi', 'mardi', 'lundi' }

- Notation **..** pour spécifier des intervalles
  - **Sequence** { 1..5, 2..4 }
- Utile pour réaliser des itérations
  - **Sequence** { 0 .. nbétages-1 } -> **forall**(i | possèdeArrêt(i))

Opérations traditionnelles sur les collections

- Cardinalité : coll -> **size**()
- Vide : coll -> **isEmpty**()
- Non vide : coll -> **nonEmpty**()
- Nombre d 'occurrences : coll -> **count**(elem)
- Appartenance : coll -> **includes**(elem)
- Non appartenance : coll -> **excludes**(elem)
- Inclusion : coll -> **includesAll**(coll)
- Exclusion : coll -> **excludesAll**(coll)
- Somme des élements coll -> **sum**()

Exemples

- **Set** { 3, 5, 2, 45, 5 }->**size**()
- Sequence { 1, 2, 45, 9, 3, 9 } ->**count**(9)
- **Sequence** { 1, 2, 45, 2, 3, 9 } ->**includes**(45)
- **Bag** { 1, 9, 9, 1 } -> **count**(9)
- c->**asSet**()->**size**() = c->**size**()
- c->**count**(x) = 0
- **Bag** { 1, 9, 0, 1, 2, 9, 1 } -> **includesAll**(Bag{ 9,1,9})

Opérations sur les ensembles

- Union — ens -> **union**(ens)
- Intersection — ens -> **intersection**(ens)
- Difference — ens1 - ens2
- Ajout d 'un élément — ens -> **including**(elem)
- Suppression d 'un élément — ens -> **excluding**(elem)
- Conversion vers une liste — ens -> **asSequence()**
- Conversion vers un sac — ens -> **asBag()**
- Conv.vers un ens. Ord. — ens -> **asOrderedSet()**

Filtrage : select, reject et any

coll -> **select**(cond)

coll -> **reject**(cond)

coll -> **any**(cond)

- select retient les éléments vérifiant la condition
  - (extension d 'un prédicat sur un ensemble)
- reject élimine ces élements
- any sélectionne l'un des éléments vérifiant la condition
  - opération non déterministe
  - utile lors de collection ne contenant qu'un élément
  - retourne la valeur indéfinie si l'ensemble est vide

self.enfants ->**select**(age>10 **and** sexe = Sexe::Masculin)

self.enfants ->**reject**(enfants->**isEmpty()**)->**notEmpty()**

membres->**any**(titre='président')

Filtrage : autres syntaxes

Il est également possible de

- nommer la variable
- d 'expliciter son type

self.employé->**select**(age > 50)

self.employé->**select**(p | p.age>50)

self.employé->**select**(p : Personne | p.age>50)

self.employé->**reject**(p : Personne | p.age<=50)

Quantificateurs : forAll, exists, one

coll -> **forAll**(cond)

coll -> **exists**(cond)

coll -> **one**(cond)

- Quantificateur universel et existentiel

self.enfants->**forall**(age<10)

self.enfants->**exists**(sexe=Sexe::Masculin)

self.enfants->**one**(age>=18)

Quantificateurs : autres syntaxes

- Il est possible
  - de nommer la variable
  - d 'expliciter son type
  - de parcourir plusieurs variables à la fois

```
self.enfants->forall( age < self.age )
self.enfants->forall( e | e.age < self.age - 7)
self.enfants->forall( e : Personne | e.age < self.age - 7)
self.enfants->exists( e1,e2 | e1.age = e2.age )
self.enfants->exists( e1,e2 | e1.age = e2.age and e1<>e2 )
self.enfants->forall( e1,e2 : Personne |
                        e1 <> e2 implies e1.prénom <> e2.prénom)
```

Unicité

coll -> **isUnique** (expr)

- Retourne vrai si pour chaque valeur de la collection, l 'expression retourne une valeur différente

self.enfants -> **isUnique** (prénom)

à la place de

```
self.enfants->forall( e1,e2 : Personne |
                      e1 <> e2 implies e1.prénom <> e2.prénom)
```

- Pratique pour définir la notion de clé importée et de clé

Image d 'une expression : collect

coll -> **collect**(expr)

- Correspond à l 'image d 'une fonction (map, apply, ...)
- L 'expression est évaluée pour chaque élément
- Le résultat est
  - un sac, si l'opération est appliqué à un ensemble ou un sac
  - une séquence, si l'opération est appliqué à une séquence ou un ens. Ord.

self.enfants->**collect**(age) = Bag{10,5,10,7}

self.employés->**collect**(salaire/10)->**sum()**

Image d'une expression : autre syntaxe

- Collect est une opération de navigation courante
- Pour simplifier on peut utiliser le raccourci suivant

self.enfants->**collect**(age) <=> self.enfants.age

- Rappel : le résultat est un sac ou une séquence à cause des doublons
- Si l 'on souhaite obtenir un ensemble :

self.enfants.age->**asSet**()

- Permet de naviguer

self.enfants.enfants.voitures

Collections ordonnées vs. triées

- Collections ordonnées
  - Sequence
  - OrderedSet
- … mais pas forcément trièes
  - Sequence { 12, 23, 5, 5, 12 }
  - OrderedSet { 10, 4, 20 }

  Sequence { 5, 5, 12, 12, 23 }
  OrderedSet { 4, 10, 20 }

- `Sequence{1..s->size()-1} -> forall(i | s.at(i) < s.at(i+1) )`

Tri d'une collection

coll -> **sortedBy**(expr)

- L'opération < doit être définie sur le type correspond à expr
- Permet de trier une collection en fonction d'une expression
  - enfants->**sortedBy**(age)
  - enfants->**sortedBy**(enfants->**size**())->**last**()
  - **let** ages = enfants.age->**sortedBy**(a | a) **in** ages.**last**() - ages.**first**()
- Le résultat est de type
  - OrderedSet si l'opération est appliquée sur un Set
  - Sequence si l'opération est appliquée sur un Bag

Iterateur général : Iterate

- L'itérateur le plus général
- Autres itérateurs (forall, exist, select, etc.) : cas particulier
- coll -> iterate(élém : Type ;
  accumulateur : Type2 = <valeur initiale>
  | <expr>)

- Exemple
  - enfants -> **iterate(** e : Enfant ; acc : Integer = 0 | acc+e.age)
- Equivalent en pseudo code à

```
acc : Integer = 0 ;
foreach e:Enfant in enfants
    acc := acc+e.age
return acc
```