Notes de Cours de l'UE HLIN101
Première partie
Algorithmique et Programmation – L1

Philippe Janssen, Michel Leclère

Université Montpellier

Organisation prévue

- 6 séances de cours en Amphi.
- 14 séances de Travaux Dirigés par groupe d'une quarantaine. La première séance de TD aura lieu la semaine prochaine (regardez EDT sur le site de la Faculté des Sciences ou votre ENT).
- 8 séances de Travaux Pratiques dans les salles informatiques par groupe de 20. Les TP commenceront plus tard (en octobre).
- Attention vous êtes affecté à un groupe de TD et serez évalué dans ce groupe. Impossible de changer de groupe pour convenance personnelle.

Les documents

- 2 supports de cours. Le document distribué aujourd'hui correspond à la 1ère partie du cours. Ces documents sont des copies incomplètes des diapositives projetées. Vous devez les compléter.
- Des fiches de TD et TP qui seront distribuées en cours.
- Tous ces documents sont disponibles sur l'ENT (cours HLIN101 sur MOODLE). Cet espace contient également une rubrique *Soutien* qui vous aide à réviser le cours et propose des exercices supplémentaires pour évaluer vos connaissances.

Modalités de Contrôle des Connaissances

Votre note à l'UE sera calculée à partir de 3 notes :

- un contrôle en TD, début octobre (CTD)
- un contrôle en amphi sur la première partie du cours le mercredi 21 octobre à 18h30 (CA1)
- un contrôle en amphi après la fin des enseignements (CA2).

selon la formule : (1xCTD + 2xCA1 + 3xCA2) / 6

- En cas de non acquisition de l'UE, par une note supérieure à 10 ou par compensation semestrielle ou annuelle, vous pourrez passer l'épreuve de 2ème session
- En cas d'absence justifiée à un contrôle (CTD ou CA1), une épreuve de rattrapage vous sera proposée à condition d'envoyer un justificatif officiel dans les trois jours suivant la date de contrôle à philippe.janssen@umontpellier.fr
  Pas de rattrapage pour l'épreuve CA2 et l'épreuve de session 2.

La Discipline Informatique

L'*Informatique* est l'association d'une science et d'une technologie.

- Une **science**, ensemble de théories, outils formels et méthodes s'intéressant à :
  - la modélisation de l'information
  - la résolution de problèmes à l'aide d'ordinateurs,
- Une **technologie**, consistant en la conception, la réalisation et le maintien d'infrastructures matérielles et logicielles.

Positionnement du cours

- La partie scientifique de ce cours :
  - Utilise des « modèles » qui vous sont familiers : les nombres entiers $\mathbb{N}$ et $\mathbb{Z}$, les nombres réels $\mathbb{R}$, les booléens, les fonctions, les vecteurs et séquences de nombres et un peu de géométrie du plan et de calcul numérique.
  - Un modèle étant choisi, on pose un problème. On cherche alors un **algorithme** qui décrit la suite des étapes qui permet de résoudre le problème. La *machine* utilisée reste abstraite.
- La partie technologique de ce cours utilise les machines réelles (nos ordinateurs) et un environnement logiciel pour traduire les algorithmes en des fonctions (des programmes) **C**/**C++**, qu'on exécutera.

Algorithmique

- L'algorithmique n'est pas une science récente
  -300 : Euclide expose une méthode de calcul pour obtenir le plus grand diviseur de deux nombres entiers.
- Les algorithmes ne traitent pas que des problèmes numériques.
  - Algorithme de recherche de chemin dans un réseau (routier, informatique).
  - Algorithmes de traitement d'images.
  - Algorithmes de jeux.
- Le langage algorithmique n'est pas un langage de programmation.
- L'exécution des algorithmes ne nécessitent pas d'ordinateur.
- Attention à l'orthographe du mot « algorithme » !
  Pas d'agglo, ni de rythme !
  Le mot « algorithme » vient du nom du mathématicien perse du 9ième siècle Abu Abdullah Muhammad ibn Musa al-Khwarizmi, souvent considéré comme le « père de l'algèbre ».

Introduction

Objectif : résolution de problèmes sur ordinateur.

Cette résolution s'effectue en 2 étapes :

- Écriture de l'algorithme : méthode permettant de calculer le résultat à partir de la donnée du problème (sur une machine abstraite).
- Écriture du programme : traduction de l'algorithme pour une machine physique et un langage de programmation donnés.

Définition (Algorithme)

Un algorithme est une description finie d'un calcul qui associe un résultat à des données. Il est composé de 2 parties :

- sa spécification indique le nom de l'algorithme et décrit le problème résolu par l'algorithme (la fonction calculée par l'algorithme) : quelles sont les données du problème (ses paramètres) et quel est le résultat attendu.
- son corps décrit comment l'algorithme résout le problème. Ce calcul est écrit en langage d'algorithme qui fournit des opérations et objets primitifs et des moyens de les composer.

Exemple

Algorithme : estPair
Données : *a* : Nombre ; *a* est un nombre entier
Résultat : Booléen, *true* si *a* est pair, *false* sinon.

Le résultat est :
`cond( (a mod 2)=0, true, false )`
fin algorithme

Traduction dans des langages de programmation

Un algorithme peut être traduit dans plusieurs langages de programmation :

Exemple (Traduction de `estPair` en SCHEME)

```
(define estPair (lambda (a)
    (if (= ( modulo a 2) 0) #t #f)))
```

Exemple (Traduction de `estPair` en CAML)

```
let estPair : int -> bool  = fun
  (x) ->  if x mod 2 = 0 then true else false ;;
```

Exemple (Traduction de `estPair` en C++)

```
bool estPair(int n)
{ return
    n % 2 == 0 ? true : false;
}
```


Sommaire de la première partie

Le langage d'algorithme

Dans cette première partie, nous étudierons :

- Définition d'un langage d'algorithme :
  - Quels sont les éléments de base : valeurs et opérations primitives ?
  - Quelles sont les règles de composition ?
- Le langage de programmation qui servira à traduire et tester l'exécution de nos algorithmes sera le langage C/C++.

Les types

Les *objets* manipulés ont un type.

Définition (Type)

Un type est défini par :

- un domaine : l'ensemble des valeurs que peuvent prendre les *objets* du type
- un ensemble d'opérations et de fonctions qu'on peut appliquer aux valeurs du type.

Type Nombre

- domaine : $\mathbb{R}$
- opérations binaires classiques comme `+, *, -, ^, ...`
  qui associent un nombre à 2 nombres.
  Leur signature est : $\mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$
- fonctions numériques :
  - à un paramètre, de signature $\mathbb{R} \to \mathbb{R}$, comme `abs, log, sin, ...`
    Par exemple **abs** : $\mathbb{R} \longrightarrow \mathbb{R}$
    $x \longmapsto |x|$
  - à deux paramètres, de signature $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$, comme `max, min, ...`.
    Par exemple **max** : $\mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$
    $(x, y) \longmapsto$ le plus grand nombre parmi x et y

Type Nombre (suite)

Certaines opérations et fonctions ne sont définies que sur une partie des réels, par exemple :

- l'opération de division réelle : $/ : \mathbb{R} \times \mathbb{R}^* \longrightarrow \mathbb{R}$
- les opérations de division entière :
  quo : $\mathbb{Z} \times \mathbb{Z}^* \longrightarrow \mathbb{Z}$
  $(x, y) \longmapsto$ le quotient de la division entière de x par y
  mod : $\mathbb{Z} \times \mathbb{Z}^* \longrightarrow \mathbb{Z}$
  $(x, y) \longmapsto$ le reste de la division entière de x par y

Exemple

Attention `13 / 5` vaut `2.6`
`13 quo 5` vaut `2`
`13 mod 5` vaut `3`

Type Booléen

- domaine : `Bool={ true, false }`
- opérations
  - **non** : *Bool* $\longrightarrow$ *Bool*
  - **et, ou** : *Bool* $\times$ *Bool* $\longrightarrow$ *Bool*

  dont la sémantique (la valeur) est définie dans les tables :

a	non(*a*)
`true`	`false`
`false`	`true`

a	*b*	*a* et *b*	*a* ou *b*
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`

On utilise également des opérateurs de comparaison associant un booléen à 2 nombres : $=, \neq, <, \leqslant, >, \geqslant$.
Leur signature est : $\mathbb{R} \times \mathbb{R} \longrightarrow$ *Bool*
Exemple : $2 > 3$ vaut `false`

Les expressions

Avec les constantes de base (valeurs des domaines des types), les opérations et fonctions de base, on construit **récursivement** des expressions.

Définition (Expression)

Une expression est soit :

- une constante
  ex : 2, 17, true
- l'application d'une opération :
  `(expression opération expression)`
  ex : (2 + 3), (2 > 3)
- l'application d'une fonction :
  `fonction(expression,...,expression)`
  ex : abs(7), max(2,9)

Exemple

- expressions syntaxiquement incorrectes :
  (5 + * 7)   max 2 (5 , 2)
- expressions syntaxiquement correctes :
  `8`   `(true et false)`
  `(6 + (5 * 3))`   `((3 < 8) et ((1 + 5) = 7))`
  `max((15-12),5)`   `abs(max((1-5),-2))`
- `(true et 2)` est une expression syntaxiquement correcte qui, on le verra, n'a pas de sens : elle est sémantiquement incorrecte.

Remarque

Lorsqu'il n'y a pas d'ambiguïté on peut supprimer certaines parenthèses :
`(6 + (5 * 3))` est également noté `6+5*3`
`max((15-12),5)` est également noté `max(15-12,5)`

Définition (Valeur d'une expression)

La valeur `Val(exp)` de l'expression `exp` est définie récursivement par :

- **Constante**
  Si `exp` est une constante, `Val(exp)` est cette constante
- **Application d'une opération**
  Si `exp` est de la forme `(exp1 op exp2)`
  - Évaluer `exp1` et `exp2` ; soit `v1 = Val(exp1)` et `v2 = Val(exp2)`
  - `Val(exp)` est le résultat de l'application de l'opération `op` aux opérandes `v1` et `v2`
- **Application d'une fonction**
  Si `exp` est de la forme `fonct(exp1,... ,expn)`
  - Évaluer chaque expression ; soit `v1 = Val(exp1)`,..., `vn = Val(expn)`
  - `Val(exp)` est le résultat de l'application de la fonction `fonct` aux arguments `v1`,...,`vn`
- **Cas d'erreur**. L'évaluation d'une expression provoque une **Erreur** quand l'application d'une opération ou d'une fonction ne correspond pas à sa signature : nombre d'arguments différents, type des opérandes ou arguments différents.

Exemple

`exp`	`Val(exp)`	Type de `exp`
8	8	Nombre
(6 + (5 * 3))	21	Nombre
max((15-12),5)	5	Nombre
(true et false)	false	Booléen
(13 quo 5)	2	Nombre
(13 mod 5)	3	Nombre
((3 < 8) et ((1 + 5) = 7))	false	Booléen
abs(max((1-5),-2))	2	Nombre
(1 + true)	ERREUR	
abs(2,3)	ERREUR	

Remarque

On considérera que l'ordre d'évaluation des sous-expressions d'une expression n'a pas d'importance (`Val(a + b) = Val(b + a)`).
Exception pour les opérateurs booléens `et` et `ou` :

- Lors de l'évaluation de l'expression `(a et b)`,
  - on calcule `Val(a)`
  - Si `Val(a)= false` alors `Val((a et b))=false` (on n'évalue pas `b`)
  - Si `Val(a)= true` on calcule `Val(b)` ; `Val((a et b))=Val(b)`
- Lors de l'évaluation de l'expression `(a ou b)`,
  - on calcule `Val(a)`
  - Si `Val(a)= true` alors `Val((a ou b))=true` (on n'évalue pas `b`)
  - Si `Val(a)= false` on calcule `Val(b)` ; `Val((a ou b))=Val(b)`

⇒ `(a et b)` et `(b et a)` n'ont pas nécessairement la même valeur

Exemple

`Val( (2>0)et((10/2)<20) )`	`true`
`Val( ((10/2)<20)et(2>0) )`	`true`
`Val( (0>0)et((10/0)<20) )`	`false`
`Val( ((10/0)<20)et(0>0) )`	ERREUR

Définition (L'opérateur conditionnel : un nouvel opérateur)

- **Syntaxe** : une expression conditionnelle s'écrit : `cond( exp1, exp2, exp3 )` où :
  - `exp1` est une expression de type `Booléen`
  - `exp2` et `exp3` sont deux expressions de même type
- **Sémantique** : pour calculer `Val(cond( exp1, exp2, exp3 ))` :
  1. On calcule `v1=Val(exp1)`
  2. Si `v1=true` on calcule `v2=Val(exp2)`; la valeur de l'expression conditionnelle est `v2`
  3. Si `v1=false` on calcule `v3=Val(exp3)`; la valeur de l'expression conditionnelle est `v3`
- **Type** : le type de l'expression conditionnelle est celui de `exp2` et `exp3`
- **Cas d'erreur** : lorsque l'évaluation d'une sous-expression provoque une erreur ou lorsque `exp1` n'est pas de type booléen ou lorsque `exp2` et `exp3` ne sont pas de même type.

Exemple

`exp`	`Val(exp)`	Type de `exp`
cond((1+2)=3, 5, 6**)**	5	Nombre
cond((1+1)=3, 5, 6**)**	6	Nombre
cond((1<3) et (3<5), true et (1<3), false**)**	true	Booléen
cond(abs(-2)=2 et false , abs(2)+1, max(3,4)**)**	4	Nombre
cond((abs(2)<1), 8, 1 / 0**)**	ERREUR	
cond((abs(2)>1), 8, 1 / 0**)**	8	Nombre
cond((2=3), 1, true**)**	ERREUR	
cond(0, 2, 3**)**	ERREUR	
cond(max(2,3)=2, 3, **cond(** max(3,4)=3, 3, 4**))**	4	Nombre

Les algorithmes

Définition (algorithme)

Un algorithme décrit le calcul d'une fonction ; il est composé de 2 parties :

- **Ses spécifications**. Cette partie décrit la fonction calculée par l'algorithme ; elle est composée de :
  - nom de l'algorithme
  - description des données : nom et type de chaque paramètre ; plus conditions éventuelles
  - description du résultat : type et définition du résultat en fonction des paramètres
- **Son corps**. Cette partie décrit **comment** le résultat est calculé à partir des données. C'est une expression composée d'applications de fonctions, d'applications d'opérations, d'expressions conditionnelles, de constantes et **des paramètres de l'algorithme**.

Algorithme

Schéma d'algorithme

Algorithme : nom de l'algorithme
Données : description des paramètres
Résultat : description du résultat

Le résultat est : `une expression`
fin algorithme

Exemple

On souhaite définir un algorithme calculant la moyenne de 2 nombres, correspondant à la fonction **moy** : $\mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$

$(x, y) \longmapsto$ Moyenne de x et y

Algorithme : moy
Données : x : Nombre, y : Nombre
Résultat : Nombre, la moyenne de x et y

Le résultat est : `(x+y)/2`
fin algorithme

Correspondance des types

Le type du résultat d'un algorithme doit correspondre au type de l'expression résultat.

Définition (Type d'une expression contenant des paramètres)

Le type `Type(exp)` d'une expression paramétrée `exp` est défini par :

- **Constante**
  Si `exp` est une constante, `Type(exp)` est le type de la constante
- **Paramètre**
  Si `exp` est un paramètre, `Type(exp)` est le type de ce paramètre
- **Application d'une opération**
  Si `exp` est de la forme `(exp1 op exp2)` et si la signature de l'opération est **op** : $T1 \times T2 \longrightarrow TR$
  Alors `Type(exp)=TR`.
  Il y a une erreur de typage lorsque
  `Type(exp1)` $\neq$ `T1` ou `Type(exp2)` $\neq$ `T2`

Définition (Type d'une expression (suite))

- **Application d'une fonction**
  Si `exp` est de la forme `fonct(exp1, ... ,expn)` et si la signature de la fonction est **fonct** : $T1 \times \cdots \times Tn \longrightarrow TR$
  Alors `Type(exp) = TR`.
  Il y a une erreur de typage lorsque les nombres d'arguments et de paramètres diffèrent ou si pour un argument `expi` `Type(expi)` $\neq$ `Ti`.
- **Expression conditionnelle**
  Si `exp` est de la forme **cond(** `exp1, exp2, exp3` **)**
  Alors `Type(exp)=Type(exp2)=Type(exp3)`.
  Il y a une erreur de typage lorsque `Type(exp1)` $\neq$ `Booléen` ou si `Type(exp2)` $\neq$ `Type(exp3)`

Exemple (Type d'expressions avec paramètre)

Supposons que `x` et `y` sont des paramètres de type `Nombre`. Supposons que `z` et `w` sont des paramètres de type `Booléen`.

`Exp`	`Type(Exp)`
(x+y)/2	Nombre
abs(x-3) * y	Nombre
x<3	Booléen
(x=y) et w	Booléen
cond((x+y)=9, x-y, abs(x) **)**	Nombre
x+z	ERREUR
x+abs(w)	ERREUR
cond(x<10, y, z **)**	ERREUR
cond((x+y)=9, x<y, (z et w) **)**	Booléen

Application d'un algorithme

Définition (Application d'un algorithme : une nouvelle expression)

- **Syntaxe** : Comme celle de l'application d'une fonction : `nom-algorithme(exp1,...,expn)`
- **Sémantique** : `Val(nom-algorithme(exp1,...,expn))` est calculé comme suit :
  1. On évalue chaque expression `v1=Val(exp1),... vn=Val(expn)`
  2. On substitue dans l'expression paramétrée du corps de l'algorithme chaque `vi` au paramètre correspondant de l'algorithme. Le résultat de cette substitution est une expression (non paramétrée) `exp`
  3. On évalue l'expression `exp`
     La valeur de l'application de l'algorithme est la valeur de l'expression substituée :
     `Val(nom-algorithme(exp1,...,expn))=Val(exp)`

Définition (Application d'un algorithme (suite))

- **Type** : le type de l'expression application d'algorithme est celui du résultat de l'algorithme
- **Erreur** : Une erreur se produit lorsque :
  - l'algorithme n'est pas défini
  - Le nombre d'arguments de l'appel ne correspond pas au nombre de paramètres de l'algorithme
  - le type d'un argument est différent du type du paramètre correspondant
  - l'évaluation d'un argument provoque une erreur
  - l'évaluation de l'expression substituée provoque une erreur.

Exemple

Algorithme : moy
Données : x : Nombre, y : Nombre
Résultat : Nombre, la moyenne de x et y

Le résultat est : `(x+y)/2`
fin algorithme

Évaluation de l'expression `moy(1+2,6)` :

- évaluation des arguments d'application : `Val(1+2)=3; Val(6)=6`
- substitution de 3 à `x` et de 6 à `y` dans `(x+y)/2` ⇒ `(3+6)/2`
- évaluation de l'expression substituée : `Val((3+6)/2)=4.5`

`Val(moy(1+2,6))=4.5`

L'application d'un algorithme est une nouvelle expression qui peut être composée avec les autres formes d'expressions.

Exemple

`exp`	`Val(exp)`	Type de `exp`
moy(1+2,max(4,7))	5	Nombre
moy(2,4,6)	ERREUR	ERREUR
moy(true,true)	ERREUR	ERREUR
moy(5 quo 0,3)	ERREUR	Nombre
1+moy(1+2,max(4,7))	6	Nombre
moy(2,4)>2	true	Booléen
moy(moy(2,4),7)	5	Nombre
cond(2>5, 2+4, moy(3,5) **)**	4	Nombre

Le corps d'un algorithme peut contenir des expressions conditionnelles.

Exemple (algorithme testant si un nombre entier est pair)

$$\textbf{estPair} : \mathbb{Z} \longrightarrow \textit{Bool}$$
$$n \longmapsto \text{true si n est pair false sinon}$$

Algorithme : estPair
Données : n : Nombre ; n est un entier
Résultat : Booléen, `true` si `n` est pair, `false` sinon

Le résultat est :
cond(`n mod 2 = 0, true, false` **)**
fin algorithme

Lorsqu'il est défini, un algorithme peut être utilisé pour définir de nouveaux algorithmes.

Exemple (Définir un algorithme testant si la moyenne de 2 nombres est supérieure à 10)

$$\textbf{moySup10} : \mathbb{R} \times \mathbb{R} \longrightarrow \textit{Bool}$$
$$(x, y) \longmapsto \begin{cases} \text{true} & \text{si } \frac{x+y}{2} \geqslant 10 \\ \text{false} & \text{sinon} \end{cases}$$

Algorithme : moySup10
Données : `x : Nombre, y : Nombre`
Résultat : `Booléen` qui est `true` si la moyenne de `x` et `y` est supérieure ou égale à 10, `false` sinon.

Le résultat est : **cond(** `moy(x,y)⩾ 10, true, false` **)**
fin algorithme

Ou bien
Le résultat est : `moy(x,y)⩾ 10`

La récurrence

La récurrence est « naturellement » présente en Mathématiques comme en Informatique. On l'utilise pour définir des objets ou des fonctions.

Définition

On définit `objet` par récurrence en spécifiant :

- Les **cas de base** : des constantes donnant la valeur de `objet` sans faire appel à la récurrence.
- Les **équations de récurrence** : des définitions de `objet` qui font appel à d'autres valeurs de `objet`.

Une récurrence sera dite **correcte** si dans l'utilisation des équations de récurrence, la suite des appels récursifs est finie. C'est à dire si le processus d'appel de la définition, qui appelle la définition avec une deuxième valeur, qui appelle la définition avec une troisième valeur, qui ..., se termine toujours.

Exemple

Je définis `ExprB` par récurrence, en disant :
Un `ExprB` est soit :

- Cas de Base : une constante
- Équation de récurrence1 : `( ExprB opération ExprB )`, où `opération` est un nom d'opération binaire définie ailleurs.
- Équation de récurrence2 : `fonction ( ExprB , ExprB )` où `fonction` est un nom de fonction à deux paramètres définie ailleurs.

Par application de la définition on reconnaît des objets qui sont des `ExprB` :

`8`	`( 5 + 7 )`
`( 5 + ( 5 + 9 ) )`	`( ( 5 + 2 ) + 5 )`
`f( 5 , ( 5 + 2 ) )`	~~`x( f , ( 5 + 1 ))`~~
`f( f ( f ( 1 , 1 ) , 2 ) , 1 )`	~~`5( 1 + 1 )`~~
~~`f( 1 , 2 , 3 )`~~	~~`( a + b + c )`~~

Définition récursive d'une fonction

Ce procédé peut être utilisé pour définir des fonctions.

Exemple (Factorielle $(n) = 1 \times 2 \times \cdots \times (n-1) \times n$)

Je définis la fonction `factorielle`, en disant : `factorielle` s'applique à un entier $\mathtt{n} \in \mathbb{N}$:

- Cas de Base : quand $n = 0$, `factorielle(n)` vaut `1`
- Équation de récurrence : quand $\mathtt{n} > 0$ `factorielle(n)` vaut `n * factorielle( n-1 )`

on obtient donc la définition :

$$\textbf{factorielle} : \mathbb{N} \longrightarrow \mathbb{N}$$
$$\mathtt{n} \longmapsto \begin{cases} 1 & \textbf{si } \text{n} = 0 \\ \text{n * factorielle(n-1)} & \textbf{sinon} \end{cases}$$

Ce procédé est utilisé pour définir des algorithmes. Les règles d'application d'un algorithme récursif sont les mêmes que pour un algorithme non récursif.

Algorithme : fact
Données : *n* : Nombre ; *n* est un entier naturel
Résultat : Nombre, la factorielle de n

Le résultat est :
cond(`n=0, 1, n*fact(n-1)` **)**
fin algorithme

Val(fact(2))= 2

- Substitution : **cond(** 2=0, 1, 2*fact(2-1) **)**
- Evaluation : Val(2*fact(2-1))=2*Val(fact(1))=2*1=2
- Val(fact(1)) =1
  - Substitution : **cond(** 1=0, 1, 1*fact(1-1)**)**
  - Evaluation : Val(1*fact(1-1))=1*Val(fact(0))=1*1=1
  - Val(fact(0))=1
    - Substitution : **cond(** 0=0, 1, 0*fact(0-1)**)**
    - Evaluation : 1

Arrêt d'un calcul récursif

Propriété

Pour toute valeur de l'entier $\mathtt{n} \in \mathbb{N}$ *le calcul de* `fact(n)` *termine. En effet la suite des valeurs de l'argument est la suite* `n, n-1, n-2, ...` *décroissante, admettant pour minimum 0. Et on connaît la valeur de* `fact(0)` *pour ce cas de base.*

Exemple (de mauvaise définition récursive)

Je définis la fonction `prob`, en disant : `prob` s'applique à un entier $\mathtt{n} \in \mathbb{Z}$:

- Cas de Base : quand $n = 0$ `prob(n)` vaut `1`
- Équation de récurrence : quand $\mathtt{n} \neq 0$ `prob(n)` vaut `n * prob(n-2)`

La manière de spécifier ceci en Mathématiques pour `prob(n)` : est

$$\textbf{prob} : \mathbb{Z} \longrightarrow \mathbb{N}$$
$$\mathtt{n} \longmapsto \begin{cases} 1 & \textbf{si } \text{n} = 0 \\ \text{n * prob(n-2)} & \textbf{sinon} \end{cases}$$

La définition a l'air correcte pour un entier positif, mais ... le calcul de `prob(5)` ne termine jamais.

Schéma des fonctions récursives sur les entiers

Définition (L'énoncé du problème)

On se donne une fonction $\mathbf{f} : \mathbb{N} \times \ldots \longrightarrow \ldots$, $(n, \ldots) \longmapsto \ldots$

Comment définir *f* par récurrence ?

- **Cas de base** On indique la valeur de $V_B = f(n, \ldots)$ quand l'argument *n* vaut 0 ou, parfois, 1 ....
- **Récurrence** Quand *n* est différent du cas de base, on définit la valeur de $f(n, ..)$ en fonction de la valeur de la fonction *f* pour un argument inférieur à *n*, typiquement $n - 1$.

D'où le schéma fréquent

$$\mathbf{f} : \mathbb{N} \times \ldots \longrightarrow \ldots$$
$$(n, \ldots) \longmapsto \begin{cases} V_B & \textbf{si } \text{n=0} \\ \text{une expression avec } f(n-1, \ldots) & \textbf{sinon} \end{cases}$$

Exemple

On cherche à définir un algorithme pour calculer la fonction

somCarrés : $\mathbb{N} \longrightarrow \mathbb{N}$
$n \longmapsto 0^2 + 1^2 + 2^2 + \cdots + n^2$

- **Cas de base** : quand $n = 0$, `somCarrés(n)` vaut 0
- **Récurrence** : quand $n > 0$ comment définir `somCarrés(n)` en fonction de `somCarrés(n-1)` ?
  somCarrés(n) = $0^2 + 1^2 + \cdots + (n-1)^2 + n^2$ =
  $0^2 + 1^2 + \cdots + (n-1)^2$+$n^2$ = somCarrés(n-1)+n^2

Algorithme : somCarrés
Données : *n* : Nombre ; *n* est un entier naturel
Résultat : Nombre, l'entier naturel $0^2 + 1^2 + 2^2 + \cdots + n^2$
Le résultat est :
```
cond( n=0, 0, n*n+somCarrés(n-1))
```
fin algorithme

somCarrés(2)=2*2+somCarrés(1)=4+1*1+somCarrés(0)=4+1+0=5

Exemple

nbPairs : $\mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{N}$
$(a, b) \longmapsto$ Nombre d'entiers pairs dans l'intervalle$[a, b]$

- **Cas de base** quand $b < a$ nbPairs(a,b) vaut 0
- **Récurrence** quand $a \leqslant b$, on a 2 cas selon la parité de b :
  - quand *b* est pair, nbPairs(a,b) vaut 1+nbpairs(a,b-1)
  - quand *b* est impair, nbPairs(a,b) vaut nbpairs(a,b-1)

Algorithme : nbPairs
Données : *a* : Nombre, *b* : Nombre ; *a* et *b* sont des entiers.
Résultat : Nombre, le nombre d'entiers pairs dans l'intervalle $[a, b]$

Le résultat est :
```
cond( b<a, 0,
cond( estPair(b), 1+nbPairs(a,b-1),
      nbPairs(a,b-1)))
```
fin algorithme

nbPairs(2,4)=1+nbPairs(2,3)=1+nbPairs(2,2)=1+1+nbPairs(2,1)=1+1+0=2

PGCD de 2 entiers

Définition (Rappels)

Soient $a \in \mathbb{N}$ et $b \in \mathbb{N}^*$. Il existe de manière unique 2 entiers *q* et *r* tels que $a = b\,q + r$ avec $0 \leqslant r < b$.
q, *r* sont appelés le quotient et le reste de la division euclidienne de *a* par *b*.
On dispose d'opérations pour calculer *q* et *r* :

- `r` est le résultat de `a mod b`
- `q` est le résultat de `a quo b`

`a` divise `b` si `b mod a = 0`
Tout nombre a au moins pour diviseurs : 1 et lui même.
Dans l'ensemble des diviseurs communs à `a` et `b`, il y a au moins 1.
Le plus grand de ces diviseurs communs est appelé leur pgcd.

Exemple

Les diviseurs de 12 sont 1,2,3,4,6,12
Les diviseurs de 30 sont 1,2,3,5,6,10,15,30
`pgcd(12,30)` est 6

Propriété (PGCD)

Le pgcd de a et b est aussi pgcd de b et r.
(car x divise a et b si et seulement si x divise b et r).

Récurrence

Cette propriété fournit une récurrence :

- Pour calculer `pgcd( a , b )` on calcule `pgcd( b , a mod b )`.
- Quel est le cas de base ?
  Cas où la récurrence n'est pas applicable : $b = 0$.
  Quelle est la valeur de `pgcd(a,0)` ? *a*

Schéma de récurrence correct ?

Dans cet exemple, l'équation de récurrence ne lie pas pgcd(...,b) à pgcd(...,b-1). Cependant l'arrêt est assuré car à chaque appel récursif, l'argument *b* est remplacé par *a mod b* qui est un entier naturel strictement inférieur à *b*. La suite des valeurs du $2^{ème}$ argument est décroissante admettant pour minimum 0. Et on connaît la valeur de `pgcd(...,0)`.

L'algorithme

Algorithme : pgcd
Données : a : Nombre, b : Nombre ; a et b sont des entiers naturels ; ils ne doivent pas être tous les 2 nuls.
Résultat : Nombre, le nombre entier pgcd de *a* et *b*

Le résultat est :
```
cond( b=0, a, pgcd(b,a mod b))
```
fin algorithme

pgcd(25,9)=pgcd(9,25 mod 9)=pgcd(9,7)=pgcd(7,2)=pgcd(2,1)=pgcd(1,0)=1

pgcd(30,9)= pgcd(9,3)= pgcd(3,0)= 3

pgcd(9,30)=pgcd(30,9)= . . . =3

Le langage de programmation C/C++

Motivations

- Pourquoi un langage de programmation ? On écrit nos algorithmes dans un langage abstrait et indépendamment de toute « machine ». On veut ensuite tester/exécuter. Il nous faut donc un environnement d'exécution, composé d'une machine réelle, un langage de programmation et les **règles de traduction** qui permettent de passer du langage algorithmique au langage de programmation.
- Choix du langage C/C++ (plus exactement C++ sans la partie objet). C'est un langage compilé, typé, librement distribué ayant une sémantique claire. Nous n'utiliserons dans le cadre de ce cours qu'une partie infime du langage.
- Pourquoi ne pas écrire directement les algorithmes en C++ ? Lorsqu'on traite un problème on se concentre dans un premier temps sur la méthode de résolution sans tenir compte des problèmes de représentations des données, de gestion de la mémoire ... détails importants qui seront abordés au moment de la phase de programmation.

Les types de base en C/C++

Les nombres en C/C++

Contrairement à notre langage algorithmique, C/C++ fait une distinction entre les nombres entiers et les nombres réels :

- Il existe 2 types différents
- Le domaine des entiers est inclus dans celui des réels.

Les entiers relatifs

- Nom du type : `int`
- Domaine : sous intervalle de $\mathbb{Z}$: $[-2^{31}, 2^{31} - 1]$ pour une architecture 32 bits ; les constantes de type `int` sont notées de manière classique : 0 12 -3
- Opérations et fonctions : `+, *, -, /, %, abs`.
  Attention
  l'opération `quo` est notée / en C/C++ : 14 / 4 = 3
  l'opération `mod` est notée % en C/C++ : 14 % 4 = 2

Les réels

- Nom du type : `float`
- La représentation des réels en C/C++ utilise le codage sur 32 bits du format `IEEE 754`.
  Pour simplifier, une constante de type *float* s'écrit : `<nombre décimal>` ou bien `<nombre décimal>e<exposant>` où :
  - le nombre décimal est une suite de chiffres avec éventuellement un point le nombre de chiffres est limité à 20 chiffres.
  - l'exposant est un nombre entier relatif à 2 chiffres au plus

  Exemple : `41.87` est un flottant qui peut également être noté `4187e-2` ou encore `0.4187e2`.
  Attention lorsqu'il n'y a pas de partie exposant l'écriture d'un nombre réel doit contenir un point pour le distinguer de l'entier correspondant.
  Exemple : le réel *quatre* s'écrit `4.0` ou encore `4.`
  Attention le domaine est une partie finie des décimaux !
  Ce n'est ni $\mathbb{R}$ ni un intervalle de $\mathbb{R}$.

Le type réel (suite)

- Les opérations sont classiques :
  `+, -, *, /`
  Exemples : `3. * 2.1 = 6.3`   `3.2 / 2. = 1.6`
- Les fonctions sont classiques :
  `sqrt` (racine carrée), `floor` (valeur plancher), `ceil` (valeur plafond), `pow` (fonction puissance), `log, sin, exp...`
  Exemples : `floor(4.2)=4`   `floor(-4.2)=-5`   `ceil(4.2)=5`
- Les entiers sont assimilés à des réels.
  Exemples : `3.6 * 2 = 7.2`   `3.5 * 2 = 7`
  Mais pas l'inverse : un réel sans partie décimale n'est pas un entier
  Exemples : `3. % 2`   provoque une erreur de type
- Le symbole / désigne à la fois l'opérateur de division réelle et l'opérateur de division entière. C'est le type des opérandes qui détermine l'opération appliquée :
  Lorsque les 2 opérandes sont de type entier, la division entière est appliquée, sinon la division réelle est appliquée
  Exemples : `3 / 2 = 1`   `3. / 2. = 1.5`   `3. / 2 = 1.5`

Les booléens `bool`

- Nom du type : `bool`
- Les deux constantes du domaine sont notées `true` et `false`.
- Les opérations sont notées :
  ! pour l'opération `non` ; && pour l'opération `et` ; || pour l'opération `ou`

Opérateurs de comparaison

==, !=, <, >, <=, >= : *float* $\times$ *float* $\longrightarrow$ *bool*

Exemple : `3 < 4` vaut `true` ; `6.1 >= 2.` vaut `true` ; `3 < 4.5` vaut `true`

Attention $\neq$ est noté != en C/C++
$=$ est noté == en C/C++ (et non = qui aura une autre signification)

Autres types C/C++

Il existe d'autres types en C/C++, en particulier pour désigner les nombres. Ils diffèrent selon l'étendue et la précision du codage utilisé pour représenter les nombres.

Les expressions en C/C++

Expressions

La syntaxe des expressions est identique à celle définie en langage d'algorithme, à l'exception de celle des expressions conditionnelles

Langage d'algorithmes	Langage C/C++
cond(exp1, exp2, exp3 **)**	exp1 **?** exp2 **:** exp3

Exemple : 1>2 ? 3 : 4   vaut   4

Les algorithmes en C/C++

Définition d'un algorithme

La définition d'un algorithme correspond en C/C++ à la définition d'une nouvelle fonction. Sa syntaxe est :

Langage d'algorithme	Langage C/C++

Algorithme : nom
Données : x1 :T1,. . .,xn :Tn
Résultat : Tr . . .
 	
Le résultat est :
expression
fin algorithme
	`Tr nom(T1 x1, ..., Tn xn)`
	`{`
	`return`
	`expression ;`
	`}`

Exemple

langage d'algorithme

Algorithme : moy
Données : x : Nombre, y : Nombre
Résultat : Nombre, la moyenne de x et y

Le résultat est :
(x+y)/2
fin algorithme

langage C/C++

```
float moy(float x, float y)
{
  return
    ( x + y) / 2 ;
}
```


La syntaxe des algorithmes récursifs est identique :

Exemple

Algorithme : pgcd
Données : a : Nombre, b : Nombre ; a et b sont 2 entiers naturels ; ils ne doivent pas être tous les 2 nuls.
Résultat : Nombre, le nombre entier pgcd de *a* et *b*

Le résultat est :
cond(b=0, a, pgcd(b,a mod b)**)**
fin algorithme

```
int pgcd(int a, int b)
// a et b sont 2 entiers naturels non nuls
{
  return
    b==0 ? a : pgcd(b,a % b);
}
```

Remarque : on peut inclure des commentaires en les commençant par //

Programme C/C++

Structure d'un programme

Pendant les séances de TP, vous écrirez et exécuterez des programmes écrits en C/C++. Un programme sera composé :

- d'un fichier contenant les définitions de fonctions traduction d'algorithmes
- d'un fichier contenant le *programme principal* constitué d'une suite d'évaluations d'expressions, en particulier des applications de fonctions.
- ... et d'autres fichiers ...

Exécution d'un programme

Pour exécuter le programme, il faudra le compiler. Cette opération consiste à traduire l'ensemble des fichiers composant le programme en un fichier (*fichier exécutable*) qui contient le code exécutable par la machine. Les éventuelles erreurs de syntaxe et/ou sémantique sont signalées par le compilateur. Sinon l'exécution du programme (*fichier exécutable*) revient pour chaque expression du *programme principal* à :

- lire l'expression, l'évaluer, afficher sa valeur

Exemple de programme C/C++

Fichier *Définitions des fonctions*

```
...
int pgcd(int a, int b)
{ return
    b==0 ? a :
    pgcd(b,a % b);
}
...
```

Fichier *Programme principal*

```
...
int main()
{
  EVAL(9/2);
  EVAL(2<3 && 2==3);
  EVAL(pgcd(24,16));
...}
```

⇓ Compilation ⇓

Fichier *exécutable*

00110100111010010011101

⇓ Exécution ⇓

```
Valeur de 9/2 : 4
Valeur de 2<3 && 2==3 : false
Valeur de pgcd(24,16) : 8
...
```

Les listes

Notion de Liste

Une `Liste` est une structuration des données correspondant à une séquence de valeurs toutes de même type et de longueur arbitraire.
La valeur d'une `liste` sera représentée par la séquence de ses éléments, encadrée par des parenthèses :
Par exemple `(1 6 2)` est la `Liste` composée des 3 nombres 1, 6 et 2.
La *Liste Vide* composée d'aucun élément est notée `( )`

Remarque

- Une liste permet de regrouper des valeurs :
  par exemple une série de relevés de températures pourra être représentée par une liste de nombre réels.
- L'ordre des éléments d'une liste est significatif :
  Les listes `(1 6 2)` et `(6 1 2)` sont 2 listes différentes.

Définition (Récursive d'une liste)

On peut donner une définition récursive des listes. Une `Liste` est

- soit la *liste vide*
- soit une liste non vide qui est un couple composé :
  - du premier élément de la liste (appelé *Tête de la liste*)
  - et de la liste constituée des autres éléments (le 2ème, le 3ème, ...) (appelée *Queue de la liste*)

Exemple

La liste `(3 6 2)` est composée :

- de la tête de liste 3
- de la queue de liste `(6 2)` qui est la liste composée :
  - de la tête de liste 6
  - de la queue de liste `(2)` qui est la liste composée :
    - de la tête de liste 2
    - de la queue de liste `( )` qui est la liste vide

Le type `Liste de Nombres`

Définition (Type Liste de Nombres)

On introduit un nouveau type, le type `Liste de Nombres`

- **Domaine** : les valeurs de ce type sont des séquences, éventuellement vides, de nombres.
- **Fonctions** permettant d'accéder aux membres d'une liste :

  tête : Liste de Nombres non vide $\longrightarrow$ *Nombre*
  $L \longmapsto$ Le 1er élément de la liste L

  queue : Liste de Nombres non vide $\longrightarrow$ Liste de Nombres
  $L \longmapsto$ { La liste L privée de son 1er élément

  estVide : Liste de Nombres $\longrightarrow$ Booléen
  $L \longmapsto$ *true* si L est la liste vide, *false* sinon

  cons : *Nombre* $\times$ Liste de Nombres $\longrightarrow$ Liste de Nombres
  $(e, L) \longmapsto$ { La liste dont :
  - le 1er élément est e
  - la queue est la liste L

Expressions avec des listes

Exemple

`exp`	`Val(exp)`	Type de `exp`
tête((5 2 8))	5	Nombre
queue((5 2 8))	(2 8)	Liste de Nombres
cons(8, (5 2 8))	(8 5 2 8)	Liste de Nombres
queue(queue((5 2 8)))	(8)	Liste de Nombres
tête(tête((5 2 8)))	ERREUR	ERREUR
cons(8, 3)	ERREUR	ERREUR
queue((7))	( )	Liste de Nombres
tête(queue((7)))	ERREUR	
queue(queue((7)))	ERREUR	
cons(6, queue((5 8)))	(6 8)	Liste de Nombres
cond(estVide(queue((7))), 1, 2 **)**	1	Nombre

Les algorithmes sur les listes

Exemple

- Algorithme calculant le deuxième élément d'une Liste :
  Algorithme : deuxièmeElément
  Données : li : Liste de Nombres ; li a au moins 2 éléments
  Résultat : Nombre, le deuxième élément de la liste li
  Le résultat est : `tête(queue(li))`
  fin algorithme
- Algorithme supprimant le deuxième élément d'une Liste :
  Algorithme : oterEl2
  Données : li : Liste de Nombres ; li a au moins 2 éléments
  Résultat : Liste de Nombres, la liste li sans son deuxième élément
  Le résultat est : `cons( tête(li), queue(queue(li)))`
  fin algorithme

deuxièmeElément((2 4 1))=tête(queue((2 4 1)))=tête((4 1))=4
oterEl2((2 4 1))=cons(tête((2 4 1)), queue(queue((2 4 1))))=cons(2,(1))=(2 1)

Schéma des algorithmes récursifs sur les listes

La plupart des algorithmes sur les listes utilisent un schéma récursif. Ce schéma est basé sur la définition récursive des listes :

Pour définir par récurrence un algorithme calculant une fonction
g : *Liste* × ... ⟶ ...

- **Le cas de base** donne la valeur `VB` de `g` quand la liste est vide.
- **La récurrence** définit la valeur de `g(li,..)` en fonction de `g(queue(li),..)`

D'où le schéma fréquent

Algorithme : calculDeG
Données : li : Liste de Nombres ...
Résultat : ...
Le résultat est :
```
cond( estVide(li), VB,
...calculDeG(queue(li),...) ...)
```
fin algorithme

Algorithmes récursifs sur les listes

Exemple (Calcul du nombre d'éléments d'une liste)

Algorithme : longueur
Données : li : Liste de Nombres
Résultat : Nombre, nombre d'éléments de la liste li

Le résultat est :
```
cond( estVide(li), 0,
1+longueur(queue(li)))
```
fin algorithme

- Quand `li` est la liste vide, quelle est la longueur de `li` ? 0
- Quand `li` n'est pas vide, connaissant `longueur(queue(li))`, quelle est la longueur de li ? 1+longueur(queue(li))

longueur((4 5 2 7)) = 1+longueur((5 2 7)) = 1+3 =4

Exemple (Ajouter 1 à chaque élément d'une liste)

Algorithme : ajout1
Données : li : Liste de Nombres
Résultat : Liste de Nombres, liste obtenue à partir de li en ajoutant 1 à chacun de ses éléments.

Le résultat est :
```
cond( estVide(li), li,
cons( tête(li)+1 , ajout1(queue(li)) ))
```
fin algorithme

- Ajouter 1 à chaque élément de la liste vide est la liste vide
- Connaissant ajout1(queue(li)), quelle est la liste ajout1(li) ?
  C'est la liste dont la tête est tête(li)+1
  et dont la queue est ajout1(queue(li))
  c'est à dire la liste cons(tête(li)+1, ajout1(queue(li)))

ajout1((4 5 2 7)) = cons(4+1, ajout1((5 2 7))) = cons(5, (6 3 8)) =(5 6 3 8)

Pour certains problèmes le schéma récursif de l'algorithme est plus compliqué :

Exemple (Vérifier si un nombre appartient à une liste)

Algorithme : appartientLi
Données : e : Nombre, li : liste de Nombres
Résultat : Booléen qui vaut `true` si `e` est un élément de la liste `li`, `false` sinon.

Le résultat est :

```
cond( estVide(li), false,
(tête(li)=e) ou appartientLi(e,queue(li)))
```

fin algorithme

- Est-ce que e appartient à la liste vide ? non
- Si li n'est pas vide, à quelle condition e appartient-il à la liste li ?
  e est égal à tête(li) ou e appartient à queue(li)

appartientLi(2,(4 3 2)) =(tête((4 3 2))=2) ou appartientLi(2,queue((4 3 2))) = (4=2) ou appartientLi(2,(3 2)) = false ou true = true

Autre version de l'algorithme `appartientLi`

Algorithme : appartientLi
Données : e : Nombre ; li : Liste de Nombres
Résultat : Booléen : `true` si `e` est un élément de la liste `li`, `false` sinon.

Le résultat est :

```
cond( estVide(li), false,
(tête(li)=e) ou appartientLi(e,queue(li)))
```

fin algorithme
Ou bien

Le résultat est :

```
cond( estVide(li), false,
cond( tête(li)=e, true,
appartientLi(e,queue(li))))
```

fin algorithme

Algorithme de concaténation de deux listes

Définition (Concaténation de Listes)

La concaténation des listes `li1` et `li2` est la liste formée des éléments de `li1` (dans le même ordre), puis des éléments de `li2` (dans le même ordre).

Exemple (Concaténation)

- La concaténation des listes (1 2) et (5 3 6) est la liste (1 2 5 3 6)
- La concaténation des listes (2) et (2 5 5 2) est la liste (2 2 5 5 2)
- La concaténation des listes (1 2) et ( ) est la liste (1 2)
- La concaténation des listes ( ) et (3 2) est la liste (3 2)

Algorithme de concaténation

Algorithme : concaténer
Données : li1 : Liste de Nombres, li2 : Liste de Nombres
Résultat : Liste de Nombres, résultat de la concaténation des listes li1 et li2

Le résultat est :

```
cond( estVide(li1), li2,
cons( tête(li1) , concaténer(queue(li1),li2) ))
```

fin algorithme

- Choisir le paramètre par rapport auquel sera définie la récursivité : li1
- Le résultat de la concaténation de la liste vide avec li2 est li2
- Si li1 n'est pas vide, le résultat de la concaténation de li1 et li2 est la liste dont la tête est tête(li1) et dont la queue est concaténer(queue(li1),li2). C'est à dire la liste cons(tête(li1), concaténer(queue(li1),li2))

concaténer((3 7),(2 9)) = cons(tête((3 7)), concaténer(queue((3 7)),(2 9))) = cons(3, concaténer((7),(2 9))) = cons(3, (7 2 9)) = (3 7 2 9)

Jusqu'à présent nous n'avons étudié que les listes de nombres. Nous pourrons par la suite manipuler d'autres types de listes : des listes de booléens et même des listes de listes de nombres…

Définition

Pour chaque type `T` déjà défini (pour le moment `Nombre, Booléen, Liste de Nombres`), on définit le nouveau type `Liste de T` comme suit :

- **Domaine** : les valeurs sont des séquences de valeurs de type `T`
- **Fonctions** :

tête : Liste de T non vide $\longrightarrow$ T
$L \longmapsto$ Le 1[er]élément de la liste L

queue : Liste de T non vide $\longrightarrow$ Liste de T
$L \longmapsto$ La queue de la liste L

estVide : Liste de T $\longrightarrow$ Booléen
$L \longmapsto$ *true* si L est la liste vide, *false* sinon

cons : $T \times$ Liste de T $\longrightarrow$ Liste de T
$(e, L) \longmapsto$ { La liste dont le 1[er] élément est e et dont la queue est la liste L

Les types `Liste de T` en C/C++

Pour chaque type `T` de C/C++ le type `Liste de T` est défini en C/C++ par :

- **Nom du type** : `list<T>` (par exemple le type `Liste de nombres réels` s'appelle en C/C++ `list<float>`)
- **Domaine** : séquences de valeurs de type `T`, la liste vide est notée `liVide<T>()`.
- **Fonctions, opération, égalité**

estVide : $list < T > \longrightarrow bool$
$L \longmapsto$ *true* si L est la liste vide, *false* sinon

tete : $list < T > \longrightarrow T$
$L \longmapsto$ tête de la liste non vide L

queue : $list < T > \longrightarrow list < T >$
$L \longmapsto$ queue de la liste non vide L

cons : $T \times list < T > \longrightarrow list < T >$
$(e, L) \longmapsto$ liste de tête e et de queue L

Pour simplifier l'écriture des expressions, on ajoute la fonction `liste` qui étant donné une séquence de valeurs séparées par une virgule, encadrée par une paire d'accolades, a pour résultat la liste des valeurs de la séquence.

Exemple (fonction `liste`)

L'expression `cons(3,cons(1,cons(5,liVide<int>())))`, dont la valeur est la liste `(3 1 5)`, est équivalente à l'expression `liste({3, 1, 5})`.

Exemple (de session C/C++)

EVAL(cons(1, cons(3, cons(7, liVide<int>())))) ;
=> Valeur de cons(1,cons(3,cons(7,liVide<int>()))) : (1 3 7)
EVAL(liste({1, 3, 7})) ;
=> Valeur de liste({1, 3, 7}) : (1 3 7)
EVAL(cons(1.6, cons(1, liVide<int>()))) ;
=> Erreur pas de fonction cons(float, list<int>)
EVAL(tete(liste({1, 3, 7}))) ;
=> Valeur de tete(liste({1, 3, 7}))) : 1
EVAL(queue(cons(3, cons(7, liVide<int>())))) ;
=> Valeur de queue(cons(3, cons(7, liVide<int>()))) : (7)
EVAL(estVide(queue(cons(7, liVide<int>())))) ;
=> Valeur de estVide(queue(cons(7, liVide<int>()))) : true
EVAL(tete(liVide<int>())) ;
=> ERREUR application de la fonction tete(list<T> l)

Exemple (fonction sur les listes en C/C++)

Algorithme : appartientLi
Données : e : Nombre, li : liste de Nombres
Résultat : Booléen qui vaut `true` si `e` est un élément de la liste `li`, `false` sinon.

Le résultat est :
```
cond( estVide(li), false,
cond( tête(li)=e, true,
appartientLi(e,queue(li))))
```
fin algorithme

```
bool appartientLi(int e, list<int> li)
{
  return
    estVide(li) ? false :
    tete(li) == e ? true :
    appartientLi(e,queue(li));
}
```