Introduction à la programmation
TD du 13 octobre 2005

Jusqu'à maintenant, nous avons utilisé Maple comme un logiciel de maths, mais il permet aussi de faire de la programmation. Maple connaît un grand nombre de fonctions, comme `plot`, `sqrt`, `floor`, `factor`... Derrière chacune de ces fonctions, un *algorithme* plus ou moins compliqué a été implémenté par un informaticien. Nous allons dorénavent essayer décrire nous-même certains algorithmes simples sans utiliser les fonctions toutes faites de Maple.

1 *Types* d'objets Maple

En informatique, on manipule toutes sortes d'objets: nombres, équations, images, sons... L'ordinateur les traîte de façon très différente et doit savoir les reconnaître. Essayez: `igcd(10,15);` puis `igcd(10.0,15);` (`igcd` donne le pgcd de deux entiers...). Quel est le problème rencontré?
Maple reconnaît plus de 200 types. Quels sont les principaux?

1.1 Les types les plus célèbres

Pour connaître le type d'un objet Maple, on utilise la commande `whattype(`*objet*`);`
On peut aussi utiliser `type(`*expr*`,`*type*`);` et Maple nous répond *true* ou $false$...

Les types de nombres

- *integer*: nombre entier
  Quel est le type de $12.4 + 3 - 12.4$?
- *fraction*: fraction (de nombres entiers)
  Essayez `type(22/7,fraction);`
  Attention: les entiers et les nombres à virgule (même s'ils peuvent être écris comme fraction de 2 entiers, par exemple: 12,4=124/10 pour le mathématicien...) ne sont pas de type $fraction$.
- *float*: nombre avec une virgule (c'est-à-dire un point!). On les appelle *float* (en français, on dirait "nombre à virgule flottante") parceque l'ordinateur voit le nombre comme un couple $(mantisse, exposant)$ (par exemple, 0.00235907 est vu comme $(235907, -8)$).
- *rational*: rassemble les types $fraction$ et *integer*.
  Attention: ce n'est pas l'ensemble des rationnels! Les ordinateurs ne sont pas doués pour la distinction rationnel/irrationnel. D'une certaine façon, un ordinateur ne "connaît" que les nombres décimaux (c'est-à-dire les

réels qui ont un nombre fini de chiffres après la virgule), car il a une place limitée pour stocker chaque nombre.

- *numeric*: rassemble les types $float$ et *rational*.
- *complex*: nombres complexes (ce type a des raffinements...).
- *constant*: un type vaste. Grossièrement, les expressions qui ne contiennent pas d'inconnues. Les constantes symboliques de Maple comme `Pi` ou `infinity` sont des constantes (ce ne sont pas des $float$, elles sont évaluées seulement quand on force Maple à le faire...).

  Essayez: `whattype(Pi); type(Pi,float); type(Pi,constant); type(3+I,numeric); whattype(3+I); type(3+I,constant);`

Symboles et chaînes de caractères

- *symbol*: toute suite de chiffres, lettres et "underscore"(_) qui ne commence pas par un chiffre. On utilise couramment les symboles quand on fait du Maple. Dès qu'on affecte une valeur à un symbole, son type devient celui de sa valeur.
  Essayez: `whattype(grrr_09); grrr_09:=523; whattype(grrr_09);`
- *string*: (=chaînes de caractères) s'écrit entre des guillemets (") et n'a d'autre valeur qu'elle même. Elle est toujours évaluée telle quelle.
  `whattype("grrr_09"); whattype(b); b="grrr_09"; whattype(b);`

Expressions algébriques et relations

- Expressions algébriques:
  Maple connaît trois types d'expressions algébriques: +, * et $\wedge$.
  Quel est le type de: `x+y, x-y, 2*y, 3*5, x/y, -z, 1/x, x^4, sqrt(2), 3*x+2, (x^4+y)/(3*z-t)`?
- Relations d'égalité:
  En Maple, les (in)égalités s'écrivent comme ceci: =, <, > (facile...), <= pour $\leq$, >= pour $\geq$ et <> pour $\neq$.
  Quel est le type de `3*x<=y^z, c=3*d`?
- Relations logiques: *and*, *or* et *not*
  `whattype(A and B);`

1.2 Arrêt sur les "amas" d'objets: ensembles, séquences, listes...

Nous mettons la lumière sur trois types fondamentaux et leur utilisation.

Séquences (*exprseq*)

Une séquence est une suite ordonnée d'expressions séparées par des virgules, qui peuvent être de natures très différentes.
Essayez `a,3,x,tg,"grrr_09",grrr_09,x->sin(x),x>2; whattype(%);`
On peut assigner rapidement des valeurs à plusieurs variables en utilisant les

séquences: `p,q,r:=3,x->cos(x^2),"aie!";`
La séquence vide s'appelle `NULL`.

Listes (*list*)

Une liste est une suite entourée par des crochets.
Essayez `[a,x,grrr_09,"grrr_09"]; whattype(%);`
En revanche, on ne peut pas affecter des valeurs aux éléments d'une liste comme on le ferait avec une séquence. Essayez `[a,b,c]:=[1,2,3];`...

Commandes sur les séquences et les listes

- Conversions:
  Passer d'une séquence à une liste: `[`*sequence*`];`
  Passer d'une liste à une séquence: `op(`*liste*`)` (sort les *opérandes* de la liste).
  Essayez `sequence:=a,b,c; liste:=[sequence]; op(liste);`

- Obtenir le n-ème élément d'une séquence ou d'une liste P: `P[n]`
  On peut aussi utiliser `op(n,P)` pour une liste ou `op(n,[P])` pour une séquence (de toute façon, `op` ne mange que des listes et ne recrache que des séquences).

- Obtenir une sous-séquence à partir d'une séquence S: `S[n1..n2]` ou `op(n1..n2,[S])`.
  Essayez `A:=2,5,3,1,4; op(1..3,[A]); A[4..5];`

- Obtenir une sous liste à partir d'une liste L: `L[n1..n2]` ou `[op(n1..n2,L)]`

- Dans une liste de listes, obtenir le n2-ème élément de la n1-ème liste LdL: `op(n2,op(n1,LdL))` ou `LdL[n1,n2]`
  Exemple: `LdL:=[[a1,a2,a3],[b1,b2],[c1,c2,c3]]; LdL[1..2]; op(2,LdL); op(1,op(3,LdL)); LdL[2,1];`

- Concaténer deux séquences $S1$ et $S2$: `S1,S2`

- Concaténer deux listes $L1$ et $L2$: `[op(L1),op(L2)]` (On décompose en deux séquences qu'on concatène puis on en refait une liste...)

- Nombre d'éléments d'une liste L: `nops(L)`

- Nombre d'éléments d'une séquence?

- Remplacer le n-ème élément d'une liste L par une valeur x: `subsop(n=x,L)`
  Exemple: `L:=["bien!","mal!","bien!"];` Remplacez le deuxième élément par le troisième.

- Créer une séquence définie par une formule: `seq(f(i),i=X)` (X est un intervalle, une liste...)
  Essayez: `seq(i^2,i=1..5);` et `X:=[seq(i,i=0..6)]; seq(cos(i*Pi),i=X);`

- Créer une séquence avec n fois un même élément x: `x$n`

- Trier une liste de nombres L par ordre croissant: `sort(L)`

Ensembles (*set*)

Un ensemble s'écrit comme une séquence, entourée d'accolades {}. Dans un ensemble, il n'y a ni ordre ni répétition possible d'un élément.
Essayez: `{a,x,x,u,v,w,1,3,6,3,u,w};`
Comme précédemment, on peut prendre les éléments d'un ensemble avec la commande `op`. On peut faire l'union, l'intersection, etc... de deux ensembles:
`A union B; A intersect B;`

2 La syntaxe de Maple pour la programmation

2.1 La logique sous Maple

Nous allons regarder un nouveau type Maple qui permet de faire de la logique: *boolean*. Une expression booléenne contient des égalités, des inégalités, des "et", "ou", "non", etc... On évalue sa véracité avec la commande `evalb`.
Essayez: `type(1<2 and x>5,boolean); evalb(1<2 and x>5);`
`x:=3; evalb(1<2 and not x>5);`
On remarque que Maple essaie de répondre *true* ou *false*. Lorsqu'il ne peut pas, il rend une expression booléenne dont la valeur est la réponse recherchée.

2.2 Les commandes *if* et *for*

Nous allons voir les deux commandes de base de l'informatique, qui permettent de faire toutes sortes de programmes: *if* et *for*.

La commande conditionnelle: *if*

L'idée est de donner un ordre à l'esclave qu'est Maple, cet ordre pouvant varier en fonction des circonstances.
Exemple: "Si (la pièce est sale) alors (nettoie la pièce) sinon (fait la cuisine)".
Sous Maple, la syntaxe est la suivante:
`if` *conditionbooleenne* `then` *sequenceinstruction*(*s*)
    (`elif` *conditionbooleenne* `then` *sequenceinstruction*(*s*))
    (`else` *sequenceinstruction*(*s*))
`end if;`

sequenceinstruction(*s*) est comme son nom l'indique une suite d'instructions séparées par une virgule.
conditionbooleenne est une proposition logique avec éventuellement des *and*, *or*, *not*...
Elles n'ont pas besoin d'être terminées par `;` ou `:`, ni d'être mises entre accolades ou parenthèses, etc...

Remarque: si on termine par `fi:`, le résultat ne sera pas affiché...

Répéter des opérations: *for*

Souvent en algorithmique, il y a des opérations qu'on veut répéter un grand nombre de fois.

`for` *nom* `from` *expr* `to` *expr* `by` *expr*
`do` *sequenceinstruction*(*s*) `end do;`

OU

`for` *nom* `in` *expr* (liste, séquence, ensemble...)
`do` *sequenceinstruction*(*s*) `end do;`

Remarque: on peut ommettre `from` *expr* et `by` *expr*.
Leur valeur par défaut est 1.

3 Exercices

Exercice 1.

1. Construisez une liste appelée N_2 (variables indexées: la syntaxe est `v[i]` pour v_i), composée des carrés de nombres entiers qui sont compris entre 1000 et 10000.
2. Créez une liste N_3 des cubes compris dans ce même intervalle.
3. Combien y a-t-il d'éléments communs entre ces deux listes?
4. Créez un ensemble E contenant les parties fractionnaires des éléments de N_2 et N_3 divisés par 1000.

Exercice 2.

Ecrire une fonction qui prend deux nombres en argument, et renvoie "Hourrah!!" si les deux nombres sont premiers, le pgcd des nombres et "non!!" si les deux sont non-premiers, et "bof" sinon (utiliser la commande `isprime`).

Exercice 3.

1. Trouver tous les triplets d'entiers (a, b, c) tels que $a^2 + b^2 = c^2$ avec $c \leq 1000000$.
2. Faire de même en remplaçant les carrés par des cubes, des puissances de 4 et de 5.