TD 2 : Structures de contrôle
Octobre 2011

Après avoir vu comment calculer des choses simples numériquement ou formellement, nous allons nous intéresser au bases de la programmation en Maple.

Bases de programmation

Maple offre un environnement sommaire de programmation. Les briques de base que vous allez manipuler pour programmer sont les procédures. Une procédure est un groupe d'instructions qui forme une commande Maple que vous pourrez ensuite appeler comme une fonction. On utilisera d'ailleurs fréquemment l'abus de language "fonction" pour parler d'une procédure.

Création d'une procédure

Une procédure se crée en 6 étapes :

1. Choisir un nom dans lequel mettre sa procédure (un nom de variable)
   `nom :=`
2. Écrire les arguments (paramètres) reçus par la procédure
   `proc(arg_1, ... , arg_n)`
3. Déclarer à Maple les variables locales
   `local var_1, ... , var_n ;`
4. Déclarer à Maple les variables globales
   `global varg_1, ... , varg_p;`
5. Écrire le corps de la procédure
   `instruction_1;`
   `...`
   `instruction_m;`
6. Terminer sa procédure
   `end proc;`

Si on met tout ensemble, la syntaxe pour créer une procédure est la suivante :

```
>nom := proc (argument_1,...,argument_n)
  local var_1,...,var_i;
  global var'_1,...,var'_j;
  instruction_1;
  ...
  instruction_p;
end proc;
```

Par rapport aux fonctions vues au TD1, les procédures ont l'avantage de pouvoir contenir des instructions de contrôle – que l'on va voir à la section suivante – et des variables intermédiaires. Utilisez `Maj + Entrée` pour revenir à la ligne sans évaluer le contenu de la procédure que vous êtes en train d'écrire.

Au niveau syntaxique, faites bien attention aux mots clés (`proc`, `local`, `end proc`). Remarquez aussi dans l'exemple de synthèse que des espaces ont été placés au début de certaines lignes : cette pratique se nomme l'*indentation* et est essentielle à quiconque qui veut que son code soit lu !!!

Pour indenter correctement – modulo quelques habitudes dogmatiques – son code, il suffit d'introduire deux espaces supplémentaires en début de ligne pour chaque blocs ouverts ($\simeq$ chaque instruction de contrôle imbriquée, voir juste aprés).

Arguments, variables locales et globales Il est important de comprendre la différence entre les trois notions :

Les arguments sont des valeurs passées à la fonction. En tant que tel, on ne peut pas modifier directement le contenu d'un argument.

Les variables locales sont définies uniquement pour la portée de la procédure (jusqu'au `end proc`). Elles ne sont pas corrélées à l'environnement de la procédure (c'est à dire qu'une variable locale peut avoir le même nom qu'une variable définie avant la procédure sans en partager le contenu).

Les variables globales correspondent à des variables définies dans le contexte. Elles nous seront très utiles quand nous nous intérresserons aux effets de bords dans quelques TDs.

Structure de saut : le *RETURN*

Une procédure, comme une fonction, renvoit un résultat qui consiste en la dernière chose calculée avant le `end proc`. Lorsque la procédure est très compliquée, on peut avoir besoin de renvoyer explicitement le résultat en interrompant la procédure à un certain point. Cela se fait via un appel à `RETURN(...)` en précisant la valeur à retourner entre les paranthèses.

Structure conditionnelle : le *if*

Les branchements conditionnels sont des structures de contrôle permettant de tester une condition puis d'agir diffèrement selon que la condition soit vraie ou fausse. Par exemple :

`Si` Dark Vador est là
`Alors` sortir son sabre laser
`Sinon Si` il pleut
    `Alors` sortir son parapluie
    `Sinon` sortir son chien

La syntaxe à employer dans Maple est `if` ... `then` ... `else` ... `end if`. De plus, les clauses de la forme `else if` peuvent s'abréger en `elif`, ce qui donne pour la forme générale :

```
if condition then instruction;
elif autre_condition then autre_instruction;
elif ...;
else derniere_instruction;
end if;
```

Notez que la clause `else` n'est pas obligatoire : une construction `if` ... `then` ... `end if` est parfaitement valide.

Exercice 1

Écrire une procédure `bidule:=proc(x)` *qui renvoie* $x/2$ *si* x *est pair et* $3x+1$ *sinon.*

Aparté : les conditions

La plupart des structures de données sont paramétrées par une condition. Par condition, on entend une expression Maple qui peut s'évaluer soit à `true` soit à `false` (valeur booléenne). On construit de telles expressions :

- ▷ soit à partir d'une variable booléenne (qui contient donc `true` ou `false`)
- ▷ soit en utilisant un opérateur de comparaison (=, <>, <, <=, >, >=) entre deux expressions (souvent des nombres mais ce n'est pas une obligation)
- ▷ soit en employant un prédicat (fonction à valeur booléenne) donnée par Maple ou fabriquée par vos propres soins (voir plus loin dans le TD la fonction `isprime`)

▷ soit en combinant deux conditions avec les opérations booléennes suivantes : `and` (et), `or` (ou), `not` (négation)

Structures itératives : le *while* et le *for*

Les boucles itératives vous permettent de réaliser des opérations répétitives *tant qu'* une certaine condition n'est pas remplie (boucle `while`) ou *pour toute* valeur dans un certain intervale (boucle `for`).

Boucle for Sa syntaxe générale est :

```
>for variable from début to fin by pas do
   instruction;
   ...;
 end do;
```

Maple exécutera les instructions entre le `do` et le `end do` pour chaque valeur de la variable du `for` entre les bornes de début et de fin, par saut de *pas* (par défaut cette valeur vaut 1).

Exercice 2

Écrire une procédure `chose:=proc(n)` qui affiche tous les entiers pairs entre 0 et n. On utilisera une boucle `for` et `print(k)` pour afficher l'entier k.

Boucle while On l'écrit de la façon suivante :

```
>while condition do
   instruction;
   ...;
 end do;
```

Les instructions comprises entre le `do` et le `end do` seront effectuées en boucle tant que la condition du `while` sera vérifiée.

Exercice 3 (*Équivalence des boucles for et while*)

Refaire l'exercice précédent en remplaçant la boucle `for` par une boucle `while`.

Exercice 4

Indentez correctement le code suivant :

```
> f:=proc(n,z) local a,b,i;if z then a:=0;b:=1;for i from 1 to n do a:=a+b;b:=a-b;
  end do;a;else a:=n;b:=0;while b < a do while b < a do b:=b+1;print(42);
  end do;a:=a-1;b:=0;end do;end if;end proc;
```

Bonus : *à défaut d'être lisible, ce code est complétement valide. Que fait t'il ?*

Vos premières structures de données : suites et listes

Les suites et les listes sont deux façons de ranger plusieurs éléments ensemble en Maple. Bien que se ressemblant beaucoup, elles n'ont pas le même usage.

Les Suites Une suite est simplement un ensemble fini d'éléments donnés dans l'ordre, séparés par des virgules.

On peut assigner rapidement plusieurs variables grâce aux suites.

```
> p,q,r := 34, evalf(Pi), 2.5;
                        p, q, r := 34, 3.141592654, 2.5

> # Lorsqu'il y a plusieurs solutions le résultat de la commande solve est une liste.
  racine1, racine2, racine3 := solve (x^3+x^2+x+1,x);
                    racine1, racine2, racine3 := -1, I, -I

> racine3;
                                 -I
```

Les Listes Une liste est juste une suite entourée de crochets.

```
> a := 2 , 5, 7 , 11 , 13 , 17 , 23 ;
                        a := 2 , 5, 7 , 11 , 13 , 17 , 23
> [a];
                        [ 2 , 5, 7 , 11 , 13 , 17 , 23 ]
```

Quelques commandes sur les listes et les suites

*Les commandes présentées ici sont **fondamentales**. Si vous ne devez retenir qu'une chose de ce TD, voire de l'année, retenez ce qui suit.*

Conversion On peut transformer une suite en liste simplement en l'entourant de crochets :

```
> suite := 1,a,b;
                               suite := 1, a, b

> liste := [suite]; whattype (liste);
                              liste := [1, a, b]
                                     list
```

On utilise la fonction `op` pour transformer une liste en suite.

```
> suite2 := op(liste);
                               suite2 := 1, a, b
```

Création Pour créer une suite, on se sert de la commmande `seq(...)`. La syntaxe générale est `seq( f(i), i=debut..fin)`

```
> seq ( k^2 , k=5..20 ); # la suite des carrés de 5 à 20
   25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400
```

Accès L'accès au i-ème élément d'une liste se fait avec la notation `liste[indice]` :

```
> liste := [37,78,25];
                              liste := [37, 78, 25]

> liste[1]; liste[3];
                                     37
                                     25

> liste[-1]; # le dernier terme de la liste (-2 pour l'avant dernier, etc...)
                                     25
```

Cette notation s'emploie aussi pour extraire des sous-listes

```
> liste[2..3]; # la sous-liste composée des termes 2 à 3.
                                  [78, 25]
```

Ajout d'un élément On peut également construire une liste en ajoutant des éléments un par un. On le fait de la manière suivante :

```
> liste := [3,5,7,11];
                                    liste := [3, 5, 7, 11];

> liste := [op(liste),13];
    liste := [3, 5, 7, 11, 13];
```

Des choses plus exotiques On utilise `nops` pour obtenir le nombre d'éléments dans une liste ou une suite. On peut bien entendu faire des listes de listes, concaténer deux suites (attention pour les listes, il faut utliser `op`). On utilise `sort` pour trier une liste.

Exercices

Exercice 5 (*Rappels*)

1. *Définir sous forme d'une fonction f le polynôme* $x^3 + 4 * x - x + 5$.
2. *Calculer la valeur de f en 1 et en 7. Développer et simplifier f(a+b).*
3. *Dériver les expressions (commande* `diff`*)* $\ln(x) + e^{5*x}$, $g(x) = \sin(x) - \cos(x) + tan(x)$
4. *Trouver la dérivée troisième de* $g(x)$.
5. *Soit* $g(x, y) = \frac{x^3 y^2}{x^4+y^2+1}$. *Calculer les dérivées partielles de g.*
6. *Déterminer un développement limité à l'ordre 4 en 0 de* $cos(exp(x))$
7. *Calculer une primitive de* $f(x)cos(x)$.
8. *Calculer l'intégrale entre 1 et 2 de cette même fonction.*

Exercice 6

1. *Construire la liste (ordonnée) des 200 premiers nombres premiers (la fonction* `ithprime(i)` *renvoie le i-ème nombre premier).*
2. *Diviser tous ces nombres par 17 et appliquer la fonction* `frac(...)` *pour ne garder que la partie fractionnaire des nombres (vous pouvez regarder la fonction* `map`*).*
3. *Refaire rapidement les deux premières questions en utilisant des procédures et des boucles* `for` *(ou si c'est déjà le cas en employant la commande* `seq`*).*

Exercice 7

1. *Sans utiliser* `ithprime`*, écrire une procédure* `n_prem := proc(n)` *rendant la liste des n premiers nombres premiers. On pourra cependant utiliser* `isprime(n)`*, à moins de vouloir implémenter directement le crible d'érathostène.*
2. *En utilisant la procédure ci dessus, écrire une procédure* `double_somme := proc(n)` *rendant la liste des sommes des paires de nombres premiers entre 1 et n.*
3. *En utilisant la procédure ci-dessus, montrer que tous les nombres pairs entre 1 et 1000 s'écrivent sous la forme d'une somme de deux entiers premiers.*
   Ce problème est connu sous le nom de "conjecture de Goldbach" : tout nombre pair strictement supérieur à 2 peut-il s'écrire comme somme de deux nombres premiers ?
4. Bonus *: Démontrer la conjecture de Goldbach.*

Exercice 8

1. *Créer une liste* L_2 *constitué des carrés compris entre 1000 et 10000.*

2. *Créer une liste L_3 constitué des cubes compris entre 1000 et 10000.*
3. *Combien y-a-t'il d'éléments en commun dans les deux listes L_2 et L_3. Combien y-a-t'il d'éléments dans les deux listes réunies ?*

Exercice 9

En utilisant les fonctions `irem` *et* `iquo` *(respectivement le reste et le quotient de la division euclidienne), écrire un programme rendant une suite représentant la décomposition d'un entier $n > 0$ en base b.*

Exercice 10

- *Écrire un programme qui rend une liste représentant la décomposition d'un entier n en facteurs premiers.*
- *Écrire un programme prenant en entrée la décomposition d'un entier en facteur premier et rendant l'entier en question.*
- *Vérifier que ces deux programmes sont bien l'inverse l'un de l'autre.*
- Bonus : *Prouver que ces deux programmes sont bien l'inverse l'un de l'autre.*

Un dernier point de syntaxe

Dans la présentation précédente des instructions de contrôles, chaque structure de contrôle finissait par quelque chose de la forme `end bidule`. Il en fait possible de mettre des mots clés plus courts : `od` à la place de `end do`, `fi` à la place de `end if`, `end` à la place de `end proc`.