TP4

Programmation

1 Définition d'une procédure

Une **procédure** est un groupe d'instructions Maple qui sont placées dans un même bloc d'instructions débutant par un *nom*`:=proc()` et se terminant par un `end;`. Lorsqu'on appelle la procédure, les instructions qui la composent sont exécutées les unes après les autres.

Sur le plan technique, une procédure se présente sous la forme suivante :

```
> nom:=proc(arg1, arg2, ..., argn)
  local var1, var2, ..., varm;
    instructions;
  end;
```

ou bien

```
> nom:=proc(arg1, arg2, ..., argn)
  local var1, var2, ..., varm;
    instructions;
  end proc;
```

où $arg_1 \ldots$ désignent les arguments appelés dans la procédure, et $var_1 \ldots$ désignent les variables utilisées dans le programme.

Comme on le voit dans cette définition, toute la procédure doit être entrée dans un seul et même bloc d'instructions. On rappelle que pour rester dans le même bloc d'instructions, au lieu de taper **Entrer** en fin de ligne, on tape **Majuscule + Entrer**, qui provoquent un simple retour à la ligne sans changer de bloc.

On appelle une procédure comme on appelle une fonction Maple :

```
> nom(arg1, arg2, ..., argn);
```

Voyons tout de suite un premier exemple de procédure :

```
> exemple:=proc(n)
  (n+1)^2;
  end;
```

$$exemple := \mathbf{proc}(n)\ (n+1)^2\ \mathbf{end\ proc}$$

Notons que lorsqu'il n'y a pas d'erreurs de syntaxe grossières dans la procédure que vous venez d'entrer, Maple la reproduit en Output lorsque vous la validez. On peut alors lancer la procédure que nous venons de créer comme suit :

```
> exemple(5);
```

36

On remarque que Maple renvoie le dernier résultat calculé. Nous reviendrons là-dessus au §2. On vérifie aussi que, comme dans toute les fonctions Maple, si l'on omet l'argument, le système renvoie un message d'erreur.

```
> exemple();
  Error, (in exemple) exemple uses a 1st argument, n, which is missing
```

La procédure `exemple` que nous venons de créer était simple et nous n'avons pas eu besoin de recourir à des variables locales. Voici un exemple de procédure (division euclidenne) avec des variables locales :

```
> div:=proc(a,b)
  local c,d;
    c:=iquo(a,b);
    d:=irem(a,b);
  end;
```

$$div := \mathbf{proc}(a,b)\ \mathbf{local}\ c,d;\ c := iquo(a,b);\ d := irem(a,b);\ \mathbf{end\ proc}$$

Les variables locales ne sont affectées qu'à l'intérieur de la procédure, c'est-à-dire que si une variable du même nom est définie dans la feuille de calcul Maple, sa valeur n'est pas modifiée lors de l'exécution de la procédure. C'est ce que nous constatons ici :

```
> c:=0; div(14,3); c,d;
```

c:=0
2
0,d

Remarquons par ailleurs que Maple renvoie à la fin d'une procédure le dernier résultat calculé. Il existe des outils qui permettent d'afficher en sortie d'une procédure d'autres résultats que le seul dernier résultat. C'est ce que nous voyons dans le paragraphe suivant.

2 Entrées et sorties de données dans une procédure

La définition d'une procédure nous a déjà donné une méthode pour entrer des données dans une procédure. La deuxième méthode permet un échange entre l'utilisateur et Maple, au cours duquel le système pose une question en clair à l'utilisateur. Pour cela, on utilise la fonction `readstat`. Cette fonction affecte à *var* la réponse à la question :

```
> var:=readstat(‘Entrez votre question‘);
```

Considérons tout de suite un exemple d'utilisation de cette fonction :

```
> essai:=proc()
  local nom;
  nom:=readstat(‘Quel est ton nom?  ‘);
  cat(‘Bonjour ‘,nom);
  end;
```

essai:=**proc**() **local** *nom*; *nom*:=readstat(‘*Quel est ton nom*?‘);cat(‘*Bonjour* ‘,*nom*); **end proc**

L'utilisation de la procédure conduit alors à :

```
> essai();
  Quel est ton nom?  Lionel;
```

Bonjour Lionel

On a vu au paragraphe précédent que les procédures renvoient de manière systématique le dernier résultat calculé. La première possibilité d'obtenir un autre résultat est d'utiliser la fonction `RETURN`. Cette fonction renvoie alors ses arguments et quitte la procédure. Tout ce qui est situé après le `RETURN` sera donc non exécuté et ignoré. Voici un exemple de l'utilisation de la fonction `RETURN` :

```
> essai2:=proc(x,y);
  RETURN(x*y);
  x+y;
  end;
```

*essai*2:=**proc**(x,y) RETURN(x*y); x+y; **end proc**

On observe bien ce à quoi on s'attendait :

```
> essai2(3,2);
```

6

Nous pouvons aussi utiliser la fonction `print` pour afficher un résultat. Toutefois, cette fonction n'empêche pas l'affichage du dernier résultat calculé, et ne quitte pas la fonction, comme le faisait la fonction `RETURN`. Voici un exemple :

```
> essai3:=proc(x,y);
  print(x*y);
  x+y;
  end;
```

*essai*3:=**proc**(x,y) print(x*y); x+y; **end proc**

```
> essai3(3,2);
```

6
5

3 Structures conditionnelles et itératives

3.1 Structures conditionnelles

Pour tester des propriétés dans une procédure en Maple, on doit utiliser une boucle `if`. Cette boucle doit être fermée par un `fi` ou `end if`.
La syntaxe générale d'une boucle `if` est :

```
> if condition_1 then
    instruction_1;
  elif condition_2 then
    instruction_2;
  ...
  else
    instruction_k;
  fi;
```

Seul le premier `if` et le `fi` sont obligatoires : `elif` et `else` sont des éléments facultatifs d'une telle boucle. Les *conditions* sont des conditions booléennes et pourront faire intervenir les connecteurs logiques (`or, and...`) pour former un groupe de conditions coordonnées.
Voici un exemple de procédure qui donne le maximum entre deux nombres réels (notons qu'on aurait pu utiliser directement la fonction `max`):

```
> maximum:=proc(a,b)
  if a>b then
  a;
  else
  b;
  fi;
  end;
```

maximum:=**proc**(a,b) **if** b < a **then** *a* **else** *b* **end if end proc**

Il est possible de faire une programmation récursive avec la fonction `RETURN`. Cette programmation est particulièrement adaptée aux suites.
Donnons ici l'exemple de la suite de Fibonacci définie par $u_0 = u_1 = 1$ et $u_n = u_{n-1} + u_{n-2}$:

```
> u:=proc(n)
  option remember
  if n=0 then RETURN(1);
  elif n=1 then RETURN(1);
  else RETURN(u(n-1)+u(n-2));
  fi;
  end;
```

u:=**proc**(*n*) **option** remember **if** $n = 0$ **then** RETURN(1) **elif** n=1 **then** RETURN(1) **else** RETURN($u(n-1) + u(n-2)$) **end if end proc**

Cette procédure fait appel à elle-même pour calculer le terme u_{n-1}, u_{n-2} et ainsi de suite. L'option `remember` permet un gain de temps lors de l'exécution de la procédure. En effet, dans ce cas Maple stocke ses calculs; ce qui permet de calculer une seule fois chaque terme.

3.2 Structures itératives

Le principe de la boucle itérative est de faire une boucle qui va réaliser une action un certain nombre de fois ou tant qu'une condition est réalisée. On appelle aussi ce type de structure les boucles `for` et `while`. La syntaxe de ce genre de boucle est :

```
> for var from ini to fin by pas do
    instruction1;
    ...
  od;
```

et

```
> while condition do
    instruction1;
    ...
  od;
```

Ici, seuls `do` et `od` (ou `end do`) sont indispensables. Pour la boucle `for`, on n'est pas obligé de mettre une variable, ni de borne de départ (par défaut cette borne est fixée à 1), ni de borne d'arrivée (par défaut Maple boucle à l'infini), ni de pas (par défaut il vaut 1). Pour la boucle `while`, la *condition* permet de fournir une condition pour rentrer dans la boucle.
Si l'on omet tous ces éléments, il faudra bien prendre garde à ce que la boucle ait une fin. Signalons donc l'existence d'une commande "STOP" qui permet de stopper une procédure en cours d'exécution, afin de revenir à la feuille de calcul sans trop de dommages.
Pour terminer, voici deux exemples de procédure qui vous montreront comment utiliser l'une ou l'autre de ces boucles: la première donne la liste des nombres premiers inférieurs ou égaux à 100, la deuxième permet de calculer le pgcd de deux entiers.

```
> premier:=proc()
  local s,k,L;
  s:=NULL;
  for k from 1 to 100 do
  if isprime(k) then
  s:=s,k;
  fi;
  od;
  L:=[s];
  end;
```

et

```
> pgcd:=proc(A,B)
  local a,b,r;
  a:=abs(A); b:=abs(B);
  while b<>0 do
  r:=irem(a,b);
  a:=b;
  b:=r;
  od;
  a;
  end;
```

Exercice 1 Écrire deux procédures `som` et `prod` qui, étant donné un entier naturel n, calculent respectivement la somme $\sum_{k=0}^{n} k^4$ et le produit $\prod_{1\leq p<q\leq n} \frac{p}{p+q}$.

Exercice 2 Écrire une procédure récursive `facto` qui permet de calculer $n!$, pour un entier naturel n donné, en n'utilisant que l'opération de multiplication.

Exercice 3

1. Écrire une procédure `divis` permettant d'obtenir la liste des diviseurs, rangés dans l'ordre croissant, d'un entier naturel non nul donné.
2. Écrire une procédure `decomp` qui donne la décomposition en produit de nombres premiers, d'un entier naturel non nul donné.
3. Écrire une procédure `parfait` qui, étant donné un entier naturel N, cherche les nombres parfaits $\leq N$ (n est parfait si la somme de ses diviseurs vaut $2n$).
4. Deux nombres naturels a et b sont dit jumeaux si la somme des diviseurs de a distincts de 1 et de a est égale à b et vice-versa. Écrire une procédure `jumeaux` qui détermine tous les couples de jumeaux (a, b) vérifiant $a \leq b \leq N$, pour un entier N donné.
5. Fermat (1601-1665) avait conjecturé que tous les nombres de la forme $2^{2^k}+1$, avec k un entier naturel, étaient premiers. Écrire une procédure `fermat` permettant d'obtenir les nombres de Fermat successifs jusqu'au premier qui ne soit pas un nombre premier; donner alors la décomposition primaire de celui-ci.

Exercice 4 Écrire une procédure `nombre` qui demande à l'utilisateur un nombre premier inférieur ou égal à 100, et, en cas de réponse incorrecte, affiche "SVP un nombre premier inférieur à 100" jusqu'à ce que la réponse soit bonne.

Exercice 5 On cherche à calculer a^n avec $a \in \mathbb{R}$ et $n \in \mathbb{N}$, en n'utilisant que des multiplications et avec le moins d'opérations possibles.

1. Méthode naïve : on multiplie a par lui-même n fois de suite.
   - Combien faut-il de multiplications?
   - Programmer cette procédure `puissance` sur Maple.
2. Exponentiation rapide : avec l'écriture en base 2 de n. Par exemple, $a^{13} = a^{1+4+8} = a \times (a^2)^2 \times ((a^2)^2)^2$. De façon générale, si $n = \sum_{k=0}^{p} x_k 2^k$, avec $x_k \in \{0, 1\}$, alors

$$a^n = a^{\sum_{k=0}^{p} x_k 2^k} = \prod_{k=0}^{p} a^{2^k x_k} = \prod_{k=0}^{p} (a^{2^k})^{x_k} = \prod_{k, x_k=1} a^{2^k}$$

   Programmer un algorithme `exporapide` d'exponentiation rapide. Compter le nombre de multiplications nécessaire avec cette méthode. On pourra utiliser la fonction `time` pour chronométrer les deux procédures avec 2^{13} et $1.1^{1000000}$.