TP d'AP2 (algorithmique et programmation 2)
Université de Lorraine, site de Nancy,
Licence première année, S2 : M-I et I-SPI

Site du cours : `http://homepages.loria.fr/JLieber/cours/ap2/`
responsable de l'UE : Jean Lieber Dernière version : 12 janvier 2019

Remarques générales sur les travaux pratiques

L'objectif principal des travaux pratiques est de vous faire passer de l'algorithmique et programmation sur papier (cf. enseignements intégrés) à une programmation sur machine.

Ce document regroupe les énoncés de toutes les séances de TP (sauf la dernière qui constitue un examen de TP). Les exercices sont de difficultés variés : certains exercices ne nécessitent que quelques minutes, d'autres prendront plusieurs heures.

Préparation des TP

Une séance de travaux pratiques doit être préparée avant cette séance. Vous devez avoir lu l'énoncé en entier. Si un point de l'énoncé n'est pas clair, vous pourrez demander une explication aux enseignants (d'EI ou de TP). Il faut aussi préparer au moins une partie des TP sur feuille (au moins le premier exercice).

Pendant les séances de TP

Les TP se feront dans un environnement de programmation simple consistant en :
- Un éditeur de texte adapté – ou adaptable – aux programmes C, tel que emacs (mise en évidence des mots-clefs du langage, aide à l'indentation, etc.) ;
- Une fenêtre terminal pour la compilation et l'exécution des programmes ;
- Un navigateur pour pouvoir téléverser vos travaux sur Arche ;
- Sous forme papier : les notes de cours et les énoncés de TP, ainsi que les papiers sur lesquels vous avez préparé les TP et du papier brouillon (ainsi que de quoi écrire).

Toute activité ne concernant pas les TP (tel que la consultation de sites non indiqués dans les énoncés) est interdite.

Les intervenants en TP sont là pour vous aider : en cas de problème, il faut les solliciter. De plus, il est utile de leur demander de temps en temps des conseils sur leur programme : même si votre programme compile correctement et donne le résultat attendu, il peut être améliorable (lisibilité, efficacité, etc.).

Tout programme doit :
- Avoir été testé (si vous écrivez une fonction `f`, il faut aussi écrire une procédure `test_f_param` faisant appel à `f` avec la même liste de paramètres que `f` et une procédure `test_f` faisant appel à `test_f_param` et sans paramètre : le jeu de test choisi sera défini dans cette dernière procédure) ;
- Respecter les règles de lisibilité (commentaires, indentation, choix soigneux du nom des fonctions, procédures et variables, sauts de ligne entre les fonctions, etc.).

À la fin d'un TP il faut mettre ce que vous avez produit et qui est correct et testé sur Arche. L'exécution de votre programme sera mise en commentaire dans votre programme.

---

0 À titre d'exemple (0 séance)

Cette section donne un exemple de ce qui est attendu pour le rendu d'un TP.

0.1 Énoncé

Soit la suite réelle $(u_n)_n$ définie par

$$u_0 = 1$$
$$u_{n+1} = \frac{u_n}{2} + \frac{1}{u_n} \qquad \text{pour tout entier naturel } n$$

Implantez de façon récursive et en C cette suite par une fonction `u`.

Testez cette fonction avec les valeurs suivantes du paramètre : 0, 1, 2, 10, 20 et 100. Vous mettrez le résultat en commentaire dans le fichier C.

Que constatez-vous ? Répondez également à cette question en commentaire.

0.2 Corrigé

Le fichier `tp0.c`, également accessible depuis la page Web du cours, est reproduit ci-dessous :

```
/* tp0.c
   Corrigé de l'énoncé de TP numéro 0 (AP2)
   auteur : Jean Lieber
   date : 10 janvier 2018
*/

#include <stdio.h>
#include <stdlib.h>

double u(unsigned int n)
{
  double unm1 ;
  if (n == 0)
    {
      return 1.5 ;
    }
  unm1 = u(n-1) ;
  return unm1 / 2. + 1. / unm1 ;
}

void test_u_param (unsigned int n)
{
  printf("u%u = %f\n",
     n, u(n)) ;
}

void test_u()
{
  test_u_param(0) ;
  test_u_param(1) ;
  test_u_param(2) ;
  test_u_param(10) ;
  test_u_param(20) ;
  test_u_param(100) ;
}

int main()
{
  test_u() ;
  return EXIT_SUCCESS ;
}

/*
L'exécution du programme donne :
u0 = 1.500000
u1 = 1.416667
u2 = 1.414216
u10 = 1.414214
u20 = 1.414214
u100 = 1.414214

On constate que la suite semble converger
vers une valeur proche de 1,414 et donc,
proche de la racine carrée de 2.
*/
```

---

1 Fonctions et procédures (2 séances)

Ex. 1 : Traduire un algorithme sur les entiers naturels en un programme. On considère l'algorithme suivant :

```
Fonction dpn (n : entier naturel) : entier naturel
Variables
  | d : entier naturel
Début
  | Si est_nul(n) Alors
  |   | retourner succ(zéro)
  | Finsi
  | d ← dpn(préc(n))
  | retourner plus(d, d)
Fin
```

Écrivez un programme qui implante cette fonction et qui la teste, à l'aide de deux procédures (de noms respectifs `test_dpn_param` et `test_dpn`).

Que fait ce programme ? Répondez à cette question dans un commentaire du programme.

Ex. 2 : Traduire un algorithme sur les entiers naturels en un programme. On considère l'algorithme suivant :

```
Procédure afficher_entiers_jusqu_à (n : entier naturel)
Début
  | afficher_entiers_intervalle(zéro, n)
Fin

Procédure afficher_entiers_intervalle (a : entier naturel, b : entier naturel)
Début
  | Si a > b Alors
  |   | retourner
  | Finsi
  | afficher(a)
  | afficher_entiers_intervalle(succ(a), b)
Fin
```

Écrivez un programme qui implante ces procédures et qui les teste, à l'aide de deux nouvelles procédures (de noms respectifs `test_afficher_entiers_jusqu_a_param` et `test_afficher_entiers_jusqu_a`).

Ex. 3 : Traduire un algorithme sur les tableaux en un programme. On considère l'algorithme suivant :

```
Fonction est_trié (T : tableau de réel[n]) : booléen
Variables
  | i : entier naturel
Début
  | i ← 0
  | Tant que i < n − 1 et T[i] ⩽ T[i + 1] Faire
  |   | i ← i + 1
  | Fintantque
  | retourner i ⩾ n − 1
Fin
```

Écrivez un programme qui implante cette fonction et qui la teste, à l'aide de deux procédures (de noms respectifs `test_est_trie_param` et `test_est_trie`).

Ex. 4 : Traduire un jeu d'axiomes en programme récursif. La fonction `somme_cubes` est la fonction de profil `somme_cubes : entier naturel` $\longrightarrow$ `entier naturel` définie par le jeu d'axiomes suivant :

[1] $\texttt{somme\_cubes}(\texttt{zéro}) = \texttt{zéro}$ (ou $\texttt{somme\_cubes}(0) = 0$)

[2] $\texttt{somme\_cubes}(\texttt{succ}(x)) = \texttt{plus}(\texttt{puissance}(\texttt{succ}(x), \texttt{succ}(\texttt{succ}(\texttt{succ}(\texttt{zéro})))), \texttt{somme\_cubes}(x))$
(ou $\texttt{somme\_cubes}(x+1) = (x+1)^3 + \texttt{somme\_cubes}(x)$)

Q1 Traduisez ce jeu d'axiomes en programme récursif (vous pouvez, dans un premier temps, le traduire en algorithme récursif sur le papier).
Q2 Testez ce programme (par deux procédures).
Q3 Modifiez le programme pour que l'exécution de `somme_cubes` soit « tracée », ce qui signifie que l'appel à $\texttt{somme\_cubes}(n)$ doit être affiché ainsi que son résultat. Ainsi, $\texttt{somme\_cubes}(3)$ doit donner :

```
appel à somme_cubes(3)
 appel à somme_cubes(2)
  appel à somme_cubes(1)
   appel à somme_cubes(0)
   somme_cubes(0) retourne 0
  somme_cubes(1) retourne 1
 somme_cubes(2) retourne 9
somme_cubes(3) retourne 36
```

L'indentation de l'affichage (espaces en début de lignes) est optionnelle.
Q4 On veut vérifier l'égalité suivante (pour tout entier naturel n) :

$$\texttt{somme\_cubes}(n) = \left(\frac{n(n+1)}{2}\right)^2$$

Pour cela, on doit faire une preuve, mais pour se convaincre que cela vaut le coup de faire cette preuve, on peut tester cette égalité pour un nombre fini de valeurs.

Écrivez un programme qui teste cette égalité pour les valeurs de n entre 0 et 100.

Ex. 5 : Traduire un jeu d'axiomes en programme récursif et itératif. La suite de Fibonacci est la fonction de profil `fibo` : `entier naturel` $\longrightarrow$ `entier naturel` définie par le jeu d'axiomes suivant :

[1] $\texttt{fibo}(\texttt{zéro}) = \texttt{succ}(\texttt{zéro})$ (ou $\texttt{fibo}(0) = 1$)

[2] $\texttt{fibo}(\texttt{succ}(\texttt{zéro})) = \texttt{succ}(\texttt{zéro})$ (ou $\texttt{fibo}(1) = 1$)

[3] $\texttt{fibo}(\texttt{succ}(\texttt{succ}(x))) = \texttt{fibo}(x) + \texttt{fibo}(\texttt{succ}(x))$ (ou $\texttt{fibo}(x+2) = \texttt{fibo}(x) + \texttt{fibo}(x+1)$)

Q1 Traduisez ce jeu d'axiomes en programme récursif (vous pouvez, dans un premier temps, le traduire en algorithme récursif sur le papier).
Q2 Testez ce programme (par deux procédures).
Q3 Modifiez le programme précédent pour qu'à chaque appel à la fonction `fibo` soit affichée la valeur des paramètres. Ainsi, $\texttt{fibo}(2)$ doit donner l'affichage :

```
fibo(2)
fibo(0)
fibo(1)
```

Testez alors le programme pour le paramètre 5.
Que constatez-vous ?
Q4 Écrivez un nouveau programme pour `fibo` qui remédie au problème mis en évidence à la question précédente.
On demande que ce programme soit itératif.
Q4 (faculative) Écrivez un programme récursif pour `fibo` tel que l'exécution de $\texttt{fibo}(n)$ fasse appel à moins de n appels récursifs.

---

2 Récursivité et enregistrements (3 séances)

Ex. 6 : échauffement Choisissez une des deux questions suivantes.
Q1 Implantez et testez la procédure suivante :

```
Procédure affichage_binaire (a : entier_nat)
Début
    Si a ⩾ 2 Alors
        affichage_binaire(a div 2)
    Finsi
    afficher(a mod 2)
Fin
```

À titre d'exemple, l'exécution de `affichage_binaire`(45) donnera l'affichage de `101101`.
Q2 On considère le jeux d'axiomes suivant (où `BASE` est une constante) :

[1] `somme_chiffres(zéro)` $=$ `zéro`

[2] `somme_chiffres(succ(`x`))` $=$ `(succ(`x`) mod BASE)` $+$ `somme_chiffres(succ(`x`) div BASE)`

Implantez et testez la fonction `somme_chiffres`. Vous pourrez choisir `BASE` $= 10$.

Ex. 7 : Les tours de Hanoï Le jeu des tours de Hanoï est un jeu à un seul joueur qui se présente ainsi. On a 3 piliers, A, B et C disposés de gauche à droite et des disques posés sur les piliers, en respectant la règle suivante : un disque ne doit jamais être posé sur un disque de taille plus petite ou égale à lui. Initialement, les piliers B et C sont vides et le pilier A contient n disques, où $n \geqslant 1$ est un paramètre du jeu. Ainsi, pour $n = 4$, l'état initial du jeu resssemblera à [1] :

Une action consiste à déplacer un disque en sommet de pilier vers un autre pilier. Le but du jeu est de trouver une séquence d'actions partant de l'état initial vers l'état final, dans lequel tous les disques sont sur le pilier C :

L'exercice est divisé en 3 questions principales. La première vise à implanter les types de données à manipuler. La deuxième vise à implanter un programme permettant à un joueur de jouer aux tours de Hanoï. La troisième vise à implanter un programme qui résout ce jeu. On peut éventuellement changer l'ordre des deux dernières questions.
Q1 Le but de cette question est de vous faire implanter les types utiles pour représenter les piliers (avec des disques) et les états représentés par trois piliers.

Cette question est décomposée comme suit.
q1.2 Implantez le type `pilier` par un enregistrement dont les champs sont :

— `nombre_disques_max` de type `unsigned int` ;
— `tableau` : tableau de `nombre_disques_max` éléments de type `unsigned int`.

Par exemple, le pilier pouvant soutenir 4 disques et dont les disques sont, de bas en haut, de tailles 3, et 1 est (en langage algorithmique) :

$$p = \texttt{écrire\_tableau}(\texttt{[3, 1, 0, 0]}, \texttt{écrire\_nombre\_disques\_max}(4, \texttt{pilier\_vide}()))$$

1. Cette illustration a été reprise de `http://www.texample.net/tikz/examples/towers-of-hanoi/`.

Dans le tableau, `0` indique une absence de disque. La règle interdisant un disque d'être posé sur un disque plus grand ou de même taille que lui dans le pilier p signifie, pour $T =$ `lire_tableau(p)` que :

$$\text{Avec } h \text{ le nombre de disques sur le pilier,} \quad \begin{array}{l} \text{si } 0 \leqslant i < j \leqslant h \text{ alors } T[i] > T[j] \\ \text{si } i \geqslant h \text{ alors } T[i] = 0 \end{array}$$

(pour $i, j \in \{0, 1, \ldots,$ `lire_nombre_disques_max`$(p) - 1\}$). h est appelé la hauteur du pilier (le nombre de disques qu'il contient).

Cette implantation doit naturellement s'accompagner des fonctions d'écriture et de lecture (implantant les opérations primitives du type abstrait).

q1.3 Implantez et testez les fonctions et procédures suivantes :

- `hauteur` (donnant la hauteur h d'un pilier) ;
- `est_vide` (testant si un pilier est vide) ;
- `est_plein` (testant si un pilier est plein : aucun disque ne pourra y être ajouté) ;
- `disque_au_sommet` (donnant le disque au sommet d'un pilier — `0` si le pilier est vide) ;
- `afficher_pilier` (permettant d'afficher joliment un pilier) ;
- `ajouter_disque_au_sommet` (dont les paramètres sont un entier naturel d représentant un disque et un pilier p et qui a comme effet de modifier p en ajoutant un disque à son sommet) ;
- `copier_pilier` (qui construit et retourne le clone d'un pilier) ;
- `pilier_A_initial` (de paramètre l'entier naturel n et qui construit un pilier constitué des n disques suivants : n, $n-1$, ..., 1).

q1.4 Implantez le type `tours`, qui va représenter un état du jeu et qui est donné par trois champs : `pilier_A`, `pilier_B`, `pilier_C`, tous trois de type `pilier` (avec les opérations de lecture et d'écriture correspondantes).

q1.5 Implantez et testez les fonctions et procédures suivantes :

- `état_initial` (qui à un entier naturel n associe un état initial des tours de Hanoï avec n disques) ;
- `afficher_tours` (permettant d'afficher joliment un état) ;
- `copier_tours` (qui construit et retourne le clone d'un état) ;
- `est_final` (qui teste si un état est final).

Q2 Écrivez une procédure qui prend en paramètre un entier naturel non nul n et lance le jeu à partir de l'état initial à n disques. Le joueur (humain) devra choisir une action sous la forme d'une chaîne de deux caractères. Si cette chaîne vaut `"STOP"` (ou commence par le caractère `'S'`), le jeu s'arrête. Si l'action n'est pas possible, on redemandera à l'utilisateur son action. Sinon l'action sera effectuée et on redemandera une nouvelle action et ainsi de suite, jusqu'à ce que l'utilisateur ait trouvé l'état final (ou qu'il ait arrêté le programme).

Q3 Écrivez une procédure qui prend en paramètre un entier naturel non nul n et fait jouer la machine (elle a bien le droit de s'amuser aussi).

Vous pourrez suivre la suite d'indications suivantes, sachant que l'indication $i+1$ est plus informative que l'indication i et qu'il est plus utile — voire plus amusant ? — de trouver avec le moins d'indications possibles (pliez votre feuille pour ne voir qu'une indication après l'autre en suivant les pointillés).

. . . . . .

Indication 1 : Vous pouvez utiliser, pour ce faire, une fonction récursive (il semblerait qu'une solution itérative soit en l'occurrence plus délicate à mettre en œuvre).

. . . . . .

Indication 2 : L'URL indiqué en note de bas de la page précédente donne un exemple de résolution du problème pour $n \in \{1, 2, 3, 4\}$.

. . . . . .

Indication 3 : Si $n \geqslant 2$, On peut décomposer le problème en 3 étapes. D'abord, déplacer $n-1$ disques de A vers B (appel récursif). Puis, déplacer le disque restant de A vers C. Enfin, déplacer $n-1$ disques de B vers C (appel récursif).

. . . . . .

Indication 4 : Vous pouvez utiliser l'algorithme suivant :

Fonction `déplacer_disques` (e : `tours`, d : `entier_nat`, X : `chaîne de caractères`, Y : `chaîne de caractères`) : `tours`
Variables
  Z : `chaîne de caractères`
Début
  Si $d = 0$ **Alors**
    retourner e
  Finsi
  $Z \leftarrow$ le nom de pilier qui n'est ni X ni Y
  $e \leftarrow$ `déplacer_disques`$(e, d-1, X, Z)$
  déplacer le disque au sommet du pilier X vers le pilier Y (modification de e et affichage)
  $e \leftarrow$ `déplacer_disques`$(e, d-1, Z, Y)$
  retourner e
Fin

Pour résoudre le problème des tours de Hanoï à partir de l'état initial e, il suffit alors de calculer la valeur suivante : `déplacer_disques`$(e, n,$ `"A"`, `"C"`$)$.

---

3 Listes (4 séances)

Ex. 8 : échauffement

Récupérez sur la page du site (en cliquant sur le lien `Des fichiers pour les TP`) les fichiers pour la représentation de listes d'entiers en C.

Soit l'opération de profil `sauf_dernier` : `liste` $\longrightarrow$ `liste` définie par les axiomes suivants :

[1] `sauf_dernier`(`cons`$(x,$ `cons`$(y, L))) =$ `cons`$(x,$ `sauf_dernier`(`cons`$(y, L))$)

[2] `sauf_dernier`(`cons`$(x,$ `l_vide`$)) =$ `l_vide`

[3] `sauf_dernier`(`l_vide`$) =$ `l_vide`

Q1 Donnez une implantation récursive de `sauf_dernier` et testez-là.

Q2 Donnez une implantation itérative de `sauf_dernier` et testez-là.

Ex. 9 : grands entiers

Soit `base` un entier naturel tel que `base` $\geqslant 2$. Par exemple, `base` $= 10$. Tout entier naturel n peut s'écrire :

$$n = c_0 \times \texttt{base}^0 + c_1 \times \texttt{base}^1 + \ldots + c_p \times \texttt{base}^p$$

(où $p+1$ est le nombre de chiffres de n dans cette base) : c_0 est le chiffre en $\texttt{base}^0$, c_1 est le chiffre en $\texttt{base}^1$, etc. Par exemple :

$$8695 = 5 \times 10^0 + 9 \times 10^1 + 6 \times 10^2 + 8 \times 10^3$$

On peut donc représenter un entier naturel par la liste L de ses chiffres, par exemple

$$8695 \qquad \text{sera représenté par} \qquad (5\ 9\ 6\ 8)$$

On notera que cette représentation n'est pas unique : on peut ajouter autant de 0 qu'on veut en fin de liste pour représenter le même entier naturel : 8695 est aussi représenté par $(5\ 9\ 6\ 8\ 0\ 0\ 0)$.

Ainsi, 0 pourra être représenté par n'importe quelle liste ne contenant que des 0, dont la liste vide.

Une telle représentation est utile en particulier pour la description d'entiers naturels de grande taille (ce qui n'est pas possible en C avec `unsigned int`).

On appellera dans cet énoncé « grand entier » toute liste d'entiers représentant un entier naturel.

Q1 Définissez la constante `base` dans votre programme.

Q2 Écrivez et testez une procédure qui affiche un grand entier.

Par exemple, l'exécution de

```
L = cons (5, cons (9, cons (6, cons (8, cons (0, cons (0, l_vide ()))))))) ;
afficher_grand_entier (L) ;
```

provoquera l'affichage suivant :

```
8695
```

Q3 Écrivez et testez une fonction qui donne le successeur d'un grand entier (opération `succ`).
Q4 Écrivez une fonction qui à un entier naturel associe le grand entier correspondant.
Q5 Écrivez une fonction qui à un grand entier associe l'entier naturel correspondant.
Q6 Écrivez et testez une fonction qui calcule la somme de deux grands entiers.

Remarque : Une solution (très) inefficace consisterait à s'appuyer directement sur les axiomes de la fonction `plus` vus en cours. Ici, ce qu'on vous demande est de vous appuyer sur la méthode pour additionner deux entiers telle que vue à l'école primaire :

$8639 + 4790$ est calculé ainsi :

$$\begin{array}{cccccc} & & (1) & (1) & & \\ & & 8 & 6 & 3 & 9 \\ + & & 4 & 7 & 9 & 0 \\ \hline 1 & & 3 & 4 & 2 & 9 \end{array}$$

Q7 Écrivez et testez une fonction qui calcule la différence de deux grands entiers.
Q8 Écrivez et testez une fonction qui calcule le produit de deux grands entiers.
Q9 Écrivez et testez une fonction qui calcule la factorielle d'un grand entier.
Le test devra se faire sur le paramètre $(0\ \ 0\ \ 1)$ (i.e., le grand entier représentant l'entier naturel 100).

Ex. 10 : test de parenthésage

Il a été vu en cours :

— Qu'une pile peut être représentée à l'aide d'une liste (`l_vide`, `cons`, `est_vide`, `prem` et `reste`, respectivement à la place de `pile_vide`, `empiler`, `est_vide`, `sommet` et `dépiler`);
— Qu'on pouvait se servir des piles pour tester qu'une chaîne de caractères était bien parenthésée.

Le but de cet exercice est d'implanter un tel test.

Q1 Implantez et testez le type des listes de caractères (en copiant les fichiers implantant le type des listes d'entiers et en modifiant ce qui doit l'être).
Q2 Écrivez et testez une fonction qui teste qu'un caractère est un signe de parenthésage ouvrant : `'('`, `'['` ou `'{'`.
Q3 Écrivez et testez une fonction qui teste qu'un caractère est un signe de parenthésage fermant : `')'`, `']'` ou `'}'`.
Q4 Écrivez et testez une fonction qui prend deux caractères en entrée et qui teste que le premier est un signe de parenthésage ouvrant, que le second est un signe de parenthésage fermant et que ces deux signes se correspondent : `'('` avec `')'`, etc.
Q5 Écrivez et testez une fonction qui teste qu'une chaîne de caractères est bien parenthésée.
Q6 Parmi les chaînes de caractères mal parenthésées, on distingue :

(a) Celles qui le sont parce qu'il manque des signes de parenthésage fermant en fin de chaîne, par exemple : `"a+(b - [c * d * e / (f + g)]"`.
(b) Les autres.

Écrivez et testez une fonction qui permette de distinguer les chaînes de caractères bien parenthésées (valeur de retour `1`), les chaînes de type (a) (valeur de retour `-1`) et les chaînes de type (b) (valeur de retour `-2`).
Q7 Écrivez et testez une fonction qui, à une chaîne de type (a), associe une chaîne de caractères bien parenthésée obtenue en allongeant la chaîne donnée en argument.