Fiches outils

Table des matières

Générer un nombre aléatoire en C

Pour générer un nombre pseudo-aléatoire en C, on utilise la fonction `rand`. Celle-ci est codée dans la librairie standard, qu'il faut donc inclure. Cette fonction est un générateur, à chaque appel au cours d'une exécution elle donne un nouvel entier, la suite des entiers ainsi obtenus étant une suite de nombres pseudo-aléatoires. Cependant, utilisée telle quelle, cette fonction a le même comportement à chaque exécution. En commentant la ligne 6 du programme ci-contre avant compilation, on peut observer qu'il s'exécute toujours de la même manière.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<time.h>
4
5  int main(){
6    srand(time(NULL));
7    for(int i=0;i<=20;i++){
8      printf("%d\n", rand());
9    }
10   return 0;
11 }
```

Pour pallier ce problème, il faut initialiser cette suite avec une nouvelle valeur à chaque exécution. Pour cela, on utilise généralement la fonction `time` de la librairie `time.h` qui renvoie le nombre de secondes écoulées depuis le 01/01/1970 à 00h00mm00s.

Les entiers générés simulent une loi uniforme sur $[\![0, N]\!]$ avec $N =$ `RAND_MAX`, où `RAND_MAX` est une constante supérieure à $2^{15} - 1$ qu'on peut utiliser (et afficher par curiosité). Pour obtenir des entiers de $[\![0, p-1]\!]$, on pourra prendre les valeurs obtenues modulo p, et pour obtenir des entiers de $[\![a, b]\!]$, on pourra prendre les valeurs obtenues modulo $p = b - a + 1$ puis leur ajouter a, même si ces opérations ne préservent pas tout à fait l'uniformité du tirage.

Exercice Écrire un programme qui affiche la proportion d'entiers plus petits que `RAND_MAX`$/2$ pour 1000 tirages effectués avec la fonction `rand`.

Mesurer le temps en C

Pour mesurer des temps en C, on utilise la librairie `time.h` (on prendra garde à ne pas oublier *##include<time.h>*). Celle-ci fournit notamment les fonctions suivantes :

- la fonction `time` donne le temps en secondes écoulé depuis un instant fixé (le 1er janvier 1970, mais peu importe). Elle prend en argument un pointeur dans lequel enregistrer le résultat. Ce résultat étant aussi retourné, on peut passer `NULL` en argument. Son type de retour est **`time_t`**, ce qui correspond au type **`int`** ou **`long int`**.
- la fonction `clock` donne le temps en **micro**secondes écoulé depuis un instant fixé (le début de l'exécution du programme). Elle ne prend aucun argument et son type de retour est **`time_t`**, ce qui correspond aussi au type **`int`** ou **`long int`**.

Dans les deux cas on obtient une par différence entre les instants de début et de fin.

- Si on utilise la fonction `time`, la précision est de l'ordre de la seconde. Si on souhaite convertir ce nombre entier de secondes en une valeur de type **`double`**, par exemple en vue de calculer un temps moyen par division (réelle !), on utilise la fonction `difftime` qui permet à la fois de faire la différence entre les deux instants de type `time_t` et de la convertir en **`double`**.
- Si on utilise la fonction `clock`, la précision est de l'ordre de la microseconde, on peut afficher le résultat comme un nombre décimal de secondes grâce une division réelle.

Exemple. On propose deux fonctions pour étudier `fibo_bete` (volontairement mal codée). En regard d'un `main` appellant ces deux fonction, on présente l'affichage obtenu.

```c
int fibo_bete (int n){
  if ( n == 0 || n == 1){return 1;}
  else {return fibo_bete (n-1) + fibo_bete (n-2);}
}
```

```c
void fibo_affiche_tps (int n){
  time_t deb = time (NULL);
  fibo_bete(n);
  time_t fin = time (NULL);
  double duree = difftime(fin,deb);
  printf("n=%d : %fs\n",n,duree);
}
```

```c
void fibo_affiche_tps_bis (int n){
  clock_t deb = clock ();
  fibo_bete(n);
  clock_t fin = clock ();
  clock_t duree = fin - deb;
  float duree_s = duree/1000000.0;
  printf("n=%d : %dms",n, duree);
  printf(" soit %fs\n",duree_s);
}
```

```c
  printf("%d ----\n", clock());
  fibo_affiche_tps(36);
  fibo_affiche_tps(38);
  fibo_affiche_tps(40);
  printf("----------------\n");
  printf("%d \n", clock());
  printf("----------------\n");
  fibo_affiche_tps_bis(36);
  fibo_affiche_tps_bis(38);
  fibo_affiche_tps_bis(40);
  printf("----------------\n");
  printf("%d \n", clock());
```

```
2100 ----
n=36 : 1.000000s
n=38 : 2.000000s
n=40 : 6.000000s
----------------
8770421
----------------
n=36 : 840657ms soit 0.840657s
n=38 : 2189553ms soit 2.189553s
n=40 : 5752209ms soit 5.752209s
----------------
17553033
```

Créer un exécutable qui prend des arguments en ligne de commande en C

Il est possible de passer des paramètres à un programme C compilé `prgm` au moment où on lance son exécution en ligne de commande, à condition d'avoir déclaré dans le code source du programme la fonction `main` avec la signature suivante `int main(int argc, char* argv[]);`
Attention : contrairement à ce qu'on peut faire avec les arguments d'une fonction, les paramètres d'un programme sont tous des chaînes de caractères.
Au moment de l'exécution du programme, la fonction `main` est alors appelée avec des valeurs d'arguments qui dépendent de la ligne de commande

- `argc` a pour valeur le nombre de mots constituant la ligne de commande, y compris la commande elle-même, c'est donc le nombre de paramètres réellement souhaités plus 1
- `argv` est un tableau de `argc` chaînes de caractères initialisées selon les mots de la ligne de commande, en particulier `argv[0]` est le nom de la commande (donc rarement utile).

Exemples

• La ligne de commande `./prog a 12`, `argc` vaut 3 (la commande + les 2 paramètres), et `argv[0]` pointe vers la chaîne `"./prog"`, `argv[1]` vers `"a"` et `argv[2]` vers `"12"`. Pour récupérer la valeur d'un entier à partir d'une chaîne de caractères contenant son écriture décimale, on peut utiliser la fonction `atoi` de la librairie standard.

• Si `add` est le fichier obtenu en compilant le code ci-contre, alors `./add 12 24` affiche `12 + 24 = 36`, mais `./add 12 24 36` affiche `on attend 2 entiers` avant l'erreur d'assertion.

Exercice Créer en C un programme `echo` qui affiche la ligne de commande qui l'a lancé, hormis le premier mot `./echo`.

```c
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]){
  int nb_arg_attendu = 2;
  if(argc != nb_arg_attendu +1){
    printf("on attend 2 entiers\n");
    assert(false);
  }
  int a = atoi(argv[1]);
  int b = atoi(argv[2]);

  printf("%d + %d = %d\n", a, b,
  ↪  a+b);

  return 0;
}
```

Input/Output en C

Afin de lire ou écrire dans des fichiers en C, on utilise les fonctions de la librairie `stdio.h`, notamment les fonctions suivantes :

- `void fprintf(FILE* fp, … )` qui se comporte comme `printf` sauf qu'elle prend un argument supplémentaire : un descripteur `FILE* fp` du fichier dans lequel elle écrire ;
- `void fscanf(FILE* fp, … )` qui se comporte comme `scanf`♣ sauf qu'elle prend un argument supplémentaire : un descripteur `FILE* fp` du fichier dans lequel elle doit lire.

Le descripteurs de fichiers (de type `FILE* fp`) sont obtenus grâce à la fonction `fopen`. On ouvre un fichier nommé `file.txt` :

- en mode lecture par l'appel `FILE* fp = fopen("file.txt", "r");` ("r" pour *read*) ;
- en mode écriture par l'appel `FILE* fp = fopen("file.txt", "w");` ("w" pour *write*) ;
- en mode ajout à la fin par l'appel `FILE* fp = fopen("file.txt", "a");`.

Si le fichier `file.txt` n'existe pas encore, l'ouvrir en mode lecture renvoie le pointeur `NULL`, tandis qu'en mode écriture ou ajout il sera crée. Si au contraire il existe déjà, l'ouvrir en mode écriture avec `"w"` écrasera son contenu. Un fichier ouvert en lecture ne peut qu'être lu, un fichier ouvert en écriture ne peut être qu'écrit.
Après usage, on ferme un fichier ouvert de descripteur `fp` par l'appel `close(fp)`.

Par exemple, le programme ci-contre exécuté en présence d'un fichier `a.txt` de la forme :

```
toto
3 4
```

produit un fichier `b.txt` de la forme :

```
(toto)4 3
```

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

int main() {
  FILE* fp1 = fopen("a.txt", "r");
  FILE* fp2 = fopen("b.txt", "w");
  char buffer[100];
  int tmp1;
  int tmp2;
  fscanf(fp1, "%s\n", buffer);
  fprintf(fp2, "(%s)", buffer);
  fscanf(fp1, "%d ", &tmp1);
  fscanf(fp1, "%d", &tmp2);
  fprintf(fp2, "%d ", tmp2);
  fprintf(fp2, "%d", tmp1);
  fclose(fp1);
  fclose(fp2);
  return 0;
}
```

♣ Le comportement de `scanf` avec le spécificateur de format `%s` obéïssant à des règles difficiles à deviner, il est conseillé de faire des tests. À titre d'exemple, si la première ligne du fichier `a.txt` est changée en " `toto` " le résultat du programme donné en exemple est le même, mais avec la ligne " `toto . tutu`" on obtient des 0 au lieu de 3 et 4...

Compilation séparée en C

À retenir

- La compilation se fait en deux étapes.
- La première nécessite que tout soit bien déclaré, et si ce n'est pas la cas on obtient des erreurs de type `implicit declaration of ...`.
- La deuxième nécessite que les fonctions appelées soient bien définies, et doit avoir accès à ces définitions, si ce n'est pas le cas on obtient des erreurs de type `undefined reference to ...`.
- La première étape produit un fichier compilé intermédiaire, qu'on nomme ….`o`.
- La deuxième étape produit un fichier exécutable, qu'on nomme sans extension. Pour cela, le point d'entrée est la fonction `main`, ainsi le code à compiler doit contenir une et une seule fonction `main`.
- un `Makefile` premet de définir des raccourcis pour lancer les compilations utiles.

En pratique (sans `Makefile`)

→ on déclare les fonctions dans un fichier `xxx.h` en donnant leur signature, par exemple `int nb_zeros(int* tab, int lg);`
→ on définit toutes les fonctions déclarées dans `xxx.h` dans un fichier `xxx.c` qui inclut `xxx.h` grâce à `#include "xxx.h"`
→ on compile ces définitions via la commande `gcc -c xxx.c -o xxx.o`.
→ on teste toutes ces fonctions dans la fonction `main` d'un fichier `test_xxx.c` qui inclut lui aussi `xxx.h` grâce à `#include "xxx.h"`
→ on compile ce programme de test avec les définitions de fonctions déjà compilées via la commande `gcc xxx.o test_xxx.c -o xxx`
→ on lance le programme de test avec `./xxx`

Pour éviter d'inclure plusieurs fois

Supposons qu'on ait déclaré dans `a.h` et codé dans `a.c` un objet A, en vue de travailler sur des objets B et D qui dépendent de A. On déclare l'objet B(resp. D) dans un fichier `b.h` (resp. `d.h`), qui contient `#include "a.h"` et on code les fonctions associées dans `b.c` (resp. `d.c`). On teste ces trois objets dans un même programme, codé dans le `main` du fichier `test.c`. qui contient `#include "b.h"` et `#include "d.h"`. Nul besoin d'importer `a.h`, car les déclarations de `a.h` sont incluses à travers `b.h`. Le problème c'est qu'elles sont incluses une deuxième fois à travers `c.h`, ce qui déclenche une erreur de redéfinition... Pour éviter ce problème, on conditionne la lecture du fichier au fait qu'une "variable" soit définie :

- à la première lecture du fichier `xxx.h`, la variable `XXX_H` n'est pas encore définie, alors on la définit puis le fichier est lu, donc les déclarations qu'il contient sont incluses ;
- à la seconde lecture du fichier `xxx.h`, la variable `XXX_H` est déjà définie, le fichier n'est pas relu et les déclarations qu'il contient ne sont pas re-incluses.

`a.h`

```
#ifndef A_H
#define A_h
...
#endif
```

`a.c`

```
#include "a.h"
...
```

`b.h` (idem pour `d.h`)

```
#ifndef B_H
#define B_h
..
#include "a.h"
...
#endif
```

`b.c` (idem pour `d.c`)

```
#include "b.h"
...
```

`main.c`

```
#include "b.h"
#include "d.h"
...
```

Détecter les fuites mémoires en C avec Valgrind

Valgrind est un petit utilitaire qu'on peut lancer depuis un terminal pour analyser un exécutable C, notamment pour analyser quel usage de la mémoire fait le programme, et détecter d'éventuelles fuites mémoire. Après avoir compilé le programme avec gcc avec l'option -g, on peut lancer une exécution en demandant à Valgrind d'analyser ce qui se passe.

```
gcc -g mon_programme.c -o mon_programme
valgrind -v --leak-check=summary ./mon_programme
```

Cette opération produit beaucoup d'affichages[a], mais la partie `HEAP SUMMARY` est particulièrement intéressante puisqu'elle résume l'utilisation du tas. Ainsi, sur un programme sans fuite mémoire on obtient par exemple :

```
==8112== HEAP SUMMARY:
==8112==     in use at exit: 0 bytes in 0 blocks
==8112==   total heap usage: 11 allocs, 11 frees, 1,164 bytes allocated
==8112==
==8112== All heap blocks were freed -- no leaks are possible
```

tandis que sur un programme où des zones mémoires n'ont pas été libérées on obtient par exemple :

```
==8175== HEAP SUMMARY:
==8175==     in use at exit: 140 bytes in 10 blocks
==8175==   total heap usage: 11 allocs, 1 frees, 1,164 bytes allocated
==8175==
==8175== Searching for pointers to 10 not-freed blocks
==8175== Checked 74,744 bytes
==8175==
==8175== LEAK SUMMARY:
==8175==    definitely lost: 32 bytes in 2 blocks
==8175==    indirectly lost: 108 bytes in 8 blocks
==8175==      possibly lost: 0 bytes in 0 blocks
==8175==    still reachable: 0 bytes in 0 blocks
==8175==         suppressed: 0 bytes in 0 blocks
```

NB : Pour l'installation, vous pouvez vous référez à la page suivante. `https://doc.ubuntu-fr.org/valgrind#installation`.

a. il est sûrement possible de réduire ces affichages avec des options bien choisies

Obtenir le code assembleur d'un programme C

Pour obtenir le code assembleur d'un programme C `hello_world.c`, on le compile avec l'option en `-g`, puis on applique la commande `objdump` à l'exécutable obtenu, avec l'option `-d` pour indiquer que l'on souhaite voir le code correspondant aux sections exécutables, et avec l'option `-S` pour indiquer que l'on souhaite que le code source correspondant soit indiqué.

```
gcc -g hello_world.c -o hello_world
objdump -S -d hello_world
```

Par exemple pour `hello_world.c` dont le `main` est réduit à l'affichage de "`hello world`", on obtient, entre autres lignes qu'on ignore ici, l'affichage suivant

```
0000000000001135 <main>:
#include<stdio.h>
int main(){
    1135:  55                    push   %rbp
    1136:  48 89 e5              mov    %rsp,%rbp
  printf("hello world \n");
    1139:  48 8d 3d c4 0e 00 00  lea    0xec4(%rip),%rdi        # 2004
    ↪  <_IO_stdin_used+0x4>
    1140:  e8 eb fe ff ff        callq  1030 <puts@plt>
  return 0;
    1145:  b8 00 00 00 00        mov    $0x0,%eax
}
    114a:  5d                    pop    %rbp
    114b:  c3                    retq
    114c:  0f 1f 40 00           nopl   0x0(%rax)
```

> Si l'utilitaire `objdump` n'est pas installé sur votre machine, il faudra l'installer à l'aide d'une commande de la forme suivante (exécutée en mode administrateur, donc après s'être authentifié comme `root` grâce à `su` ou en la faisant précéder de `sudo`).
> `apt-get install binutils-x86-64-linux-gnu` Selon l'architecture de votre machine, le nom du paquet à installer peut différer : par exemple `binutils-i586-linux-gnu`. On peut obtenir l'architecture utilisée sur la machine grâce à la commande `arch` lancée dans le terminal.

Lancer deux fils d'exécution en parallèle en C

Pour gérer des fils d'exécution en C on utilise la bibliothèque `pthread`. Ainsi, on veillera à indiquer `include <pthread.h>` en tête du fichier `main.c`, et à le compiler avec l'option `-pthread` :

```
gcc -pthread main.c -o main
```

Les fils d'exécution sont des objets de type `pthread_t`. Ils doivent d'abord être déclarés puis sont lancés par la fonction `pthread_create` qui prend 4 arguments :

- l'adresse du fil d'exécution, de type `pthread_t*` donc ;
- une adresse qui ne nous est pas utile, on mettra donc `NULL` ;
- une fonction `void* todo(void* arg);`, qui est celle que le fil d'exécution doit exécuter ;
- un pointeur vers les arguments de type `void*`.

Il est donc nécessaire de formater les instructions à effectuer dans une fonction `void* todo(void* arg)`. Pour cela on crée d'abord une structure `struct arg_s` qui rassemble les données utiles à `todo` dans un même objet `a` dont l'adresse sera passée au fil d'exécution comme 4-ème argument de `pthread_create`. La fonction `todo` commence par transtyper son argument `arg` pour y reconnaître un pointeur de type `(struct arg_s)*`. Les calculs peuvent donc être codés en accédant aux données par `a->champ`. Enfin le résultat ne peut être retourné en sortie de `todo`, mais doit être enregistré dans un objet existant hors de la fonction (on ajoute parfois son adresse dans la structure `arg_s`). Autrement dit la fonction `todo` doit non seulement prendre ses entrées à travers un unique pointeur, mais doit aussi agir par effet de bord uniquement.

On attend que plusieurs fils d'exécution aient terminé leur exécution grâce à la fonction `pthread_join` qui prend 2 arguments :

- le processus, de type `pthread_t` donc ;
- une adresse qui ne nous est pas utile, on mettra donc `NULL`.

```c
#include <pthread.h>
#include <assert.h>

struct args_s {
  int nb;
  int* res;
};
typedef struct args_s arg_carre;

void* au_carre(void* args) {
  arg_carre* a = (arg_carre*) args;
  *(a->res) = a->nb * a->nb;
  return NULL;
}

int main(){
  int resA, resB;
  arg_carre argsA = {2, &resA};
  arg_carre argsB = {9, &resB};
  pthread_t pA, pB;
  pthread_create(&pA, NULL, au_carre, &argsA);
  pthread_create(&pB, NULL, au_carre, &argsB);
  pthread_join(pA, NULL);
  pthread_join(pB, NULL);
  assert((resA == 4) && (resB == 81));
  return 0;
}
```

Lancer une kyrielle de fils d'exécution en parallèle en C

Afin de lancer un grand nombre ♣de fils d'exécution en parallèle, il faut les déclarer non pas explicitement un à un, mais dans un tableau à l'aide d'un `malloc`. De même les structures qui servent à passer les arguments aux fils d'exécution doivent être stockées dans des tableaux. Il doit y avoir autant de structures différentes physiquement (placées à des endroits différents en mémoire) que de fils d'exécution. En particulier déclarer les arguments dans une structure qu'on modifie à chaque tour de boucle ne marche pas : l'espace alloué pour les arguments du premier fil est un espace dans la pile qui est modifié dès le deuxième tour de boucle, et potentiellement avant que le premier fil n'ait pu lire ses arguments, et/ou écrire son résultat. On donne ci-dessous un exemple type de lancement d'un grand nombre de fils d'exécution.

```c
#include <pthread.h>
#include <stdlib.h>
#include <assert.h>

struct arg_s {
  int i;      //indice de la case
  int* t_in;  //tableau des entrées
  int* t_out; //tableau des résulats
};
typedef struct arg_s arg;

void* moins_un(void* ptr_args) {
  arg* a = (arg*) ptr_args;
  a->t_out[a->i] = a->t_in[a->i] - 1;
  return NULL;
}

int main(){ //remplit res tq pour i\in[1..n], res[i] = val[i]-1
  int n = 10;
  int* val = (int*) malloc(n*sizeof(int));
  int* res = (int*) malloc(n*sizeof(int));

  //1-préparer les arguments
  arg* args = (arg*) malloc(n*sizeof(struct arg_s));
  for(int k = 0; k < n; k++){
    args[k] = (arg) {k,val,res};
  }
  //2-lancer les threads
  pthread_t* threads = (pthread_t*) malloc(n*sizeof(pthread_t));
  for(int k = 0; k < n; k++){
    pthread_create(threads+k, NULL, moins_un, args+k);
  }
  //3-attendre la fin des threads
  for(int k = 0; k < n; k++){
    pthread_join(threads[k], NULL);
  }
  //4-lire les résultats
  assert((res[0] == val[0]-1) && (res[n-1] == val[n-1]-1));
  //5-libérer la mémoire
  free (val);
  free (res);
  free (args);
  free (threads);
  return 0;
}
```

Cette notice fait suite à la notice "Lancer **deux** fils d'exécution en parallèle en C ".

Utilisation de Verrou en C

Le type abstrait Verrou est implémenté en C par le type `pthread_mutex_t` de la bibliothèque `pthread`. Ainsi pour l'utiliser on veillera à indiquer `#include <pthread.h>` en tête du fichier `main.c`, et à le compiler avec l'option `-pthread` : `gcc -pthread main.c -o main`.

Une fois un verrou `v` déclaré, on garantit l'exclusion mutuelle entre les sections de codes délimitées par un appel à `pthread_mutex_lock(&v);` et un appel à `pthread_mutex_unlock(&v)`. Cependant, le verrou `v` doit être initialisé dans la fonction `main` par un appel à `pthread_mutex_init(&v, NULL)` avant les appels à `pthread_mutex_lock` et `pthread_mutex_unlock`. Ainsi pour utiliser des verrous en C on réalise les étapes suivantes.

- Déclarer autant de verrous que nécessaire.
- Définir des structures pour passer aux tâches leurs arguments.
- Définir les tâches à faire, en protégeant leur sections critiques avec les verrous.
- Dans la fonction `main` :
  - initialiser chaque verrou par un appel à `pthread_mutex_init(&v, NULL);` ;
  - initialiser les variables qui seront partagées par les fils ;
  - initialiser autant de structures que besoin ;
  - déclarer puis lancer les fils en attribuant à chacun sa tâche et ses arguments ;
  - attendre la fin de tous les fils pour récupérer le résultat.

Exemple 1. On donne ci-dessous une exemple avec trois fils d'exécution devant réaliser la même tâche, à savoir incrémenter un compteur partagé un nombre donné de fois. On définit donc une seule tâche (la fonction `add_one`), et une seule structure pour passer à cette fonction ses arguments (la structure `args`). Afin de mettre en exclusion mutuelle les incrémentations du compteur faites par chaque fil réalisant `add_one`, on protège la ligne 19 grâce à un verrou déclaré au préalable (ligne 5).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  pthread_mutex_t verrou;
6
7  struct args_s {
8   int* cpt; // adr. du compteur partagé
9   int nbt;  // nb de tours souhaités
10 };
11 typedef struct args_s args;
12
13 void* add_one(void* arg) {
14  // hyp : arg est de type args*
15  // incr. arg->nbt fois *(arg->cpt)
16  args* a = (args*) arg;
17  for (int i = 0; i < a->nbt ; i++) {
18   pthread_mutex_lock(&verrou);
19   *(a->cpt)= *(a->cpt) + 1;
20   pthread_mutex_unlock(&verrou);
21  }
22  return NULL;
23 }
```

```
30 int main(int argc, char* argv[]) {
31
32  int cpt = 0; // variable partagée
33  args a1 = {.cpt = &cpt, .nbt = 10000};
34
35  pthread_mutex_init(&verrou, NULL);
36
37  pthread_t p1, p2, p3;
38  pthread_create(&p1, NULL, add_one, &a1);
39  pthread_create(&p2, NULL, add_one, &a1);
40  pthread_create(&p3, NULL, add_one, &a1);
41
42  pthread_join(p1, NULL);
43  pthread_join(p2, NULL);
44  pthread_join(p3, NULL);
45
46  printf(" [cpt: %d]\n [obj:
    ↪  %d]\n", *(a1.cpt), 3*a1.nbt);
47
48  return 0;
49 }
```

Générer un nombre aléatoire en OCaml

Pour générer un nombre pseudo-aléatoire en OCaml, on utilise le module `Random`. Ce module fournit la fonction `Random.int` de type `int -> int` telle que pour `n` un entier entre 0 et 2^{30}, l'appel `Random.int n` calcule un entier de l'intervalle $[\![0, n[\![$, tiré uniformément sur cet intervalle.

Mesurer le temps en OCaml

La fonction `time` du module `Sys` donne le temps en secondes écoulé lors de l'exécution du programme.

```
Sys.time : unit -> float
```

On obtient alors un temps de calcul par différence entre l'instant de début et l'instant de fin.

```
let rec fibo_bete (n : int) : int =
  match n with
  | 0 -> 0
  | 1 -> 1
  | _ -> (fibo_bete (n-1)) + (fibo_bete (n-2))

let fibo_affiche_tps (n : int) : unit =
  let debut = Sys.time () in
  let _     = fibo_bete n in
  let fin   = Sys.time () in
  print_string (string_of_float (fin-.debut))
```

Dans l'exemple ci-dessus on a défini une fonction qu'on sait être longue à éxecuter sur des grandes valeurs. On a ensuite encapsulé un appel à cette fonction dans une deuxième fonction qui prend soin de noter l'instant avant l'appel, l'instant après l'appel, et qui affiche ensuite la différence entre les deux (on note l'usage de `-.` et non de `-`) soit la durée d'exécution de cet appel.

Créer un exécutable qui prend des arguments en ligne de commande en OCaml

Il est possible de passer des paramètres à un programme OCaml compilé `prgm` au moment où on lance son exécution en ligne de commande. On rappelle que la commande ci-dessous compile le fichier `code.ml` en un exécutable `prgm`.

```
ocamlc code.ml -o prgm
```

Contrairement à ce qui est fait en C, le point d'entrée du programme n'est pas une fonction particulière. En effet l'exécution de `prgm` a pour effet l'évaluation de toutes les valeurs définies dans `code.ml`, en particulier celles de type **`unit`** dont on peut observer les effets de bord. Ainsi, de même qu'on peut définir un jeu de tests par **`let`** `test_f:`**`unit`** `=...`, on peut définir la suite d'instructions principales par **`let`** `main:`**`unit`** `=...`.
On accède aux arguments passés au programme à travers le tableau `Sys.argv`.
Comme en C, il s'agit d'un tableau de chaînes de caractères (*i.e.* de type **`string array`**) initialisé selon les mots de la ligne de commande. En particulier `Sys.argv(`**`0`**`)` est le nom de la commande (rarement utile), et `Array.length Sys.argv` est le nombre de paramètres +1.

Deux exemples

```
let main : unit =
  print_string ("ici on a lancé la commande \" "^Sys.argv.(0)^"\" \n");
  let n = Array.length Sys.argv in
  for i=1 to n-1 do
    print_string ("le paramètre "^string_of_int i^" est "^Sys.argv.(i)^"\n")
  done;
```

Si `affiche_param` est le programme obtenu en compilant le code ci-dessus, `./affiche_param 12 bjr` produit l'affichage suivant.

```
ici on a lancé la commande " ./affiche_param"
le paramètre 1 est 12
le paramètre 2 est bjr
```

```
let main : unit =
  let n = Array.length Sys.argv in
  let s = ref 0 in
  for i=1 to n-1 do
    s := !s + int_of_string Sys.argv.(i)
  done;
  print_string("la somme des paramètres vaut "^string_of_int !s^"\n")
```

Si `add` est le programme obtenu en compilant le code ci-dessus, alors

- `./add` affiche `la somme des paramètres vaut 0`
- `./add 1 2 6` affiche `la somme des paramètres vaut 9`
- `./add 1 2 salut` affiche l'erreur `Fatal error: exception Failure("int_of_string")`

NB : lors de l'import avec `#use` sous `utop`, il n'est pas possible de passer des paramètres, seul `Sys.argv.(0)` est rempli avec le chemin jusqu'à l'exécutable `utop`.

Input/Output en OCaml

Afin de lire/d'écrire dans des fichiers en OCaml, on utilise les deux fonctions suivantes.

- `output_string : out_channel -> string -> unit` qui se comporte comme `print_string` sauf qu'elle prend en premier argument un descripteur du fichier dans lequel elle doit écrire (de type `out_channel`).
- `input_line : in_channel -> string` qui prend en argument un descripteur du fichier dans lequel elle doit lire (de type `in_channel`) et retournant la prochaine ligne du fichier. Dans le cas où l'on est arrivé en fin de fichier, cette fonction lève l'exception `End_of_file`.

Descripteurs de fichiers. Le descripteur d'un fichier ouvert en lecture est de type `in_channel`, tandis que celui d'un fichier ouvert en lecture est de type `out_channel`. On obtient ces descripteurs à l'aide des deux fonctions suivantes.

- `open_in : string -> in_channel` qui prend en argument le nom d'un fichier existant et qui renvoie un descripteur vers ce fichier en lecture (si le fichier n'existe pas une exception `Sys_error` est levée).
- `open_out : string -> out_channel` rend en argument le nom d'un fichier et renvoie un descripteur vers ce fichier en écriture. Si le nom donné n'est pas celui d'un fichier existant, un fichier de ce nom est créé. Si le nom donné est celui d'un fichier existant, le contenu dudit fichier est écrasé dès l'ouverture.

Après usage, on ferme les fichiers ouverts à l'aide des deux fonctions suivantes.

- `close_in : in_channel -> unit` pour les fichiers ouverts en lecture.
- `close_out : out_channel -> unit` pour les fichiers ouverts en écriture.

Exemple Le programme OCaml ci-dessous, lorsqu'il est exécuté en présence d'un fichier `a.txt` de la forme :

```
toto
3 4
```

produit un fichier `b.txt` de la forme :

```
(toto)3 4
```

```
let () =
  let fp1 = open_in "a.txt" in
  let fp2 = open_out "b.txt" in
  let ligne1 = input_line fp1 in
  output_string fp2 ("(" ^ ligne1 ^ ")");
  let ligne2 = input_line fp1 in
  let tmp1 = int_of_string (String.sub ligne2 0 1) in
  let tmp2 = int_of_string (String.sub ligne2 2 1) in
  output_string fp2 (string_of_int tmp1);
  output_string fp2 " ";
  output_string fp2 (string_of_int tmp2);
  close_in fp1;
  close_out fp2
```

Lancer deux fils d'exécution en parallèle en OCaml

Pour gérer des fils d'exécution en OCaml on utilise le module `Thread` de la librairie `Thread.Posix`. On peut charger ce module sur `utop` en tapant `#require "threads.posix";;`. Afin de compiler un programme OCaml utilisant ce module on pourra utiliser la ligne de commande ci-dessous.

```
ocamlc -thread unix.cma threads.cma main.ml -o main
```

Les fils d'exécution sont des objets de type `Thread.t`. Ils doivent être créés par appel à la fonction `Thread.create : ('a -> 'b) -> 'a -> Thread.t` qui prend en arguments :

- une fonction qui est celle que le fil d'exécution doit exécuter ;
- l'argument sur lequel ce fil d'exécution doit exécuter la fonction.

La fonction `Thread.create` retourne alors le fil d'exécution ainsi créé.

De même que pour la création de fils d'exécution en C, on peut se munir d'une structure permettant la gestion des entrées/sorties de telles fonctions.
On peut demander à attendre qu'un fil d'exécution ait terminé son exécution grâce à la fonction `Thread.join : Thread.t -> unit` qui prend en argument le fil d'exécution en question.

```
type args =
  {
    nb: int           ;
    mutable res : int ;
  }

let au_carre (args: args): unit =
  args.res <- args.nb * args.nb
```

```
let () =
  let arg1 = {nb = 2; res = -1} in
  let arg2 = {nb = 9; res = -1} in
  let pa = Thread.create au_carre arg1 in
  let pb = Thread.create au_carre arg2 in
  Thread.join pa;
  Thread.join pb;
  assert (arg1.res = 4 && arg2.res = 81)
```

Utilisation de Verrou en OCaml

Le type abstrait Verrou est implémenté en OCaml par le module `Mutex` de la librairie `Thread.Posix`. Pour charger ce module sur `utop` on tape donc `#require "threads.posix";;`. Afin de compiler un programme OCaml utilisant ce module on pourra utiliser la ligne de commande ci-dessous.

```
ocamlc -thread unix.cma threads.cma main.ml -o main
```

Les verrous sont des objets de type `Mutex.t`. On les manipule à travers les trois opérations suivantes :

- la fonction `Mutex.create : unit -> Mutex.t` qui crée un verrou ;
- la fonction `Mutex.lock : Mutex.t -> unit` qui permet de verrouiller un verrou ;
- la fonction `Mutex.unlock : Mutex.t -> unit` qui permet de déverrouiller un verrou.

Une fois un verrou `v` créé, on garantit l'exclusion mutuelle entre les sections de codes délimitées par un appel à `Mutex.lock v` et un appel à `Mutex.unlock v`.

Exemple 2. On donne ci-dessous deux façons de coder la cas de deux fils d'exécution devant réaliser la même tâche, à savoir incrémenter un compteur partagé un nombre donné de fois.
Si le nombre d'incrémentation souhaité est le même pour les deux fils, ceux-ci peuvent partager la même structure pour leurs entrées/sorties, qui contient le nombre d'incrémentations et un champ mutable pour le compteur partagé. On aboutit alors au code de gauche.
Dans le cas contraire, les deux fils ont des entrées différentes, et donc nécessairement des structures différentes pour leurs entrées/sorties. Dans ce cas on ne peut pas utiliser le champ mutable comme compteur commun, car ce seront alors deux compteurs différents… Ainsi on passe plutôt aux deux structures la même référence de type entier. On aboutit alors au code de droite.

```
type args =
  {
    nbt : int; (* nombre de tours, >= 0 *)
    mutable cpt : int;
  }

let verrou = Mutex.create ()

(** Ajoute [a.nbt] fois 1 à [a.cpt] *)
let add_one (a : args) : unit =
  for i = 1 to a.nbt do
    Mutex.lock verrou;
    a.cpt <- a.cpt + 1;
    Mutex.unlock verrou
  done

let main (n : int) : unit =
  let a = {nbt = n; cpt = 0} in
  let f1 = Thread.create add_one a in
  let f2 = Thread.create add_one a in
  Thread.join f1;
  Thread.join f2;
  let res = a.cpt in
```

```
type args =
  {
    nbt : int; (* nombre de tours, >= 0 *)
    cpt_ref : int ref;
  }

let verrou = Mutex.create ()

(** Ajoute [a.nbt] fois 1 à [a.cpt] *)
let add_one (a : args) : unit =
  for i = 1 to a.nbt do
    Mutex.lock verrou;
    a.cpt_ref :=  !(a.cpt_ref) + 1;
    Mutex.unlock verrou
  done

let main (n : int) : unit =
  let cpt = ref 0 (* compteur partagé *)
  ↪  in
  let a1 = {nbt = n; cpt_ref = cpt} in
  let a2 = {nbt = 2 * n; cpt_ref = cpt} in
  let f1 = Thread.create add_one a1 in
  let f2 = Thread.create add_one a2 in
  Thread.join f1;
  Thread.join f2;
  let res = !cpt in
```

Créer un module en OCaml

→ Un module permet de regrouper plusieurs définitions de valeur, de type, de fonction ou d'exception se rapportant à un même objet. Par exemple `List` et `String` sont des modules.

→ Le nom d'un module commence toujours par une majuscule.

→ Un module est défini selon la syntaxe suivante.

```
module Nom_module =
  struct
   définition de type
   définition de fonctions
   définition d'exceptions
  end
```

→ En dehors du corps de la définition du module `Mod`, on accède à un élément (une valeur, un type simple, une fonction ou une exception) `obj` de ce module par `Mod.obj`. De plus on accède à un type paramétré `'a t` défini dans `Mod` par `'a Mod.t` à l'extérieur.

On donne ci-après un exemple de définition module pour les fractions (en supposant que la fonction `pgcd` est déjà définie), puis quelques exemples d'utilisation dans un interpréteur.

```
module Fraction =
  struct
    type t = int*int
    let un_tiers : t = (1,3)
    exception FractionMalDefinie

    let simplifie ((num,den):t) : t =
      if den = 0 then raise FractionMalDefinie
      else let d = (pgcd num den) in (num/d, den/d)
  end
```

```
# let f1:Fraction.t = 2,6;;
val f1 : Fraction.t = (2, 6)
# Fraction.simplifie f1;;
- : Fraction.t = (1, 3)
# Fraction.simplifie Fraction.un_tiers;;
- : Fraction.t = (1, 3)
# let f0 = 1,0 in Fraction.simplifie f0;;
Exception: Fraction.FractionMalDefinie.
```

Remarque : Bien qu'on l'évite pour améliorer la lisibilité et éviter des erreurs, il est possible d'utiliser la commande `open Nom_module` pour éviter d'avoir à préfixer tous les éléments du module par le nom du module. De plus, il est aussi possible de factoriser le préfixage par un nom de module, ce qui revient à réaliser une ouverture locale du module. Exemple : `fun l -> List.(length (rev l));;`

Créer une table de hachage en OCaml

Le module `Hashtbl` d'OCaml implémente le type abstrait Dictionnaire qui permet la gestion d'ensembles dynamiques d'associations clés-valeurs, dont les clés sont 2 à 2 distinctes.

Les `Hashtbl` permettent ainsi de représenter :

→ des ensembles dynamiques (les éléments sont les clés, on peut leur associer une valeur booléenne indiquant la présence ou non, on peut aussi imaginer associer une valeur (quelconque) aux clés qui sont dans l'ensemble et ne pas en associer à celles qui ne sont pas dans l'ensemble) ;

→ des multi-ensembles (en remplaçant la présence booléenne par un nombre d'occurrences entier) ;

→ ou plus généralement une fonction sur un ensemble fini.

Une table dont les clés sont de type `'a` et dont les valeurs sont de type `'b` est de type `('a, 'b) Hashtbl.t`.

On crée une table de hachage comme suit, en précisant `n` une estimation du nombre de clés que contiendra la table. Cette valeur est donnée seulement en vue d'améliorer les performances de la table, elle ne limite pas le nombre de clés.

```
let tbl = Hashtbl.create n
```

Pour manipuler la table, on dispose des fonctions élémentaires suivantes.

- `Hashtbl.mem : ('a, 'b) Hashtbl.t -> 'a -> bool` qui teste la présence d'une clé dans la table.
- `Hashtbl.find : ('a, 'b) Hashtbl.t -> 'a -> 'b` qui donne la valeur associée à une clé présente dans la table. Si la clé n'est pas présente dans la table, l'exception `Not_found` est levée.
- `Hashtbl.add : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` qui permet d'ajouter une association clé-valeur à la table. Si la clé était déjà présente, la valeur associée est masquée par la nouvelle valeur.
- `Hashtbl.replace : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` agit comme `add` sauf que si la clé était déjà présente, la valeur qui lui était associée est écrasée.

De plus on dispose aussi d'une fonctionnelle `fold` pour itérer sur les associations de la table.

```
Hashtbl.fold : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) Hashtbl.t -> 'c -> 'c
```

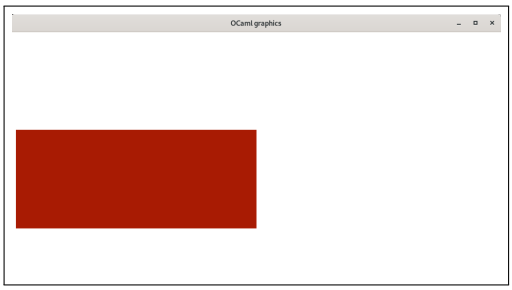
On remarque que l'ordre des arguments dans la fonction d'accumulation diffère de celui pour la fonctionelle `List.fold_left`, en effet ici l'accumulateur est passé en troisième argument.

Utilisation du module Graphics de OCaml

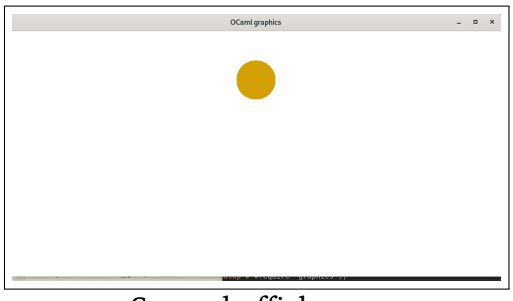
Le module Graphics de OCaml permet de rapidement produire un affichage dans une fenêtre graphique. La fonctionnalité de base du module Graphics est de fournir un **cannevas** sur lequel on peut dessiner. Le présent descriptif n'a pas vocation à remplacer la documentation : https://ocaml.github.io/graphics/graphics/Graphics/index.html, on fournit ici seulement les quelques instructions qui permettent de démarrer, par l'exemple.

```
let () =
  (* Ouvre une fenêtre de 1000*500 pixels (attention espace néc.)*)
  Graphics.open_graph " 1000x500" ;
  (* Change la couleur courante                                  *)
  Graphics.set_color (Graphics.rgb 168 27 3) ;
  (* Dessine un rectangle                                        *)
  Graphics.fill_rect 10 100 490 200 ;
  (* Attend qu'une touche du clavier soit appuyée                *)
  let _ = Graphics.wait_next_event [Key_pressed] in
  (* On supprime le contenu de la fenêtre graphique              *)
  Graphics.clear_graph ();
  (* Change la couleur courante                                  *)
  Graphics.set_color (Graphics.rgb 210 160 4) ;
  (* Dessine un rectangle                                        *)
  Graphics.fill_circle 500 400 40;
  (* Attend qu'une touche du clavier soit appuyée                *)
  let _ = Graphics.wait_next_event [Key_pressed] in
  (* On ferme la fenêtre graphique                               *)
  Graphics.close_graph ()
```

À l'exécution de ce programme, une fenêtre graphique avec un rectangle rouge s'ouvre. Après appui sur une lettre du clavier, l'image est effacée et s'affiche alors un disque jaune. Après un nouvel appui sur une lettre, la fenêtre est fermée (avec utop éviter d'utiliser la croix pour fermer). Si on commente la .11, la 2ème image présentera à la fois le rectangle et le disque.



Premier affichage



Second affichage

Attention : Pour utiliser Graphics avec utop on évaluera `#use "topfind";;` puis `#require "graphics";;` afin de charger le module, et ce avant de charger le fichier qui y fait appel. Ce chargement est à renouveler à chaque fois (comme les `#use`).

Pour compiler un fichier code.ml utilisant Graphics en un exécutable code, taper :
`ocamlfind ocamlopt -o code -linkpkg -package graphics code.ml`

Le type `'a option` en OCaml

Il n'est pas rare que l'on souhaite étendre un type OCaml `t` pour y ajouter une valeur. Considérons par exemple l'ensemble des entiers `int` et la fonction de division :

```
let ma_division (x: int) (y: int) : int =
  print_string "je fais une division";
  (x / y)
```

Cette fonction est bien définie mais lève une erreur quand on l'appelle avec une valeur nulle pour l'entier `y`. Aussi on souhaite ajouter une valeur spéciale, qui ne soit pas un entier, aux entiers, on pourra alors retourner cette valeur dans le cas d'une division par zéro.
OCaml fournit pour cela le type prédéfini `'a option`, défini de la manière suivante :

```
type 'a option =
  | None
  | Some of 'a
```

Aussi un élément de type `'a option` est : ou bien la valeur `None`, ou bien la valeur `Some(x)` où `x` est de type `'a`. On peut alors redéfinir notre division de la manière suivante :

```
let ma_division (x: int) (y: int) : int option =
  print_string "je fais une division";
  if y = 0 then None
  else Some(x / y)
```

Noter le type de retour de la fonction.

```
# ma_division 4 2 ;;
je fais une division- : int option = Some 2
# ma_division 4 0 ;;
je fais une division- : int option = None
```

Bien sûr, en tant que type défini par énumération, on peut raisonner par disjonction sur une valeur de type `'a option` au moyen de la construction `match ... with ...`. Ainsi on peut définir la fonction ci-dessous effectuant une division par 2 au moyen de la fonction `ma_division`.

```
let division_par_2 (x: int) : int =
  match ma_division x 2 with
  | None -> failwith "ma_division : division par zéro"
  | Some(r) -> r
```

Noter le type de retour de la fonction.

```
# division_par_2 3 ;;
je fais une division- : int = 1
```

Une utilisation classique du type `'a option` est en substitution au type `bool`. Considérons par exemple une fonction de recherche : **existe-t-il un élément dans le tableau vérifiant telle propriété ?** Une telle fonction aurait un type de retour booléen : oui ou non. On pourrait toutefois enrichir une telle fonction en : **existe-t-il un élément dans le tableau vérifiant telle propriété ? Si oui le retourner.** Une telle fonction aurait un type de retour optionnel : non (`None`) ou oui et voici l'élément (`Some( élément)`).

Opérateurs binaires à notation infixe en OCaml

En OCaml, les opérateurs arithmétiques usuels comme `+`,`*`,`/`,`mod` ...ou `+.`,`*.`,`/.` ..., les opérateurs booléens `&&` et `||` ou encore les opérateurs de concaténation[a] `@` et `^` sont en fait des fonctions de deux arguments qui ont la particularité de pouvoir être appelées selon la notation infixe, c'est-à-dire en étant placé entre leur deux arguments et non à gauche de ceux-ci. Afin de désigner de telles fonctions en OCaml, on encadre l'opérateur de parenthèses. Par exemple `(+)` est une expression de type fonctionnel `int -> int -> int`, alors que l'expression `+` n'est pas syntaxiquement correcte. De même :

- `(mod)` est de type `int -> int -> int`;
- `(+.)` est de type `float -> float -> float`;
- `(&&)` est de type `bool -> bool -> bool`;
- `(^)` est de type `string -> string -> string`;
- `( |> )` est de type `'a -> ('a -> 'b) -> 'b`;
- ...[b]

De tels opérateurs ne sont pas réservés à la librairie standard, on peut définir de tels opérateurs à condition de choisir un nom de fonction valide. La description précise des noms valides est donnée dans la documentation : `https://v2.ocaml.org/manual/expr.html`, rubrique *Operators*. Sans être exhaustif, avec $\Sigma_1 = \{\texttt{\$}, \texttt{\&}, \texttt{*}, \texttt{+}, \texttt{-}, \texttt{/}, \texttt{=}, \texttt{>}, \texttt{\^{}}, \texttt{|}, \texttt{\%}, \texttt{<}\}$ et $\Sigma_2 = \Sigma_1 \cup \{\texttt{\~{}}, \texttt{!}, \texttt{?}, \texttt{:}, \texttt{.}\}$, les noms dans $\Sigma_1 \cdot \Sigma_2^*$ sont valides.

Exemple. On pourrait par exemple définir l'opérateur `%%` pour le produit scalaire sur les vecteurs d'entiers encodés par des tableaux comme suit.

```
exception ProduitScalaireImpossible
let ( %% ) (x: int array) (y: int array) : int =
  if Array.length x = Array.length y
  then
    let res = ref 0 in
    for i = 0 to Array.length x - 1 do
      res:= !res + x.(i) * y.(i)
    done; !res
  else raise ProduitScalaireImpossible
```

Après une telle définition le jeu de tests suivant est vérifié.

```
  assert ([|0; 0; 0|]    %% [|1; 2; 3|]    = 0 );
  assert ([|0; 1; 0|]    %% [|1; 2; 3|]    = 2 );
  assert ([|1; 1; 1; 1|] %% [|1; 2; 3; 4|] = 10)
```

a. Attention le symbole `::` n'est pas un opérateur de concaténation, en effet c'est un constructeur du type `'a list` (même s'il a un statut particulier qui lui permet d'être placé entre ses paramètres).

b. Les seules exceptions sont `( * )` et `( *. )` qu'il faut écrire en laissant un espace entre la parenthèse et l'étoile car l'enchaînement `(*` est réservé pour marquer un début de commentaire.

Manipulation de chaînes de caractères en OCaml

Les chaînes de caractères en OCaml sont représentées en mémoire comme des tableaux non mutables de caractères. Le type des chaînes de caractères est le type `string` qui est un alias pour `String.t`. Le module `String` fournit de nombreuses fonctions de manipulation des chaînes de carcatères.

Création. On peut fabriquer une chaîne de caractères :

- au moyen de sa description littérale : l'expression `"toto"` s'évalue en la chaîne de caractères `"toto"` de type `string` ;
- au moyen de la fonction `String.make : int -> char -> string` prenant en arguments un entier $n \geqslant 0$ et un caractère c et retournant la chaîne de caractères composée de n occurences du caractère c ;
- au moyen de la fonction de sélection d'une sous-chaîne `String.sub : string -> int -> int -> string` prenant en arguments une chaîne de caractères s, un indice de début i et une longueur l, et retournant le facteur de s de longueur l commençant à l'indice i ;
- au moyen de la fonction de concaténation de sous-chaînes `(^) : string -> string -> string` prenant en arguments deux chaînes de caractères et retournant leur concaténation, par exemple si `s1` vaut `"tata"` et `s2` vaut `"tutu"`, l'expression `s1^s2` s'évalue en `"tatatutu"`.

Accès.

- La fonction `String.length : string -> int` calcule la longueur d'une chaîne.
- Si `s` est une chaîne de longueur n et `i` un indice dans $[\![0, n[\![$, l'expression `s.[i]` vaut le caractère à la `i`-ème position dans `s`.

Modification. Il n'est **pas possible** de modifier une chaîne de caractères, au sens où il s'agit d'un tableau **non mutable** de caractères. Ainsi les expressions de la forme `s.[i] <- 'a'` ne sont pas syntaxiquement correcte. En revanche il est toujours possible de manipuler des références vers des `string`, ou des tableaux de caractères, ainsi on peut par exemple définir les fonctions[a] suivantes

```
let met_au_pluriel (sr:string ref) : unit =
  sr := (!sr)^"s"
```

```
let met_une_majuscule (c:char array) : unit =
  assert( Array.length c > 0);
  if ('a' <= c.(0) &&  c.(0) <= 'z')
  then let decalage = int_of_char 'A' -  int_of_char 'a' in
  c.(0) <- char_of_int ( int_of_char c.(0) + decalage )
  else ()
```

a. fonctions grossières qu'il est déconseillé de présenter aux prof de français...

Mesurer un temps d'exécution à partir du terminal

En Bash (la langage du terminal sous Debian notamment), la commande `time` permet de mesurer le temps d'execution d'une commande. Pour essayer, on peut par exemple l'utiliser avec la commande `sleep` qui prend en paramètre un entier `N` et qui a pour effet de laisser s'écouler `N` secondes avant d'être terminée.

```
user@machine:~/dossier$ time sleep 2

real  0m2,004s
user  0m0,001s
sys  0m0,003s
```

On peut simplifier un peu la sortie pour ne récupérer que le temps "real". Il suffit d'affecter la valeur `%R` à la variable bash `TIMEFORMAT`, comme dans l'exemple ci-dessous. Attention cette affectation [a] ne vaut que pour le terminal courant. Dans un autre terminal, ou après fermeture, il faut refaire l'affectation. En revanche si on garde le même terminal, il n'est pas utile de la refiare avant chaque appel à `time`.

```
user@machine:~/dossier$ TIMEFORMAT=%R
user@machine:~/dossier$ time sleep 2
2,004
user@machine:~/dossier$ time echo "bonjour"
bonjour
0,000
```

La commande `time` n'affiche le temps qu'à l'issue de l'exécution, de sorte qu'au cours d'une exécution qui paraît longue, cette commande ne permet pas de savoir depuis combien de temps elle dure. On pourra utiliser la commande `date` pour afficher l'heure au début et/ou à la fin d'une exécution, comme dans l'exemple suivant.

```
user@machine:~/dossier$ TIMEFORMAT=%R
user@machine:~/dossier$ echo "début à `date +"%Hh%Mm%S"`"; time sleep 3; echo
↪  "fin à `date +"%Hh%Mm%S"`"
début à 17h25m17
3,004
fin à 17h25m20
```

a. affectation à faire comme dans l'exemple, sans espace autour du =... Bash est un peu sensible

Commandes pour la navigation de fichiers en Bash

Voilà quelques commandes qui pourraient être utiles pour naviguer dans le système de fichiers depuis un terminal. Les commandes `cd`, `ls` et `pwd` sont à connaître.

`pwd`	donne le répertoire courant
`ls`	liste le contenu du répertoire courant.
`ls -l`	liste le contenu du répertoire courant avec des indications comme les droits de lecture, d'écriture et d'exécution, la date de création, la taille...
`cd` *dest*	change de répertoire, *dest* est le chemin (relatif ou absolu) du répértoire où l'on veut se rendre
`cd ..`	remonte dans le répertoire parent
`cat` *file*	affiche le contenu du fichier *file* dans le terminal est fait pour afficher du texte, à éviter sur les PDF
`mkdir` *dir*	crée un dossier vide et nommé *dir*
`echo` *chaine*	imprime la chaîne de caractères *chaine* sur la sortie standard du terminal
`man` *cmd*	imprime sur la sortie standard du terminal le manuel de la commande *cmd*
cmd `>` *file*	redirige la sortie standard du terminal dans le fichier *file* autrement dit l'affichage produit par la commande *cmd* n'est pas affiché à l'écran mais enregistré dans le fichier *file* Attention : le précédent contenu de *file* est écrasé
cmd `>>` *file*	redirige la sortie standard du terminal dans le fichier *file* autrement dit l'affichage produit par la commande *cmd* n'est pas affiché à l'écran mais enregistré dans le fichier *file*