TP d'Informatique MP2I

Julien Reichert

2024/2025

Ce document contient mes TP d'informatique pour l'année scolaire 2024/2025 en MP2I.

Les TP sont donnés dans l'ordre de traitement des chapitres et ont fait l'objet d'une réorganisation suite à la décision d'enseigner désormais le langage C avant le langage OCaml.

L'association entre les TP et un chapitre de cours, quasiment systématique, est précisée dans le TP et en cours.

Suivant le TP, un certain sous-ensemble des langages au programme sera utilisé. Il est exigé de respecter le sous-ensemble précisé, et les consignes des exercices ne rappelleront pas à chaque fois « ... en OCaml », « ...en C », etc.

Les premiers exercices des TP seront souvent des manipulations sommaires guidées, puis des exercices de niveau abordable à faire pendant la séance, et enfin des exercices plus difficiles garantissant normalement que personne ne finisse désœuvré pendant le quart du temps alloué. Dans certains cas, les exercices plus difficiles couvriront une partie du programme officiel non traitée par ailleurs, ils seront alors à terminer chez soi impérativement. Autrement, ce sera néanmoins recommandé de finir les TP après les séances.

Table des matières

TP 0 : Se préparer à la MP2I **5**

TP 1 : C - Prise en main basique **9**

TP 2 : Pointeurs et tableaux en C **15**

TP 3 : Structures en C **25**

TP 4 : OCaml - Prise en main basique **30**

TP 5 : OCaml - Prise en main avancée **34**

TP 6 : Types construits en OCaml **38**

TP 7 : Implémentation de structures de données **41**

TP 8 : Applications des piles, tables de hachage **47**

TP 9 : Arbres binaires et parcours d'arbres **53**

TP 10 : ABR, arbres bicolores et tas **57**

TP 11 : Des graphes en OCaml et en C **63**

TP 12 : Exploration exhaustive **67**

TP 13 : Sérialisation, codage de Huffman **71**

TP 14 : Formules propositionnelles et arbres **75**

TP 15 : Une base de données sommaire **79**

TP 16 : Une base de données avancée **81**

TP 0 : Se préparer à la MP2I

Ce TP consiste en quelque sorte en une *checklist* des prérequis en manipulation d'outils informatiques.

Il peut éventuellement être l'occasion de quelques découvertes, voire susciter des interrogations. Il ne faut pas hésiter à poser des questions suite à cela.

Prise en main des ordinateurs de la salle informatique

La salle informatique permet à tout le monde d'avoir accès au même environnement de travail, qui sera considéré comme l'environnement par défaut pour des explications dans les TP ultérieurs.

Un dossier contenant les lanceurs d'applications utiles figure sur le bureau sous Windows, on y retrouve notamment des IDE pour Python et OCaml (une demande a été déposée afin d'en avoir un pour C), et un très bon éditeur de texte supportant l'aide syntaxique pour de nombreux langages.

Il est important de se familiariser avec un environnement sur lequel on est amené à travailler, c'est pourquoi les épreuves de TP sur machine à l'oral se feront sur un environnement fourni à l'avance, qui sera éventuellement le même que celui des épreuves de modélisation de l'agrégation.

La « clé agrèg » contient cet environnement (on peut l'obtenir, avec instructions, ici : `https://interne.agreg.org/agregOS/` et `https://clefagreg.dnsalias.org/`), et on peut démarrer un ordinateur dessus sans toucher à sa configuration en paramétrant par exemple une clé USB à l'aide d'un créateur de disque de démarrage.

Quelques raccourcis clavier sur PC

On dit souvent qu'un bon programmeur ne se sert pas de sa souris (évidemment sans la remplacer par un pavé tactile), et c'est essentiellement vrai.

En pratique, toute la partie graphique de la manipulation informatique consiste en une façon ergonomique de passer des commandes, qui peuvent être saisies sur un terminal, rendant effectivement la souris non nécessaire.

Ceci étant, deux choses sont nécessaires pour se passer de souris : maîtriser les commandes système, et maîtriser les raccourcis clavier.

Au niveau des raccourcis clavier pour PC, les plus connus sont désormais incontournables même pour des utilisateurs occasionnels, par leur entrée dans le langage courant. Voici une liste un peu plus fournie (les raccourcis sont plutôt universels, mais parfois limités à Windows ou Linux, ou spécifiques à un (type de) logiciel) :

- Ctrl + C : copier la sélection.
- Ctrl + V : coller la sélection. [1]
- Ctrl + X : couper la sélection. Sur un tableur, l'effacement n'est effectif qu'une fois la cellule ou la zone collée ailleurs.
- Ctrl + Z : annuler la dernière « action ».
- Ctrl + Y (ou Ctrl + Maj + Z, suivant l'application) : rétablir la dernière « action » annulée. Pour information, la touche Maj est celle qu'on maintient pour produire une majuscule, à ne pas confondre avec Verr. Maj.
- Ctrl + A : tout sélectionner.
- Ctrl + Maj + A (ou Ctrl + I, et parfois aucun raccourci n'est implémenté) : inverser la sélection. C'est particulièrement utile si on veut tout sauf une ou plusieurs choses, où les sélections multiples peuvent se faire en maintenant Ctrl entre plusieurs clics (maintenir Maj entre deux clics sélectionne une plage entre les deux objets sur lesquels on a cliqué).
- Ctrl + F : rechercher.
- Ctrl + G : aller à l'occurrence suivante.
- Ctrl + Maj + G : aller à l'occurrence précédente.
- Ctrl + H (parfois Ctrl + R) : rechercher et remplacer.
- Ctrl + R : actualiser la page ou le dossier.
- Ctrl + T : ouvrir un nouvel onglet d'un navigateur / un nouveau fichier d'un éditeur de texte ou IDE.
- Ctrl + W : fermer l'onglet / le fichier.
- Ctrl + Q : quitter.
- Ctrl + D (dans une console) : quitter la console. Le raccourci Ctrl + C y arrête un programme, réassignant les raccourcis pour copier, coller et couper.
- Alt + Tab : basculer entre les fenêtres (Maj + Alt + Tab pour basculer entre les fenêtres dans l'autre sens).

1. À ce sujet, les systèmes Linux disposent d'un deuxième tampon pour copier-coller : toute sélection ou presque peut être collée ailleurs par le clic sur la molette de la souris, rare cas d'utilisation particulière d'une souris.

Le terminal unix

Le langage *bash* est utilisé dans les consoles des systèmes Unix. Il n'est pas à maîtriser obligatoirement en CPGE, mais son utilité dans la vie de tous les jours (... sur un système Unix) fait que sa découverte est recommandée aussi tôt que possible.

Les commandes de base permettent la navigation dans l'arborescence des dossiers, la manipulation (création, copie, déplacement, destruction) de fichiers, leur affichage, etc.

Une bonne initiation à ce langage, ludique de surcroît, est le jeu Terminus, créé au MIT et dont une adaptation française a été réalisée. Les liens : `https://www.mprat.org/Terminus/` (en anglais) et `https://luffah.xyz/bidules/Terminus/` (en français).

C'est l'occasion d'encourager également à posséder un ordinateur muni d'un système d'exploitation basé sur Unix, sur lequel le travail sera d'autant plus agréable.

TP 1 : C - Prise en main basique

L'organisation des séances de cette année fait que ce TP ainsi que le suivant seront les premiers contacts avec le langage C, avant toute séance de cours apportant des explications plus techniques sur le langage.

Une demande a été déposée pour le déploiement en salle de TP de l'environnement de développement intégré Code::Blocks, qu'il est recommandé d'installer sur son propre ordinateur pour être certain que toutes les instructions du TP soient pertinentes.

En l'absence d'outils sur l'ordinateur que l'on manipule, une solution en ligne est suggérée : OnlineGDB, à l'adresse `https://www.onlinegdb.com/online_c_compiler`. On notera avec satisfaction que tous les langages au programme et bien d'autres encore sont supportés.

On commencera par lire, préalablement à ce TP, le début du chapitre associé au langage C dans les notes de cours.

Premier programme

Recopier dans la zone éditeur de l'IDE retenu le code ci-après.

[Cette consigne sera désormais tacite]

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

int main()
{
  printf("Je suis un programme en C.\n");
  exit(0);
}
```

Compiler et exécuter.

[De même, tout code présenté en TP sera à tester ainsi.]

Fonctions supplémentaires

Nous allons déléguer l'impression à une fonction extérieure.

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

void message()
{
  printf("Je suis un programme en C avec une fonction extérieure.\n");
}

int main()
{
  message();
  exit(0);
}
```

Création de variables

Comme dans le cas des fonctions, pour déclarer une variable on commence par annoncer son type. Il est possible de faire la déclaration et l'initialisation dans deux instructions différentes. On notera que la fonction `printf` est réservée aux chaînes de caractères avec formats éventuels, donc nous allons en profiter pour voir le paramétrage de la fonction `printf` avec des formats.

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

int main()
{
  int x;
  x = 6;
  int y = 7;
  printf("x vaut %d et y vaut %d.\n", x, y);
  exit(0);
}
```

Calculs

Rien de surprenant dans la gestion de l'arithmétique par le langage C. On peut se référer au chapitre pour les détails des opérations. L'important ici est de voir la syntaxe qui s'adapte avec une fonction extérieure prenant deux arguments entiers.

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

int mult(int a, int b)
{
  return a * b;
}

int main()
{
  int x = 6;
  int y = 7;
  int xy = mult(x, y);
  printf("Le produit de %d par %d est %d.\n", x, y, xy);
  exit(0);
}
```

Structures de contrôle

Nous allons écrire une première fois un test conditionnel, une boucle conditionnelle et une boucle `for`.

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

int main()
{
  int x = 612657948;
  if (x % 3 == 0) printf("%d est multiple de trois.\n", x);
  else printf("%d n'est pas multiple de trois, le reste est %d.\n", x, x % 3);
  exit(0);
}
```

À présent, la syntaxe avec `while` :

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

int main()
{
  int x = 612657948;
  printf("Les chiffres de %d en partant de la droite sont :\n", x);
  if (x == 0) printf("0\n");
  while (x != 0)
  {
    printf("%d\n", x % 10);
    x /= 10;
  }
  exit(0);
}
```

Et on reprend l'exemple de la divisibilité en répétant les tests :

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

int main()
{
  int x = 612657948;
  printf("Testons la divisibilité de %d par les entiers de 2 à 11.\n", x);
  for (int i = 2 ; i < 12 ; i += 1)
  {
    if (x % i == 0) printf("%d est multiple de %d.\n", x, i);
    else printf("%d n'est pas multiple de %d.\n", x, i);
  }
  exit(0);
}
```

C'est tout pour les manipulations de base, permettant de faire quelques exercices simples.

Pour ce TP, la consigne associée à chaque exercice revenant à écrire une fonction `f` est la suivante : après l'écriture de la fonction, écrire un programme avec une fonction `main` qui crée la ou les variable(s) nécessaire(s) pour que `f` puisse s'exécuter, puis appelle `f` sur cette ou ces variable(s) et fait une impression permettant d'observer le résultat.

Exercice 1 : Écrire en C trois fonctions prenant en entrée deux entiers `a` et `b` (ce dernier étant positif) et retournant `a` puissance `b`. Faire une version itérative simple, une version récursive simple et une version itérative ou récursive procédant par exponentiation rapide.

Exercice 2 : Écrire en C une fonction prenant en entrée un entier positif et retournant la somme de ses chiffres.

Exercice 3 : Écrire en C une fonction prenant en entrée deux entiers `n` et `k` et retournant la somme des puissances `k`-ièmes des entiers de `1` à `n`. On admettra que les deux entiers sont positifs et le comportement est au choix sinon.

Exercice 4 : Écrire en C une fonction prenant en entrée un entier positif et retournant son nombre de chiffres en binaire.

Exercice 5 : Écrire en C une fonction prenant en entrée un entier positif et imprimant sa représentation en binaire avec le bit de poids fort en premier.

Exercice 6 : Écrire en C une fonction prenant en entrée deux entiers strictement positifs `n` et `x` tels que `x` soit inférieur ou égal à `n` et imprimant le résultat d'une recherche dichotomique menée par un programme devant trouver efficacement et à l'aveugle la valeur `x`.

Pour l'exercice 6, le rendu devra être le suivant en prenant par exemple `n` valant `19` et `x` valant `12` :

```
Ordinateur : 10 ?
Arbitre : C'est plus !
Ordinateur : 15 ?
Arbitre : C'est moins !
Ordinateur : 12 ?
Arbitre : C'est bon !
```

Exercice 7 : Écrire en C une fonction prenant en entrée deux entiers et retournant leur plus petit commun multiple.

Exercice 8 : Écrire en C une fonction prenant en entrée un entier et retournant un booléen déterminant si l'entier est premier.

TP 2 : Pointeurs et tableaux en C

Dans ce TP, la deuxième partie du chapitre portant sur le langage C sera illustrée, avec des manipulations et exercices sur les pointeurs et les tableaux notamment, dont les chaînes de caractères. Dans la mesure où ces seuls thèmes sont essentiels et permettent d'écrire un contenu déjà ambitieux, le reste du chapitre fera l'objet d'autres séances.

Pointeurs

Nous allons commencer par écrire deux fonctions qui tentent de modifier une variable extérieure, et comprendre l'utilité des pointeurs dans ce contexte tout simple.

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

void mutepas (int x)
{
  x = x + 1;
}
// x était une variable locale, muette par ailleurs

void mute (int *nom)
{
  *nom = *nom + 1;
}
// On signale bien que le nom n'importe pas.

int main()
{
  int x = 42;
  mutepas(x);
  printf("Après avoir fait mutepas, x vaut %d\n", x);
  mute(&x);
  printf("Après avoir fait mute, x vaut %d\n", x);
  exit(0);
}
```

Attention tout de même, en raison du passage par valeurs, tenter de muter une variable mise en argument en tant qu'elle-même va échouer :

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

void mutepasnonplus (int x)
{
  int *nom = &x;
  *nom = *nom + 1;
  printf("Ici, la valeur de x est %d\n", x);
}

int main()
{
  int x = 42;
  mutepasnonplus(x);
  printf("Après avoir fait mutepasnonplus, x vaut %d\n", x);
  exit(0);
}
```

On peut connaître l'adresse mémoire exacte pointée avec `%p`.

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

int main()
{
  int x = 42;
  int y = 73;
  printf("L'adresse de x est %p et celle de y est %p\n", &x, &y);
  exit(0);
}
```

L'exécution de l'exemple laisse voir une différence, exprimée en octets, valant parfois quatre (attention à bien compter en hexadécimal), sans qu'il n'y ait de garantie que les deux variables soient systématiquement mises côte à côte. En tout cas, une différence de quatre est un signe que le type `int` tient sur 32 bits et non pas 64.

Terminons la section par un exemple de création de pointeur avec `malloc`.

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

int main()
{
  int *p = malloc(sizeof(int));
  printf("La zone mémoire où se situe p est %p.\n", p);

  *p = 42;
  int x = *p;
  printf("La valeur de x, initialisée via p, est %d\n", x);

  free(p);

  printf("La valeur de x a survécu, c'est toujours %d\n", x);
  printf("La valeur de p a survécu, c'est toujours %p\n", p);

  int *p2 = malloc(sizeof(int));
  printf("La zone mémoire où se situe p2 est %p.\n", p2);

  free(p2);

  exit(0);
}
```

On pourra modifier le programme pour constater que l'adresse où `x` a été stocké est différente de `p`, garantissant que `x` ne pouvait pas disparaître par la libération de `p` et ne serait pas impactée par un travail avec `p2`, même lorsque l'adresse est la même.

On en profitera aussi pour constater que dans l'exemple introduisant la fonction `mutepasnonplus` précédente, l'adresse de `x` dans la fonction `main` et l'adresse de ce même `x` en tant qu'argument de la fonction `mutepasnonplus` sont différentes, justifiant l'absence d'effet dans ce cas précis.

Tableaux

Dans cette section aussi, nous allons créer des tableaux de deux façons différentes, avec ou sans initialisation au moment de la déclaration.

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

int main()
{
  int t[] = { 1 , 2 , 6 };
  int mat[][3] = { { 2 , 9 , 4 } , { 7 , 5 , 3 } , { 6 , 1 , 8 } };
  printf("L'élément à la position 1 de t est %d.\n", t[1]);
  printf("L'élément mat[2][0] est %d.\n", mat[2][0]);
}
```

Un tableau donné explicitement ne nécessite pas le nombre d'éléments dans les crochets, mais cela ne s'applique pas aux tableaux de dimension supérieure, où le nombre d'éléments par tableau est à préciser.

On se souviendra que le nombre d'éléments d'un tableau n'est pas récupérable, il y a une certaine cohérence ici.

Pour une déclaration sans initialisation, on écrira toujours la taille. Idem en dimensions supérieures.

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

int main()
{
  int t[3];
  for (int i = 0 ; i < 3 ; i += 1) t[i] = i * i;
  printf("L'élément à la position 2 de t est %d.\n", t[2]);
}
```

Par principe, on ne se servira pas d'indices interdits, même en sachant leur gestion par C.

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

int main()
{
  int mat[2][3];
  for (int i = 0 ; i < 3 ; i += 1)
  {
    mat[0][i] = i;
    mat[1][i] = 10 - i;
  }
  printf("L'élément mat[1][0] est %d.\n", mat[1][0]);
}
```

Conformément à ce qui était dit précédemment, si une fonction doit prendre un tableau en argument, il est très probable que sa taille soit importante pour délimiter une boucle par exemple. Elle sera aussi en argument. On notera au passage le prototype de la fonction.

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

int somme(int t[], int n)
{
  int rep = 0;
  for (int i = 0 ; i < n ; i += 1) rep += t[i];
  return rep;
}

int main()
{
  int t[] = { 1, 2, 6, 24 };
  printf("La somme des éléments de t est %d.\n", somme(t, 4));
}
```

Pour les langages où la taille d'un tableau peut être récupérée, mettre la taille en argument est très discutable, par ailleurs.

Une remarque sur l'utilisation de tableaux de dimension supérieure : on évitera de tenter d'écrire des fonctions agissant dessus. En pratique, c'est plutôt un pointeur qui est passé en argument qu'un tableau, dans l'esprit.

Un exemple tout de même si vraiment on ne compte pas faire autrement :

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

int somme2(int **tab, int lignes, int colonnes)
{
  int rep = 0;
  for (int i = 0 ; i < lignes ; i += 1)
  {
    for (int j = 0 ; j < colonnes ; j += 1)
    {
      rep += tab[i][j];
    }
  }
  return rep;
}

int main()
{
  int tab[3][3] = { { 4 , 3 , 8 } , { 9 , 5 , 1 } , { 2 , 7 , 6 } };
  int *tab_pointe[3] = {tab[0], tab[1], tab[2]};
  printf("Somme des éléments de tab : %d.\n", somme2(tab_pointe, 3, 3));
  exit(0);
}
```

Par ailleurs, il n'est pas non plus évident de créer un tableau dans une fonction et de le retourner. Une option est de créer le tableau dans le contexte qui appelle la fonction, de mettre un pointeur associé en argument pour que la fonction mute le tableau en question, une autre est d'utiliser `malloc` au moment de la création du tableau dans la fonction.

Chaînes de caractères

Les chaînes de caractères seront considérées ici comme des tableaux de caractères finissant forcément par le caractère `\0`. En outre, si ce caractère est présent plus tôt, la chaîne sera considérée comme plus petite.

La taille à allouer au tableau est alors le nombre de caractères en-dehors de ce caractère final plus un, et il ne faut vraiment pas oublier ce caractère en plus, au risque de voir cette délimitation écrasée et la chaîne se prolonger.

Pour manipuler les chaînes de caractères, on peut utiliser des fonctions nécessitant `string.h` :

- `strlen` : calcule la taille d'une chaîne, sachant que la complexité est linéaire. Attention, le type de retour est `size_t`, autrement dit `unsigned long int` d'après les avertissements, on évitera de mélanger avec des entiers sans caster (en théorie non pénalisable).
- `strcpy` : copie une chaîne (deuxième argument, dont son `\0` final) dans une autre (premier argument). C'est d'ailleurs l'adresse de la destination qui est renvoyée (on aurait pu penser à un simple effet de bord, mais non... l'avantage est de permettre d'enchaîner avec la fonction suivante). Il faut penser à allouer initialement suffisamment de mémoire.
- `strcat` : concatène à une chaîne (premier argument) une autre (deuxième argument). Il s'agit d'écraser le `\0` actuel en recopiant l'intégralité de l'autre chaîne, dont le `\0` final.

Ajoutons enfin la fonction de gestion de la saisie utilisateur, disponible dans `stdio.h` : la fonction `scanf`. Elle utilise la même syntaxe de formats que `printf`, et stocke dans une variable du bon type la ligne saisie, voire, dans une utilisation avancée, cherche à reconnaître dans la ligne récupérée un format en affectant des variables adéquates. En tout cas, les arguments correspondant aux formats reconnus doivent forcément être des adresses mémoires (donc pointeurs d'entiers, chaînes de caractères, etc.). **Attention cependant : une espace interrompt un format, même `\%`.**

La fonction `scanf` a deux variantes : `fscanf` qui a un argument préalable précisant un fichier d'où la saisie est récupérée, et `sscanf` qui a un argument supplémentaire en tant que chaîne remplaçant la saisie de l'utilisateur.

Leur utilisation est clarifiée par les démonstrations ci-après.

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

int main()
{
  char s[30] = "Bonjour le monde !";
  char t[] = "Hello world!";
  printf("La taille de s est %lu.\n", strlen(s)); // 18, pas 30
  strcpy(s, t);
  printf("Désormais, s est %s.\n", s);
  strcat(s, t);
  printf("Désormais, s est %s.\n", s);
  s[4] = '0';
  printf("Désormais, s est %s et t est %s.\n", s, t);
  // un seul caractère est modifié et t est préservé
  exit(0);
}
```

Au risque d'insister, il ne faut pas déborder de ce qui est alloué en utilisant `strcat`, car on ne remarquera rien au niveau de la chaîne, alors qu'on a écrit hors champ dans une zone mémoire qui peut appartenir à une autre variable, d'où un *UB*.

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

int main()
{
  char ligne[80];
  scanf("%s", ligne); // le caractère de fin de ligne n'est pas reporté
  printf("Vous avez écrit \"%s\".\n", ligne);
  int x;
  int y;
  scanf("%d-%d", &x, &y);
  // ici il faut saisir deux entiers séparés par un trait d'union
  printf("J'ai mis %d dans x et %d dans y.\n", x, y); // tester des négatifs
  exit(0);
}
```

Une autre méthode pour créer une chaîne à partir d'un entier sans passer par `sprintf` est d'utiliser la fonction `itoa`. Contrairement à `atoi` qui permet avec une syntaxe intuitive de récupérer l'entier stocké dans une chaîne (avec un comportement indéterminé si la chaîne ne correspond pas correctement à un entier, faisant que la fonction `atoi` est trop peu sécurisée pour être recommandée en pratique), la fonction `itoa` aura trois arguments : un entier et une chaîne de caractères dont on doit s'être assurée que la place allouée est suffisante (en comptant bien entendu aussi le caractère `\0`) pour éviter les comportements indéterminés, et la base à considérer pour la conversion.

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

int main()
{
    char buffer[20];
    itoa(42, buffer, 2);
    printf("buffer : %s\n", buffer);

    return 0;
}
```

Ceci étant, quelques compilateurs ne disposent pas de cette fonction, mieux vaut se rabattre sur d'autres solutions dans le doute.

Exercice 1 : Écrire une fonction qui permet d'échanger la valeur de deux variables entières (on réfléchira au prototype à utiliser).

Exercice 2 : Écrire une fonction qui prend en argument un tableau d'entiers de taille au moins deux et qui échange le premier et le deuxième élément.

Exercice 3 : Écrire une fonction qui prend en argument un tableau d'entiers de taille au moins deux et sa taille et qui envoie le premier élément à la fin, en décalant tous les autres.

Exercice 4 : Écrire une fonction qui prend en argument un tableau d'entiers et sa taille ainsi qu'un entier et qui détermine le premier indice où l'entier se trouve, en renvoyant `-1` s'il n'y est pas.

Exercice 5 : Écrire une fonction qui prend en argument un tableau d'entiers et sa taille ainsi qu'un entier et qui détermine le nombre de fois où l'entier se trouve dans le tableau.

Petite remarque au passage : il n'est pas prévu qu'une fonction renvoie un couple en C, une astuce usuelle revient à renvoyer une des valeurs et faire ressortir l'autre par la mutation d'une valeur pour laquelle un pointeur a été passé en argument. De là à n'écrire que des fonctions qui font un effet de bord et préparer une variable pour ce qui a été calculé...

Exercice 6 : Écrire une fonction qui prend en argument un tableau d'entiers et sa taille et qui le trie de manière croissante avec le tri à bulles.

Exercice 7 : Écrire une fonction qui prend en argument un tableau de dimension deux d'entiers, son nombre de lignes et son nombre de colonnes et qui détermine la première position du maximum en tant que tableau d'entiers de taille deux.

Exercice 8 : Écrire une fonction qui prend en argument une chaîne de caractères et qui calcule sa taille exactement comme `strlen`, bien évidemment sans s'en servir.

Exercice 9 : Écrire une fonction qui prend en argument une chaîne de caractères et qui détermine s'il s'agit d'un palindrome, c'est-à-dire si elle s'écrit de la même manière à l'endroit et à l'envers.

Exercice 10 : Écrire une fonction qui prend en argument une chaîne de caractères et un pointeur vers une autre chaîne dont on supposera que suffisamment de mémoire lui est allouée et qui recopie dans cette autre chaîne la chaîne en argument à ceci près que les majuscules deviennent des minuscules et vice-versa. Les autres caractères ne doivent pas être impactés.

TP 3 : Structures en C

Les structures de C ont droit à un TP exclusif.

Des types créés pour les besoins d'un algorithme seront rencontrés régulièrement à partir du deuxième semestre, donc la maîtrise des opérations de manipulation de ce genre d'objets est cruciale.

Syntaxe de base

Une démonstration pour illustrer la syntaxe donnée en cours, dont la mutation en se servant obligatoirement de pointeurs et le renommage d'un type.

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

struct Complexe { double re; double im; };
struct Complexe2 { double *re; double *im; };

typedef struct Complexe complexe;
typedef struct Complexe2 complexe2;

complexe conjugue (complexe z)
{
    complexe reponse = { .re = z.re , .im = -z.im};
    return reponse;
}

void conjugaison (complexe *z)
{
    z->im = -z->im;
}

void conjugaison2 (complexe2 z)
{
    *z.im=-*z.im;
}
```

```
int main()
{
    complexe i = { .re = 0 , .im = 1 };
    complexe ibarre = conjugue(i);
    printf("ibarre vaut %f %+f i\n", ibarre.re , ibarre.im);

    double x = -.5;
    double y = sqrt(3)/2;
    complexe2 j = { .re = &x , .im = &y};
    conjugaison2(j);
    printf("j vaut %f %+f i\n", *j.re , *j.im);
    exit(0);
}
```

En effet, une tentative de mutation sans passer par les pointeurs échouerait :

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

struct Complexe { double re; double im; };

typedef struct Complexe complexe;

void conjugaison_ratee (complexe z)
{
    z.im = -z.im;
}

int main()
{
    complexe i = { .re = 0 , .im = 1 };
    conjugaison_ratee(i);
    printf("i vaut encore %f %+f i\n", i.re , i.im);

    exit(0);
}
```

Exercices sur une structure de base

La structure que nous allons utiliser peut servir d'appui à un RPG très sommaire. Voici le code qui permet de la créer :

```
struct Personnage { char nom[80]; int pv; int atq; int def; };
typedef struct Personnage personnage;
```

Explications rapides : les champs sont le nom (maximum 79 caractères), le nombre de points de vie, la puissance d'attaque et la valeur de défense.

Une attaque d'un personnage sur un autre retranche au nombre de points de vie de ce dernier l'attaque du premier moins la défense de l'autre, en ramenant à 1 toute valeur négative du nombre de points de vie retranchés.

Si le nombre de points de vie tombe dans le négatif, il est ramené à zéro, et le personnage ne peut plus attaquer.

Une remarque importante : il faut déclarer une valeur constante pour le nombre d'éléments de tout tableau qui est un champ d'une structure.

Ceci est évidemment lié au besoin de savoir combien de mémoire il faut pour stocker une instance de la structure créée.

Pour ce TP, le travail va s'accumuler avec des tests réguliers de tout nouveau contenu écrit, plutôt que de se séparer sur différents onglets / fichiers.

Exercice 1 : Créer deux personnages. Inutile de faire intervenir des saisies de l'utilisateur.

Exercice 2 : Écrire une fonction qui imprime les caractéristiques d'un personnage.

Exercice 3 : Écrire une fonction effectuant l'attaque d'un personnage (le premier mentionné) sur un autre. On ajoutera un paramètre booléen signalant s'il faut imprimer ou non un compte-rendu de l'attaque.

Exercice 4 : Écrire une fonction effectuant des attaques entre deux personnages jusqu'à ce que l'un des deux ne puisse plus attaquer.

Exercice 5 : Créer deux tableaux de personnages.

Exercice 6 (facile...) : Écrire une fonction qui imprime les caractéristiques de tous les personnages d'un tableau.

Exercice 7 : Écrire une fonction effectuant des attaques entre des personnages du premier tableau et des personnages du deuxième tableau jusqu'à ce qu'un des tableaux ne contienne plus de personnage pouvant attaquer. Tous les personnages d'un tableau attaquent une seule cible puis le tour passe. La stratégie est totalement libre (si tant est qu'on en fasse une). On pourra aussi décider d'imprimer ou non les détails.

Exercices sur une structure utile

Nous allons cette fois-ci considérer une structure de date :

```
struct day { int annee; int mois; int jour; int jdls; };
typedef struct day jour;
```

Le champ `jdls` correspond au jour de la semaine, avec la convention que le lundi est associé à `1` et le dimanche à `7`. Les mois sont numérotés entre `1` et `12`. Ce champ ne sera pas forcément initialisé.

Exercice 8 : Créer la date du jour et son propre anniversaire sans précision du jour de la semaine, même si on le connaît.

Exercice 9 : Écrire une fonction prenant en argument une date et imprimant les informations associées de manière propre.

Exercice 10 : Écrire une fonction prenant en argument un entier correspondant à l'indice d'un mois (pas de bissextile) et renvoyant son nombre de jours.

Exercice 11 : Écrire une fonction prenant en argument deux structures de date et renvoyant le nombre de jours entre la première et la deuxième, strictement positif si la deuxième est ultérieure à la première et négatif sinon.

Exercice 12 : Écrire une fonction permettant de renseigner le jour de la semaine dans une structure de date en s'appuyant sur une structure existante avec un jour de la semaine déjà connu.

TP 4 : OCaml - Prise en main basique

Pour une familiarisation aussi rapide que possible avec la syntaxe en OCaml, nous allons écrire ici des lignes de code simples en nombre.

Chaque expression se termine par deux points-virgules (**ce n'est pas nécessaire pour les programmes compilés !**), et l'éditeur de base valide la totalité du programme sur l'appui de la touche entrée ; la touche entrée d'un éventuel pavé numérique permet de faire un retour à la ligne, ce qui est pratique si on ne veut pas avoir à copier-coller le caractère de retour à la ligne soi-même.

L'éditeur proposé, quant à lui, valide l'expression mise en lumière sur la touche entrée de l'éventuel pavé numérique ou la combinaison Ctrl + entrée, le retour à la ligne étant provoqué par la touche entrée usuelle.

Cet éditeur est WinCaml, développé par Jean Mouric. On pourra le télécharger ici : `https://www.wincaml7.fr/` (sur cette même page se trouve également MacCaml pour les utilisateurs d'Apple).

Les commentaires à la suite du code sont les passages délimités par des parenthèses accompagnées d'étoiles.

Pour pallier l'éventuelle absence d'éditeur sur un ordinateur (solution temporaire si on a la main pour installer des programmes !), il existe des interpréteurs en ligne, on pourra rechercher sur son moteur favori TryOCaml (`https://try.ocamlpro.com/`).

Voici les premières expressions à évaluer :

```
2 + 2;;
1 + 1 / 2;; (* 1 / 2 donne 0 *)
1 + 0.5;; (* interdit *)
.5;; (* interdit *)
2.;; (* autorisé *)
2. + 0.5;; (* erreur *)
2. +. 0.5;; (* correct *)
1. +. 1. /. 2.;; (* c'est long, mais il n'y a pas le choix, sauf si... *)
float_of_int(1) +. float_of_int(1) /. float_of_int(2);; (* mieux ? *)
```

```
float_of_int 42;; (* pas besoin de parenthèses pour les fonctions,
mais attention aux cas ambigus *)
cos 0;; (* interdit *)
cos;; (* la preuve *)
cos 3.141592;; (* pi n'est pas implémenté,
sauf en tant que acos (-.1.) par exemple *)
cos 0.42 ** 2. +. sin 0.42 ** 2.;; (* priorité à l'évaluation
de la fonction, et bonjour l'arrondi *)

true and false;; (* interdit *)
true && false;;
true = not false;;
```

Puisque la programmation fonctionnelle sera vue en premier, voici les listes, pour commencer simplement des constantes :

```
[1; 2; 3];;
1::2::[3];;
1::2;; (* interdit *)
[2]::1;; (* interdit *)
[1]::[2];; (* interdit *)
[1]::[[2]];; (* relevez bien le type *)
[1]@[2];; (* se méfier de ce symbole *)
```

À présent, c'est l'heure de manipuler des variables :

```
let x = 2;;
let y = 4;;
x + y;;
let x = 3;; (* beurk ! *)
x + y;;
let x = 2 * x;; (* double beurk ! *)
x + y;;
let x = 42 in x * x;;
x;; (* la définition locale est oubliée *)
let z = 42 in print_int z;;
z;; (* la preuve, voilà une erreur *)
```

```
let l = [1; 2; 3];;
0::l;;
[4; 5; 6]@l;;
let l = 0::l;; (* beurk ! *)
[4; 5; 6]@l;;
```

Les tests conditionnels ont la syntaxe suivante :

```
if 2 = 3 then 73 else 42;; (* RAS *)

let x = 3*3*3*3*3*3 in if x mod 7 = 1 then 1 else -1;;
(* Exercice d'arithmétique : anticiper *)

if true then 42 else 4.2;; (* interdit en raison du typage *)
```

Maintenant, passons aux fonctions :

```
let add x y = x + y;;

let add1 (x, y) = x + y;;

let addl [x; y] = x + y;; (* filtrage non exhaustif *)

let cast_bool n = if n = 0 then false else true;;

let cast_bool_propre n = n <> 0;;

let cast_bool_not_propre n = n = 0;; (* surprenant, non ? *)
```

Et terminons par des fonctions récursives :

```
let rec somme_liste l = match l with
| [] -> 0
| a::q -> a + somme_liste q;;

let rec taille_liste l = match l with
| [] -> 0
| _::q -> 1 + taille_liste q;;
```

```
let rec pow valeur exposant = match exposant with
| 0 -> 1
| i -> valeur * (pow valeur (i - 1));;
(* On peut remplacer i - 1 par exposant - 1 car les deux noms coexistent. *)

let rec part_ent_log2 nombre =
  if nombre = 0 then failwith "Boum !";
(* Il est toujours utile de connaître les exceptions tôt. *)
  if nombre = 1 then 0
  else 1 + (part_ent_log2 (nombre / 2));;

let rec est_dans_liste element l = match l with
| [] -> false
| a::q -> a = element || est_dans_liste element q;;
```

Exercice 1 : Si ce n'est pas déjà fait, tester les fonctions récursives ci-avant. Avant de les tester, tenter de se convaincre qu'elles font bien ce que leur nom annonce en lisant leur code.

Exercice 2 : Écrire une fonction récursive qui prend en entrée une liste et qui retourne son plus grand élément.

Exercice 3 : Écrire une fonction récursive qui prend en entrée une liste de **flottants** et qui retourne le produit de ses éléments. Vérifier que la signature est correcte et faire attention aux erreurs de typage.

Exercice 4 : Écrire une fonction récursive qui prend en entrée une liste et qui détermine si ses éléments sont dans l'ordre croissant.

Exercice 5 : Écrire une fonction récursive qui prend en entrée une liste et qui détermine si ses éléments sont dans l'ordre croissant ou dans l'ordre décroissant (auquel cas la réponse est `true`, et dans tous les autres cas la réponse est `false`).

TP 5 : OCaml - Prise en main avancée

Ce TP est aussi l'occasion d'aborder la programmation impérative en OCaml.

Commençons par étudier les structures de tableaux et de chaînes de caractères.

Tout d'abord, création et manipulation de constantes :

```
[|1; 2; 3|];;
[|1; 2; 3|].(1);;
[|1; 2.5; 4|];; (* interdit *)
[|1; 2|] + [|3; 4|];; (* interdit *)
Array.make 10 42;;

"Bonjour";;
"Bonjour".[0];;
String.make 10 'b';;
'A';;
'ABC';; (* erreur *)
'\n';; (* pas erreur, ceci compte comme un caractère *)
String.length "\\\\\n\t\\";; (* la preuve *)
```

Ensuite, manipulation de variables, et c'est l'occasion de découvrir le type `unit` :

```
let t = [|1; 2; 3|];;
t.(0);;
t.(0) <- 0;;
t.(0);;
t.(3);;
t.(-1);;
Array.length t;;
length t;; (* erreur *)

let s = "Bonjour";;
s.[3];;
s.[0] <- "b";; (* erreur *)
s.[0] <- 'b';; (* erreur dans les dernières versions d'Ocaml *)
print_string s;;
```

```
String.sub s 2 3;;
```

Les structures de contrôle (tests déjà vus précédemment, ainsi que les boucles) :

```
let x = 4059234245 in
  if x mod 3 = 0 then print_string "Multiple de trois"
  else print_string "Pas multiple de trois";;

if true then false;; (* interdit pour des raisons de typage *)

if true then print_string "Autorisé";;
(* Le type unit n'est pas compatible avec d'autres *)

for i = 1 to 10 do print_int i done;;

for i = 1 to 10 do print_int i; print_newline () done;;
(* do et done parenthèsent de manière non ambiguë *)

for i = 0 to 1 do 42 done;;

for i = 10 to 0 do print_int (1 / 0) done;; (* sauvé, c'est vide *)

for i = 10 downto 0 do print_int (1 / 0) done;; (* boum ! *)

let t = [|42|] in (* sale, mais les références seront traitées plus tard *)
  while t.(0) > 0 do
    print_int (t.(0) * t.(0)); (* ** 2. réservé aux flottants *)
    t.(0) <- t.(0) - 1
  done;;
```

Il est essentiel de savoir manipuler les références, et de savoir quand les utiliser.

À ce titre, quelques nouvelles fonctions, dont la version itérative de certaines fonctions récursives du TP précédent (on exclut le calcul de la taille d'une liste, car une boucle sur une référence de liste, c'est trop moche) :

```
let print_int_array tab =
  for i = 0 to Array.length tab - 1 do print_int tab.(i) done;;
```

```
let somme_tableau tab =
  let s = ref 0 in
    for i = 0 to Array.length tab - 1 do
      s := !s + tab.(i)
    done; !s;;

let pow valeur exposant =
  let reponse = ref 1 in
    for i = 1 to exposant do
      reponse := !reponse * valeur
    done; !reponse;;

let part_ent_log2 nombre =
  if nombre = 0 then failwith "Boum !";
  let reponse = ref 0 and n = ref 1 in
    while !n <= nombre do
      incr reponse;
      n := !n * 2
    done; !reponse - 1;;

let est_dans_tableau element tab =
  let reponse = ref false in
    for i = 0 to Array.length tab - 1 do
      if tab.(i) = element then reponse := true
    done; !reponse;;
```

Exercice 1 : Créer le tableau des 42 premiers entiers naturels.

Exercice 2 : Créer une chaîne de caractères quelconque de taille au moins 10. Créer le tableau de caractères correspondant. Remplacer tous les caractères d'indice pair par des espaces. Récupérer la chaîne correspondante à la fin.

Exercice 3 : Créer un tableau de taille au moins 2. Échanger les 2 premiers éléments.

Exercice 4 : Créer un tableau de caractères quelconque de taille au moins 10. Le modifier de sorte que le premier caractère soit envoyé à la fin, tout le reste subissant donc un décalage. Récupérer la chaîne correspondante à la fin.

Exercice 5 : Écrire une fonction récursive calculant la somme des éléments d'un tableau.

Exercice 6 : Écrire une fonction récursive déterminant si une valeur est dans un tableau.

Exercice 7 : Écrire une fonction qui prend en entrée deux références et qui échange leur contenu. Attention, il y a un piège, donc mieux vaut s'assurer que la fonction marche. Anticiper la signature avant de la consulter.

Exercice 8 : Écrire une version récursive de `print_int_array`.

TP 6 : Types construits en OCaml

L'un des domaines où beaucoup de difficultés sont rencontrées (au vu des copies de MPSI et MP pendant sept ans...) est la gestion de types construits.

La syntaxe est effectivement assez exotique, et la rigueur habituelle du langage OCaml peut faire aisément refuser des programmes approximatifs.

Nous allons commencer par un type somme avec lequel il est possible de simuler des nombres (similaire au type `num` existant), défini ainsi :

```
type nombre = Entier of int | Flottant of float
| Fraction of int * int | Moins_inf | Plus_inf;;
```

avec les notations intuitives.

On peut donc manipuler les nouveaux objets ainsi introduits :

```
Entier(42);; (* reconnu comme de type nombre *)
Fraction(19, 12);; (* aussi *)
Entier(4.);; (* erreur de typage *)
Fraction(3);; (* aussi *)
Flottant;; (* ceci n'est en fait pas une fonction,
il y a une erreur de syntaxe *)
Fraction(1, 0);; (* accepté, il n'y a pas, juste avec la syntaxe,
de raison de s'alarmer *)
Entier(2) < Entier(4);; (* vrai : même constructeur
et comparaison des paramètres *)
Entier(2) > Moins_inf;; (* vrai, mais incontrôlable *)
Entier(2) > Plus_inf;; (* la preuve ! *)
Entier(2) < Flottant(1.2);; (* vrai aussi, tant qu'à faire *)
Fraction(2, 19) < Fraction(1, 4);; (* faux (ordre lexicographique) *)
Fraction(2, 8) = Fraction(1, 4);; (* faux, même argument de syntaxe *)
Entier(4) + Entier(5);; (* erreur de typage, même remarque *)
Entier(4 + 5);; (* là, ça passe, on évalue d'abord 4 + 5 *)
```

Exercice 1 : Écrire une fonction qui prend en entrée un nombre du type ci-avant et qui détermine s'il est cohérent (donc pas une fraction avec un dénominateur nul). Pour aller plus loin, on peut aussi vérifier que la fraction est simplifiée et que son dénominateur est strictement positif.

Exercice 2 : Écrire une fonction qui prend en entrée deux nombres du type ci-avant et qui détermine s'ils sont égaux.

Exercice 3 : Écrire une fonction qui prend en entrée deux nombres du type ci-avant et qui détermine si le premier est inférieur au deuxième.

Exercice 4 : Écrire une fonction qui prend en entrée deux nombres du type ci-avant et qui calcule leur somme, puis de même pour le produit. Une forme indéterminée déclenchera une erreur.

Ensuite, un type enregistrement utilisant également deux types annexes (ces types pouvant être remplacés par les types entier, pour avoir un identifiant, ou chaîne de caractères, pour stocker leur nom) :

```
type valeur = Sept | Huit | Neuf | Dix | Valet | Dame | Roi | As;;
type couleur = Trefle | Pique | Coeur | Carreau;;

type carte = { va : valeur; coul : couleur };;
(* le mot-clé val est réservé *)
```

Une main sera considérée comme un tableau de cartes (pas de type spécifique).

De même qu'avant, quelques manipulations :

```
let neuf_de_carreau = { va = Neuf; coul = Carreau };;
(* reconnu comme de type carte *)
let huit_de_trefle = { coul = Trefle; va = Huit };;
(* l'ordre n'importe pas *)
let main_de_deux_cartes = [| neuf_de_carreau; huit_de_trefle |];;
neuf_de_carreau.va;; (* Neuf *)
huit_de_trefle.coul;; (* Trefle *)
huit_de_trefle < neuf_de_carreau;;
(* vrai, ordre d'apparition des constructeurs sans doute *)
neuf_de_carreau.(0);; (* erreur de typage *)
{ Valet ; Trefle };; (* erreur de syntaxe *)
(As, Coeur);; (* pas reconnu comme de type carte,
mais comme un valeur * couleur qui sera incompatible *)
```

Exercice 5 : Écrire une fonction qui prend en entrée une main et qui détermine le nombre de cartes par couleur, en tant que tableau d'entiers dont les indices correspondent à l'ordre dans lequel les couleurs sont données dans la création du type.

Exercice 6 : Écrire une fonction qui prend en entrée une main et qui détermine le nombre de points dans cette main, au sens de la fonction ci-après.

```
let points_carte carte = match carte.va with
| As -> 11
| Dix -> 10
| Roi -> 4
| Dame -> 3
| Valet -> 2
| _ -> 0;;
```

Exercice 7 : Écrire une fonction qui prend en entrée une main et qui détermine le nombre de cartes par valeur, en tant que liste ou tableau dont les valeurs sont ordonnées (s'il s'agit d'une liste, les plus grandes valeurs devront être en tête).

Exercice 8 : Écrire une fonction qui prend en entrée une main et qui détermine s'il existe cinq cartes dont les valeurs se suivent dans l'ordre de leur description.

Exercice 9 (à faire chez soi) : Écrire une fonction qui prend en entrée une main et qui détermine la plus forte combinaison de poker réalisée en prenant cinq cartes de cette main. On pourra étendre le type valeur pour qu'il y ait treize éléments.

TP 7 : Implémentation de structures de données

Ce TP, à combiner avec le suivant, complète la partie du cours du chapitre 8 traitant du programme du premier semestre, en faisant écrire tous les programmes présentés dans les notes de cours. Les implémentations qui y sont écrites (ou la correction, le cas échéant) permettront d'écrire des programmes où les structures de données en question seront utiles au TP 8, en fait, d'où l'intérêt de bien sauvegarder son travail.

Sur l'ensemble de ces deux TP, les notions d'interface et de barrière d'abstraction pourront être illustrées.

Au vu du travail à fournir, il est recommandé de ne faire pendant les deux heures qu'une implémentation par section. Une deuxième séance est prévue pour faire le reste du travail.

Tableaux redimensionnables

Puisque les tableaux redimensionnables seront utilisés pour les implémentations suivantes, on va commencer par cette structure.

Implémentation en C avec un `struct`

Exercice 1 : Implémenter en C la structure de tableau redimensionnable d'entiers, dont l'interface est donnée ci-après.

```
struct t_r_i { int taille; int capacite; int* donnees; };
typedef struct t_r_i itr;
itr creer_itr(int capacite);
int taille_itr(itr t);
int acces_itr(itr t, int i);
void modif_itr(itr t, int i, int x);
void append_itr(itr* t, int x);
int pop_itr(itr* t);
```

On note que la taille n'est pas mise en argument car la structure elle-même contient l'information associée. De plus, puisqu'une mutation de l'argument est nécessaire dans certains cas, en se rappelant le cours du chapitre C associé, il faudra parfois utiliser des pointeurs, justifiant l'apparition d'étoiles.

Implémentation en OCaml avec un type enregistrement

Exercice 2 : Implémenter en OCaml la structure de tableau redimensionnable, dont l'interface est donnée ci-après.

```
type 'a tr = { mutable taille : int; mutable donnees : 'a array };;
val creer_tr : int -> 'a -> 'a tr;;
val taille_tr : 'a tr -> int;;
val acces_tr : 'a tr -> int -> 'a;;
val modif_tr : 'a tr -> int -> 'a -> unit;;
val append_tr : 'a tr -> 'a -> unit;;
val pop_tr : 'a tr -> 'a;;
```

Listes chaînées

La structure de liste chaînée pouvant être persistante ou modifiable, il y a deux sortes d'implémentations bien différentes en OCaml.

Implémentation en C avec un `struct`

Le choix des opérations élémentaires, parmi toutes les possibilités, sera de permettre :
- de créer une liste chaînée :
- d'obtenir un pointeur vers la tête d'une liste chaînée :
- d'avancer à l'élément suivant un élément d'une liste chaînée ;
- d'insérer un élément avant un élément d'une liste chaînée (voire à la fin) ;
- de retirer un élément d'une liste chaînée (sans le renvoyer) ;
- de consulter la valeur d'un élément d'une liste chaînée ;
- de modifier la valeur d'un élément d'une liste chaînée.

Pour varier les possibilités d'arguments, on ajoute des fonctions plus pratiques (notamment en vue d'utiliser les listes chaînées dans le cadre de l'implémentation d'autres structures) :
- insérer un élément en tête d'une liste chaînée ;
- retirer l'élément de tête d'une liste chaînée (en le renvoyant, pour le coup) ;
- insérer un élément au fond d'une liste chaînée ;
- retirer l'élément du fond d'une liste chaînée (en le renvoyant aussi) ;

Exercice 3 : Implémenter en C la structure de liste chaînée d'entiers, dont l'interface est donnée ci-après.

```
struct m_l_c_i { int valeur; struct m_l_c_i* suivant; };
typedef struct m_l_c_i milc;
struct l_c_i { milc* debut; };
typedef struct l_c_i ilc;
ilc creer_ilc();
milc* tete_ilc(ilc l);
milc* suivant_ilc(milc m);
void inserer_ilc(ilc* l, milc* m, int x);
void retirer_ilc(ilc* l, milc* m);
int acceder_ilc(milc m);
void modifier_ilc(milc* m, int x);
void inserer_tete_ilc(ilc* l, int x);
int retirer_tete_ilc(ilc* l);
void inserer_fond_ilc(ilc* l, int x);
int retirer_fond_ilc(ilc* l);
```

Implémentation en C avec un tableau redimensionnable

Exercice 4 : Implémenter en C la structure de liste chaînée d'entiers, avec le prototype suivant pour les opérations (légèrement différentes).

```
itr creer_ilc_itr(int estimation_capacite);
void inserer_ilc_itr(itr* t, int indice, int x);
void retirer_ilc_itr(itr* t, int indice);
int acceder_ilc_itr(itr t, int indice);
void modifier_ilc_itr(itr t, int indice, int x);
void inserer_tete_ilc_itr(itr* t, int x);
int retirer_tete_ilc_itr(itr* t);
void inserer_fond_ilc_itr(itr* t, int x);
int retirer_fond_ilc_itr(itr* t);
```

L'idée d'utiliser un tableau redimensionnable aurait pu être reprise en OCaml, mais pour gagner du temps et éviter la redondance cela ne sera pas fait dans ce TP.

Implémentation en OCaml avec un type enregistrement

Les opérations élémentaires changent un peu, elles permettent cette fois :

- de créer une liste chaînée :
- d'obtenir la taille d'une liste chaînée ;
- d'accéder à la valeur au maillon « d'indice `i` » d'une liste chaînée (indice valide) ;
- d'insérer un élément au maillon « d'indice `i` » d'une liste chaînée (idem mais i peut valoir la taille) ;
- de retirer le maillon « d'indice `i` » d'une liste chaînée (idem) ;

Exercice 5 : Implémenter en OCaml la structure de liste chaînée, dont l'interface est donnée ci-après.

```
type 'a lc = { mutable tete : 'a option;
    mutable suivant : 'a lc option };;
val creer_lc : unit -> 'a lc;;
val taille_lc : 'a lc -> int;;
val acces_lc : 'a lc -> int -> 'a;;
val inserer_lc : 'a lc -> int -> 'a -> unit;;
val retirer_lc : 'a lc -> int -> unit;;
```

Implémentation en OCaml avec une liste

On notera que cette structure est persistante. Les signatures changent vu que la mutation est impossible et remplacée par le renvoi de la liste résultant de l'opération.

Exercice 6 : Implémenter en OCaml la structure de liste chaînée, avec les fonctions dont les signatures sont données ci-après.

```
val creer_lcp : unit -> 'a list;;
val taille_lcp : 'a list -> int;;
val acces_lcp : 'a list -> int -> 'a;;
val inserer_lcp : 'a list -> int -> 'a -> 'a list;;
val retirer_lcp : 'a list -> int -> 'a list;;
```

Piles

La structure de pile a plusieurs implémentations possibles. Celle qui se contente de reprendre le type des listes en OCaml ne sera pas traitée ici... On aura donc deux structures modifiables, toutes deux écrites en C.

Implémentation en C avec un tableau

Dans cette implémentation, la pile a une capacité limitée (on n'utilise pas de tableau redimensionnable), à savoir la taille du tableau d'appui moins un, car la première position du tableau sert à indiquer le nombre d'éléments de la pile. On pourrait imaginer une position de plus pour indiquer la capacité, par ailleurs.

Exercice 7 : Implémenter en C la structure de pile bornée d'entiers, avec le prototype suivant pour les opérations.

```
int* creer_pileb(int capacite);
void empiler_pileb(int* p, int capacite, int x);
int depiler_pileb(int* p);
bool est_vide_pileb(int* p);
```

Implémentation en C avec une liste chaînée

Comme la liste chaînée offre plus de possibilités que la pile, le travail devrait être facile.

Exercice 8 : Implémenter en C la structure de pile d'entiers, avec le prototype suivant pour les opérations.

```
ilc creer_pile();
void empiler_pile(ilc* p, int x);
int depiler_pile(ilc* p);
bool est_vide_pile(ilc p);
```

Files

Pour rétablir l'équilibre, les deux implémentations de la file se feront en OCaml.

Implémentation en OCaml avec un tableau

De manière analogue à l'implémentation très similaire des piles, la capacité des files sera ici limitée. Pour permettre le polymorphisme, le tableau sera accompagné d'informations en-dehors, le tout sera alors rassemblé dans un type enregistrement.

La redondance des informations en question facilitera la programmation.

Exercice 9 : Implémenter en OCaml la structure de file bornée, dont l'interface est donnée ci-après.

```
type 'a fileb = { mutable taille : int; mutable tete : int;
    mutable queue : int; donnees : 'a array };;
val creer_fileb : int -> 'a -> 'a fileb;;
val enfiler_fileb : 'a fileb -> 'a -> unit;;
val defiler_fileb : 'a fileb -> 'a;;
val est_vide_fileb : 'a fileb -> bool;;
```

Implémentation en OCaml avec deux piles

Les piles pouvant être implémentées comme on le souhaite, la structure ainsi obtenue sera persistante ou non.

Ici, en l'absence d'implémentation explicite d'une pile en OCaml dans ce TP, on utilisera deux listes, d'où une structure persistante.

Exercice 10 : Implémenter en OCaml la structure de file, avec les fonctions dont les signatures sont données ci-après.

```
val creer_file : unit -> 'a list * 'a list;;
val enfiler_file : 'a list * 'a list -> 'a -> 'a list * 'a list;;
val defiler_file : 'a list * 'a list -> 'a * ('a list * 'a list);;
val est_vide_file : 'a list * 'a list -> bool;;
```

TP 8 : Applications des piles, tables de hachage

Ce TP, à la suite du précédent, nécessite de réutiliser les implémentations qui y ont été faites. L'implémentation retenue peut changer d'un exercice à l'autre, en fonction des besoins.

Applications des piles

Analyse d'expressions bien parenthésées

Nous nous intéressons à des expressions mathématiques utilisant des parenthèses, avec la question de déterminer si les parenthèses ne provoquent pas d'erreur de syntaxe. On parle ici de *mots bien parenthésés*. Le sens de « mot » est à rapprocher de notions vues en deuxième année.

Dans la mesure où tout ce qui n'est pas une parenthèse n'a pas de pertinence dans cette étude, les programmes ignoreront tous les caractères inutiles mais pouvant être présents dans les arguments, il reste pour ainsi dire juste la suite de parenthèses comme un mot avec deux lettres possibles.

On pourra commencer à chercher une condition nécessaire et suffisante pour qu'un mot soit bien parenthésé.

Un exercice de mathématiques de niveau SPÉ (au moins) demande de prouver que le nombre de mots bien parenthésés de taille $2n$ est $\frac{1}{n+1}\binom{2n}{n}$, soit le n-ième nombre de Catalan.

Exercice 1 : Écrire une fonction en C qui vérifie si une chaîne de caractères correspond à un mot bien parenthésé au regard de `'('` et `')'`. En pratique, l'utilisation des piles n'est pas obligatoire ici.

Pour l'exercice suivant, l'implémentation des piles n'ayant pas été faite en OCaml au TP précédent, il faut commencer par compenser ce manque...

Exercice 2 : Écrire une fonction en OCaml qui, étant donné un mot bien parenthésé, retourne la liste des couples `i,j` représentant les indices (en commençant à 0) des couples de parenthèses suivant le parenthésage. L'ordre d'apparition des couples dans la liste n'est pas important.

On considère maintenant un nombre arbitraire de parenthèses différentes, qui auront un identifiant entier strictement positif. Les parenthèses ouvrantes seront marquées par l'identifiant et les parenthèses fermantes par l'opposé de l'identifiant. Une expression sera alors un tableau d'entiers, dont on assimile toute valeur nulle à autre chose qu'une parenthèse (à ignorer, pour rappel).

Exercice 3 : Écrire une fonction en C qui détermine si un tableau d'entiers, à interpréter avec la syntaxe précédente et fourni avec sa taille, correspond à une expression bien parenthésée.

Notation polonaise inversée

Une expression en notation polonaise inversée a le bon goût de ne pas nécessiter de parenthèses. La notation est postfixe, dans la mesure où l'opérateur est situé après ses opérandes, de sorte que par exemple `(2 + 3) x 5` s'écrira `2 3 + 5 x`.

En fait, rencontrer un opérateur dans la lecture de l'expression fait chercher les valeurs (opérandes ou résultats d'opérations) les plus récentes et applique l'opération.

Le nombre d'opérandes d'un opérateur étant important, il faut alors utiliser deux symboles - différents (ce qui ne nous concernera pas dans l'exercice à suivre, cependant, pour simplifier) : un pour le signe (s'appliquant à un opérande) et un pour la soustraction (s'appliquant à deux opérandes) [2].

Exercice 4 : Écrire une fonction en OCaml qui évalue une expression arithmétique qui est donnée en notation polonaise inversée, par exemple sous la forme d'une liste de chaînes de caractères. L'expression utilisera seulement des entiers ou des flottants entourés de guillemets et les opérateurs usuels sur ces nombres.

Parcours de labyrinthe

Pour sortir d'un labyrinthe dit parfait, une méthode simple est la méthode de la main gauche : on suit un chemin en maintenant constamment sa main gauche contre un mur. [3]

2. De nombreuses calculatrices ont utilisé cette notation, ce qui explique l'emploi historique des deux boutons.

3. Cette méthode consiste simplement à faire un parcours en profondeur, comme on le verra dans un chapitre ultérieur.

Les labyrinthes que nous considérons ici sont des matrices dont les cellules contiennent une information sur 4 bits, indiquant la présence ou non d'un mur en haut, en bas, à gauche et à droite. Il est essentiel que les informations soient cohérentes d'une cellule à l'autre et que les bords de la matrice indiquent des murs vers les cellules inexistantes.

Exercice 5 : Écrire une fonction en OCaml qui vérifie si une matrice correspond à un labyrinthe valide.

Nous allons implémenter un algorithme équivalent, consistant, à chaque position (en pratique à chaque intersection), à :

- empiler la position courante et mémoriser la position depuis laquelle on y est arrivé, si on la visite pour la première fois ;
- tester successivement les sorties possibles de la position courante dans le sens des aiguilles d'une montre à partir du point cardinal d'où on est arrivé dans la position.

Exercice 6 : Écrire une fonction en OCaml correspondante.

Pour information, il est possible de générer un labyrinthe parfait, c'est-à-dire dans lequel il existe un et un seul chemin de n'importe quelle position à n'importe quelle position. Une structure de données au programme de deuxième année est particulièrement adaptée à sa conception.

Tables de hachage en C

Dans cette section, nous allons implémenter la structure de table de hachage. La structure d'appui sera un tableau non redimensionnable (pour simplifier), mais la capacité n'en sera pour autant pas limitée car chaque élément du tableau sera une liste chaînée.

Exercice 7 : Écrire le type correspondant, après avoir choisi une structure au choix pour les listes chaînées, de préférence la version qui ne passe pas par les tableaux redimensionnables, et repris depuis le TP précédent l'implémentation associée.

Exercice 8 : Écrire les opérations élémentaires sur la structure associée. Le choix est laissé libre pour les cas où plusieurs possibilités de spécifications existent, et l'idéal est d'anticiper sur les exercices de la section suivante.

Applications des tables de hachage

Les exercices de cette section sont à faire en C et en OCaml. En ce qui concerne OCaml, le module `Hashtbl` peut être utilisé.

Exercice 9 : Écrire une fonction qui prend en argument une chaîne de caractères et qui détermine le caractère le plus fréquent.

Exercice 10 : Écrire une fonction qui prend en argument deux listes d'entiers en OCaml / deux tableaux d'entiers et leur taille respective en C et qui détermine si elles / s'ils contiennent les mêmes éléments, chacun avec le même nombre d'occurrences.

Exercice 11 : Écrire une fonction prenant en argument un entier entre `1` et `3999` et qui retourne son écriture en chiffres romains en tant que chaîne de caractères. Écrire aussi sa réciproque, en vérifiant à un moment que l'argument est cohérent.

Exercice 12 : Écrire une fonction prenant en argument une chaîne de caractères formée des lettres `A`, `T`, `C` et `G`, représentant une portion d'ADN, et qui construit successivement sa transcription en ARN messager (en tant que chaîne de caractères) puis la protéine obtenue (en tant que chaîne de caractères), sachant que si aucune protéine ne peut être obtenue la chaîne retournée sera vide.

Petit rappel de biologie, avec des abus et erreurs éventuelles justifiés par l'algorithmique à travailler pour cet exercice :

- La transcription de l'ADN en ARNm revient à substituer les bases azotées par leur complémentaire, selon la règle `A` $\rightarrow$ `U`, `T` $\rightarrow$ `A`, `C` $\rightarrow$ `G` et `G` $\rightarrow$ `C`.
- La construction de la protéine revient à étudier des séquences de trois bases azotées de l'ARNm, en commençant par détecter ce qu'on appelle un codon initiateur (une séquence référencée, unique chez les eucaryotes, ici aussi), au-delà duquel toutes les séquences de trois bases azotées, jusqu'à une séquence particulière appelée codon stop (il y en a plusieurs) seront traduites en un acide aminé en fonction d'un dictionnaire donnant les correspondances, ce dictionnaire pouvant être une variable globale et contenant également l'information des codons initiateurs et stop d'une manière ou d'une autre. Le codon stop, comme le codon initiateur, ne sont pas inclus dans la protéine.
- Remarque importante : il ne faut pas lire trois par trois les bases azotées de l'ARN messager dès le début, mais chercher la première occurrence d'un codon initiateur.

Pour la création du tableau associatif principal de l'exercice, on pourra se servir des fichiers `http://jdreichert.fr/Enseignement/CPGE/MP2I/tp8_base.c` en C et `http://jdreichert.fr/Enseignement/CPGE/MP2I/tp8_base.ml` en OCaml. La syntaxe dépend des noms de fonctions choisis au moment de l'implémentation...

TP 9 : Arbres binaires et parcours d'arbres

Dans ce TP, nous allons faire dans un premier temps des calculs sommaires sur les arbres binaires, afin de prendre les structures associées en main, puis réaliser des parcours d'arbres, binaires ou non cette fois.

Arbres binaires

Pour simplifier, les étiquettes seront simplement des entiers.

Les types employés dans cette section seront alors

```
type arbin = Vide | Noeud of arbin * int * arbin
```

en OCaml et

```
struct i_a_b { int etiquette; struct i_a_b* fg; struct i_a_b* fd; }
```

en C, suivi par ailleurs de

```
typedef struct i_a_b abi
```

pour le confort. On notera qu'avec cette structure les arbres vides n'existent pas en C, contrairement au choix qu'on avait fait pour les listes chaînées. Il est tout de même imaginable de distinguer la structure de nœud (celle qui est écrite ici en fait) et la structure d'arbre qui serait alors définie vers le pointeur vers la racine.

Chacun des exercices est à faire dans les deux langages, mais il est recommandé de faire pendant la séance les exercices dans un seul langage en alternant occasionnellement. L'ensemble des exercices n'est pas réalisable en deux heures, le TP sera à compléter après.

Exercice 1 : Écrire une fonction qui calcule la taille d'un arbre binaire.

Exercice 2 : Écrire une fonction qui calcule la somme des étiquettes d'un arbre binaire.

Exercice 3 : Écrire une fonction qui calcule la plus petite étiquette d'un arbre binaire.

Exercice 4 : Écrire une fonction qui calcule la hauteur d'un arbre binaire.

Exercice 5 : Écrire une fonction qui calcule le nombre d'arbres binaires de taille `n` de formes différentes (donc autant dire que toutes les étiquettes valent zéro) [4].

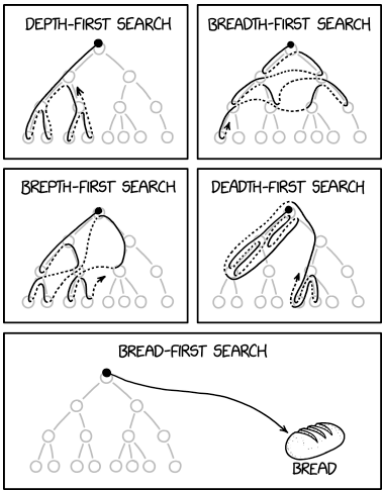
Pour ce calcul, la programmation dynamique s'impose, car la réponse ne doit pas s'obtenir en appliquant naïvement la formule de récursion sous peine d'avoir une complexité pire qu'exponentielle.

Il s'agit de stocker dans un tableau de taille `n+1` toutes les réponses calculées de gauche à droite avec une formule « de récursivité forte » qui utilise les valeurs déjà mémorisées plutôt que des appels récursifs.

Exercice 6 : Écrire une fonction qui engendre toutes les formes d'arbres binaires de taille `n`. On évitera de faire le test pour des valeurs de `n` dépassant `10`, pour des raisons d'espace mémoire (voir la question précédente...). On les stockera dans un tableau en C (ou une liste en OCaml), avec a priori un tableau de tableaux/listes intermédiaire.

Parcours d'arbres

(Crédit pour l'image : Randall Munroe, `http://xkcd.com/2407`)



4. Un exercice de combinatoire revient à donner la formule.

Nous allons utiliser des types sans feuille ici, et les définitions suivantes, d'abord en OCaml :

```
type 'a arbre = Vide | Noeud of 'a * 'a arbre list;;
type 'a arbre_bin = V | N of 'a arbre_bin * 'a * 'a arbre_bin;;
(* Rappel : un constructeur ne peut être utilisé qu'une fois. *)
```

Puis en C, en reprenant également le type `abi` de la section précédente :

```
struct i_a { int etiq; int nb_fils; struct i_a* fils; };
typedef struct i_a ai;
```

Exercice 7 : Écrire une fonction dans chaque langage et pour chacune des trois façons et sous-façons de parcours pour les arbres d'arité quelconque, en imprimant chaque nœud au moment où sa visite doit se faire.

Exercice 8 : Écrire une fonction dans chaque langage et pour chacune des quatre façons et sous-façons de parcours pour les arbres binaires, avec les mêmes conditions.

Exercice 9 : Reprendre les deux premiers exercices, et cette fois-ci il faut retourner le tableau des étiquettes rencontrées dans l'ordre. On envisagera d'écrire d'abord une fonction pour calculer la taille du tableau à créer.

Exercice 10 : Reprendre les deux premiers exercices, et cette fois-ci il faut retourner la chaîne de caractères obtenue en concaténant les étiquettes, dont on supposera en OCaml qu'il s'agira déjà de chaînes de caractères, avec une espace entre chaque étiquette.

Pour tester les programmes de l'exercice 8, voici des exemples d'arbres binaires. Le parcours adéquat imprime à chaque fois les nombres de `1` à `7`.

```
let arbre_prefixe = (N(N(N(V,3,V),2,N(V,4,V)),1,N(N(V,6,V),5,N(V,7,V))));;
let arbre_infixe = (N(N(N(V,1,V),2,N(V,3,V)),4,N(N(V,5,V),6,N(V,7,V))));;
let arbre_postfixe = (N(N(N(V,1,V),3,N(V,2,V)),7,N(N(V,4,V),6,N(V,5,V))));;
let arbre_largeur = (N(N(N(V,4,V),2,N(V,5,V)),1,N(N(V,6,V),3,N(V,7,V))));;
```

En remplaçant les constructeurs, on peut tester les fonctions de la première section.

Un arbre défini en C et équivalent à `arbre_prefixe` est `a1`, défini ci-après :

```
abi a3 = { .etiquette = 3, .fg = NULL, .fd = NULL };
abi a4 = { .etiquette = 4, .fg = NULL, .fd = NULL };
abi a6 = { .etiquette = 6, .fg = NULL, .fd = NULL };
abi a7 = { .etiquette = 7, .fg = NULL, .fd = NULL };
abi a2 = { .etiquette = 2, .fg = &a3, .fd = &a4 };
abi a5 = { .etiquette = 5, .fg = &a6, .fd = &a7 };
abi a1 = { .etiquette = 1, .fg = &a2, .fd = &a5 };
```

TP 10 : ABR, arbres bicolores et tas

Exceptionnellement, ce TP est à préparer avant la séance afin de pouvoir se concentrer sur la partie difficile une fois les structures prêtes.

Implémentations préliminaires

En OCaml, on utilisera pour toutes les structures le type suivant :

```
type 'a abr = Vide | Noeud of 'a abr * 'a * 'a abr;;
```

En C, on implémentera la structure classique avec des pointeurs pour les ABR. Pour implémenter des tas, on considèrera des arbres binaires presque complets, stockés dans des tableaux selon le principe du cours (en raison des opérations élémentaires effectuées on utilisera des tableaux redimensionnables). En vue de l'application des arbres utilisés, dans ce langage, les nœuds des ABR seront des structures avec un champ de type chaîne de caractères pour la clé et un champ de type entier pour la valeur. Quant aux éléments du tableau support pour les tas, il s'agira de structures avec les types « permutés » : chaîne de caractères pour la valeur à stocker et entier pour le critère de classement (la priorité).

Exercice 1 : Écrire les quatre opérations élémentaires pour les ABR en OCaml, en gardant une complexité asymptotiquement optimale. Pour la suppression, il est recommandé de créer aussi une fonction `minimum`.

Les opérations élémentaires sont la création, la recherche, l'insertion et la suppression. En OCaml, la structure est persistante, comme on l'aura remarqué.

Exercice 2 : Écrire de même les quatre opérations élémentaires pour les ABR en C.

Exercice 3 : Écrire les trois opérations élémentaires pour les tas-min sans modification de clé en OCaml.

Les opérations élémentaires sont la création, l'extraction du minimum et l'insertion.

Exercice 4 : Écrire de même les quatre opérations élémentaires pour les tas-min binaires presque complets avec modification de clé en C.

On ajoute aux trois opérations susmentionnées l'opération de modification d'une clé, en fournissant l'indice où elle se situe et la nouvelle clé qui la remplace.

Pour qu'un arbre binaire reste presque complet, l'insertion doit se faire à la fin du tableau, nécessitant de procéder à des échanges en remontant une branche pour assurer que l'on ait encore un tas.

De même, la suppression peut concerner n'importe quel nœud, mais ceci suppose un échange avec le dernier élément et là aussi la correction au niveau de la branche. Ceci étant, on ne supprimera que la racine en implémentant les opérations élémentaires.

On pourra se demander pourquoi on n'envisage pas de structure d'ABR presque complet...

Réalisation d'un dictionnaire

On ne fera pas plus d'un des deux exercices ci-après pendant la séance.

Exercice 5 : Implémenter la structure de dictionnaire en OCaml à l'aide d'un ABR.

Exercice 6 : L'implémenter aussi en C à l'aide d'un ABR.

Réalisation d'une file de priorité

Même remarque qu'à la section précédente. On n'hésitera pas à utiliser l'autre langage. La modification de la priorité ne sera une opération élémentaire qu'en C.

Exercice 7 : Implémenter la structure de file de priorité en OCaml à l'aide d'un tas.

Exercice 8 : L'implémenter aussi en C à l'aide d'un tas binaire presque complet.

Arbres bicolores

Nous arrivons au point crucial de ce TP, qui aborde les opérations élémentaires sur les arbres rouge-noir préservant l'invariant de structure associé.

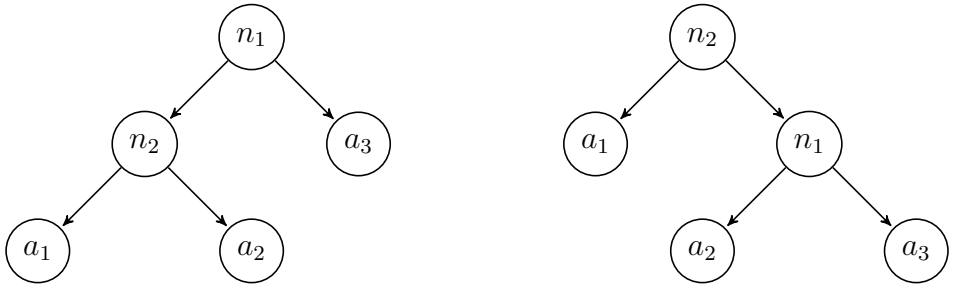
Tous les programmes de cette section seront en OCaml uniquement.

Commençons par décrire les opérations de rotation.

Sans tenir compte de la couleur, les rotations gauche et droite réorganisent un arbre binaire (ou un sous-arbre d'un arbre binaire) en remplaçant la racine (ou un nœud) par son fils droit ou gauche, respectivement, et en réassignant les fils du fils en question ainsi que l'autre fils de la racine (ou du nœud considéré) pour conserver la propriété d'ABR lorsqu'elle est vérifiée avant la rotation.

La hauteur d'un arbre peut alors changer d'un cran, ce qui tombe bien vu l'utilisation qu'on aura besoin de faire des rotations.

Concrètement, la rotation droite permet de passer de l'arbre de gauche à l'arbre de droite dans la figure ci-après, et la rotation gauche est la réciproque de la rotation droite. On notera que ces opérations sont censées échouer si l'arbre (ou le sous-arbre considéré) n'a pas au moins une racine avec un fils adéquat.



On a toutes les clés de a_1 inférieures à la clé de n_2, elle-même inférieure à toutes les clés de a_2, elles-mêmes inférieures à la clé de n_1, elle-même inférieure à toutes les clés de a_3, ceci dans les deux arbres représentés. Une rotation transforme donc bien un ABR en un ABR.

Exercice 9 : Écrire une fonction `rotation_droite` et une fonction `rotation_gauche` prenant en argument un arbre binaire (les propriétés additionnelles n'interviennent pas ici) et renvoyant l'arbre binaire obtenu par la rotation en question.

La hauteur noire d'un arbre ayant subi une rotation peut changer, donc ces rotations seront associées à des changements de couleur, comme nous allons le voir tout de suite.

Exercice 10 : Écrire une fonction `insertion_bicolore` prenant en argument un arbre rouge-noir, une clé à insérer et la valeur associée et renvoyant l'arbre rouge-noir obtenu en insérant un nœud dans cet arbre avec la clé et la valeur fournies.

La méthode suggérée pour insérer un nœud est la suivante :

- Parcourir une branche jusqu'à un arbre vide où l'insertion du nœud préserve la propriété d'ABR. Cet arbre vide est unique si (et seulement si) le nœud à insérer n'est pas déjà présent dans l'arbre. Sinon on prend celui qu'on veut.
- Créer un nœud rouge à la place de l'arbre vide, avec les informations qu'on souhaitait ajouter. Si le nœud est à la racine ou a pour père la racine ou un nœud noir, c'est terminé, sinon on passe à l'étape suivante.
- Tant que le nœud considéré, noté n, est le fils rouge d'un nœud rouge, noté p, qui n'est pas la racine donc a un père, noté g (et forcément noir, en accord avec la preuve à faire en TD), on fait un tour de la boucle de correction :
  - Si g a deux fils tous deux rouges, on met g en rouge et ses deux fils en noir, puis on considère g en vue de l'éventuel tour suivant de la boucle de correction. Sinon, on passe aux points suivants.
  - Si n est le fils gauche de p alors que p est le fils droit de g, on commence par une rotation droite sur le sous-arbre enraciné en p.
  - Inversement, si n est le fils droit de p alors que p est le fils gauche de g, on commence par une rotation gauche sur le sous-arbre enraciné en p.
  - Par la suite, si p (ou n après rotation) est le fils gauche de g, on fait une rotation droite sur le sous-arbre enraciné en g, sinon une rotation gauche, et dans tous les cas on met p (ou n après rotation) en noir et g en rouge et la boucle de correction est terminée. On remarquera qu'il est aussi possible de changer les trois couleurs de n, p et g, ce qui a deux conséquences : d'une part il y a moins de nœuds rouges, donc la hauteur est moindre (on peut envisager de la réduire de moitié ainsi), mais d'autre part la boucle de correction n'est pas forcément terminée. Pour autant, cette méthode n'est pas spécialement plus difficile à écrire et reste en complexité logarithmique dans le pire des cas, comme la méthode utilisée ici.

Suite à l'insertion, si la racine est rouge, on la met en noir parce que cela ne coûte rien et cela facilitera les opérations suivantes.

La suppression d'un nœud étant particulièrement ardue, elle ne sera pas donnée en exercice mais le code fourni dans la correction devra être testé une fois les autres opérations écrites.

Le principe est le suivant :

- On procède à une suppression comme dans un ABR, avec l'éventuel échange du nœud à supprimer avec le minimum du sous-arbre enraciné en ce nœud, les couleurs étant maintenues aux positions en question.
- La position où la suppression a lieu est désormais une position où au moins un fils est vide. La suppression a lieu et :
  - S'il y a un fils non vide (on rappelle que l'autre est alors vide), il est forcément rouge. Il prend alors la place du nœud supprimé, en devenant noir (car le nœud supprimé l'était forcément), puis la suppression est terminée.
  - S'il y a deux fils vides et que le nœud supprimé était rouge, c'est fini.
  - Sinon, le vide laissé fait baisser la hauteur noire de la branche en cours, ce qui peut être problématique. On va remonter cette branche en déplaçant le problème jusqu'à ce qu'il soit réglé, voir ci-après.

Parlons de la réparation de la modification de la hauteur noire d'une branche dans une nouvelle distinction de cas sur le nœud problématique, qui est initialement le vide laissé par la suppression et qu'on notera n :

- Si n est la racine, on considère que le problème est résolu, c'est l'arbre entier qui a perdu une unité de hauteur noire.
- Si n a un père, noté p, ayant un autre fils (forcément, en raison de la hauteur noire précédente) noté f, on étudie la couleur de f :
  - Si f est rouge, ce qui garantit que ses enfants éventuels sont tous noirs et que p l'est aussi, alors on fait une rotation gauche ou droite sur le sous-arbre enraciné en p suivant le fait que f soit le fils droit ou gauche de p, respectivement, et on échange les couleurs de f et p. On passe alors aux sous-cas suivants en adaptant les notations à la nouvelle situation. [5]
  - Si f est noir avec deux fils noirs, on met f en rouge et on déplace le problème à p dans un nouveau tour de boucle.
  - Sinon, si f est noir avec son fils droit noir, on met f en rouge, son fils gauche en noir, on fait une rotation droite dans le sous-arbre enraciné en f et on passe à l'étape suivante avec les notations adaptées.
  - Sinon, f est noir avec son fils droit rouge (peu importe son fils gauche), on fait alors une rotation gauche sur le sous-arbre enraciné en p. On met f à la couleur de p, on met p et le fils droit de f en noir, et le problème est résolu, ce qui permet là aussi de terminer la suppression.

5. **Pour ces sous-cas, on évitera d'alourdir les choses et on supposera que f est le fils droit de p. Dans le cas contraire, on changera toutes les occurrences de « droit(e) » par « gauche » et inversement, pour les fils comme pour les rotations.**

Tris

Exercice 11 : Écrire l'algorithme du tri ABR en OCaml en utilisant des arbres bicolores pour optimiser la complexité.

Exercice 12 : Écrire l'algorithme du tri par tas en C en profitant du fait que les arbres binaires presque complets soient équilibrés, de sorte que la complexité soit optimisée.

TP 11 : Des graphes en OCaml et en C

Le but de ce TP est d'implémenter les graphes en OCaml et en C, afin de programmer des algorithmes de base et de manipuler des représentations de graphes.

Reprendre la représentation graphique d'un graphe orienté revient à donner la liste des sommets et celle des arcs. C'est l'occasion de s'entraîner une fois de plus sur les types produits.

En OCaml, on écrira donc le type suivant, puis un exemple de graphe que l'on peut ainsi créer :

```
type graphe1 = { mutable sommets : string list;
mutable arcs : (string * string) list; };;

let mon_graphe1 = { sommets = ["a"; "b"; "c"; "d"];
arcs = [("a", "b"); ("a", "c"); ("b", "c"); ("c", "d"); ("d", "a")]; };;
```

Bien entendu, il serait tout aussi possible de se limiter aux arcs et donc de définir un graphe comme une liste de couples de chaînes de caractères. Récupérer la liste des sommets se ferait donc en parcourant la liste des arcs.

On se contente d'une redéfinition de type pour les deux autres représentations :

```
type graphe2 = (string * (string list)) list;;
type graphe3 = bool array array;;
```

En C, on utilisera également les trois représentations, avec les types suivants :

```
struct grapho { char* sommets; char* arcs; };
typedef struct grapho graphe_oriente;
```

Cette version est plus pratique que celle du cours, et nous imposera un encodage des sommets : les espaces et les tirets seront interdits dans les noms, de sorte que le découpage d'une seule chaîne de caractère suffise pour identifier les sommets. Quant aux arcs, ils seront délimités par des espaces et les deux noms de sommets seront séparés par un tiret.

Un exemple de graphe est alors (même exemple qu'en OCaml) :

```
char s[] = "a b c d";
char a[] = "a-b a-c b-c c-d d-a";
graphe_oriente g = { .sommets = s, .arcs = a };
```

Attention cependant, les chaînes correspondent normalement à des pointeurs qui ont été produits par `malloc`, pour la compatibilité avec les fonctions des exercices de 1 à 4. Il faudra donc potentiellement adapter cet exemple.

Comme il est important de maîtriser l'utilisation de diverses structures de données d'appui, le principe n'est pas repris dans la représentation par liste d'adjacence, mais on se servira de listes chaînées. Pour ne pas cumuler les difficultés, les sommets seront les premiers entiers naturels :

```
struct m_l_c_s { int sommet; struct m_l_c_s* suivant; };
typedef struct m_l_c_s mlcs;

struct l_c_s { mlcs* tete; };
typedef struct l_c_s lcs;

struct grapho_l_a { int nb_sommets; lcs* liste_adj; };
typedef struct grapho_l_a graphe_oriente_liste_adjacence;
```

Pour la matrice d'adjacence, ce sera un simple tableau de `n` tableaux de `n` booléens, la valeur de `n` étant précisée aussi :

```
struct grapho_m_a { int nb_sommets; bool** matrice_adj; };
typedef struct grapho_m_a graphe_oriente_matrice_adjacence;
```

Toutes les fonctions vont muter le graphe en argument sauf dans le cas de la deuxième représentation en OCaml et les fonctions d'ajout ou de retrait d'un sommet dans le cas de la troisième représentation en OCaml.

Exercice 1 : Écrire pour chaque représentation et dans chaque langage une fonction qui prend en entrée un graphe et qui lui ajoute un sommet, nommé `s` (argument supplémentaire) dans les représentations qui utilisent des chaînes de caractères. La fonction doit retourner une erreur si `s` existe déjà en tant que sommet.

On observe pour les exercices suivants l'intérêt d'avoir la liste des sommets dans la définition du graphe, afin de détecter l'existence en évitant le parcours d'une liste comportant un nombre potentiellement quadratique d'éléments par rapport au nombre de sommets.

Exercice 2 : Écrire pour chaque représentation une fonction qui prend en entrée un graphe et qui lui ajoute un arc entre deux sommets nommés `s` et `t` en argument (chaînes de caractères ou entiers suivant le cas). La fonction doit ajouter les sommets s'ils n'existent pas encore (pour les représentations sans chaînes de caractères, on pourra se contenter de déclencher une erreur si `s` et `t` ne sont pas entre `0` et le nombre de sommets moins un) mais déclencher une erreur si l'arc existe déjà.

Exercice 3 : Écrire pour chaque représentation une fonction qui prend en entrée un graphe et qui lui retire l'arc entre les sommets nommés `s` et `t` en argument. La fonction doit retourner une erreur si l'arc n'existe pas (à plus forte raison si les sommets n'existent pas).

Exercice 4 : Écrire pour chaque représentation une fonction qui prend en entrée un graphe et qui lui retire le sommet nommé `s`. La fonction doit retourner une erreur si `s` n'existe pas en tant que sommet.

Après ces exercices de mise en jambe sur la création et la suppression dans des graphes, nous allons passer à de vrais algorithmes.

Exercice 5 : Écrire dans les deux langages une fonction qui prend en entrée un graphe et deux sommets et qui détermine s'il existe un chemin du premier au deuxième sommet. La fonction doit retourner une erreur si l'un des sommets n'existe pas dans le graphe. La représentation est laissée libre.

En particulier, cet exercice et les suivants sont la seule occasion d'écrire un parcours. On s'efforcera d'utiliser au moins une fois un parcours en largeur et au moins une fois un parcours en profondeur.

Exercice 6 : Écrire dans les deux langages une fonction qui prend en entrée un graphe qu'on suppose non orienté et qui détermine si le graphe est connexe.

Exercice 7 : Même exercice sans supposer le graphe non orienté (on détermine donc si le graphe est fortement connexe).

Exercice 8 : Écrire dans les deux langages une fonction qui prend en entrée un graphe qu'on suppose non orienté et qui retourne la liste de ses composantes connexes.

Exercice 9 : Écrire dans les deux langages une fonction qui prend en entrée un graphe et un booléen indiquant si le graphe est à considérer comme un graphe orienté (donc vrai s'il est à considérer comme un graphe orienté) et qui retourne un chemin ou une chaîne eulérien(ne) s'il en existe un(e), ou alors une exception indiquant pourquoi il n'en existe pas.

Exercice 10 : Écrire dans les deux langages une fonction qui prend en entrée un graphe qu'on suppose non orienté et qui détermine s'il contient un cycle.

Exercice 11 : Écrire dans les deux langages une fonction qui prend en entrée un graphe et qui détermine s'il contient un circuit.

Exercice 12 : Écrire dans les deux langages une fonction qui prend en entrée un graphe orienté et qui retourne un tri topologique (ou une erreur si ce n'est pas possible).

Exercice 13 : Discuter de la complexité des fonctions écrites dans ce TP, ainsi que des différences entre représentations du point de vue de la complexité en espace.

Exercice 14 : Écrire enfin une fonction de sérialisation d'un graphe et la fonction de désérialisation associée.

TP 12 : Exploration exhaustive

Ce TP traite les trois exemples de situations où l'on peut utiliser des algorithmes de *backtracking* présentées dans le chapitre 14.

Le langage est au choix pour chaque section, en gardant le langage sur l'ensemble de la section en question.

Parcours en profondeur itéré

On commencera par créer un type au choix pour représenter un graphe. Ce type sera utilisé pour l'ensemble de la section.

Exercice 1 : Écrire une fonction de recherche de chemin par un parcours en profondeur avec comme arguments le graphe, le sommet de départ et le sommet à atteindre. La valeur de retour sera un chemin en cas d'existence, avec une erreur sinon.

Il faudra bien réfléchir à la manière de garantir la terminaison de la fonction.

Exercice 2 : Adapter la fonction précédente pour qu'un argument entier supplémentaire `n` fasse éliminer tous les chemins de longueur strictement supérieure à `n`.

La valeur de retour reste la même, mais l'erreur en cas de non existence d'un chemin sera éventuellement remplacée par une valeur bien distinguable d'un vrai chemin.

Exercice 3 : Écrire une fonction appelant la fonction précédente pour des valeurs croissantes de `n` jusqu'à trouver un chemin ou conclure sur la non existence d'un chemin. La valeur de retour sera de nouveau comme dans l'exercice 1.

Exercice supplémentaire : Prouver que le chemin retourné par la fonction de l'exercice précédent est effectivement optimal en longueur.

Exercice complémentaire : Commencer par écrire un parcours en profondeur itéré pour trouver la profondeur minimale où se trouve une étiquette dans un arbre, sans utiliser de représentation de graphe.

Perles de Dijkstra

Exercice 4 : Écrire une fonction prenant en argument un tableau (en C : d'entiers) et déterminant s'il est sans facteur carré (voir le problème des perles de Dijkstra).

Exercice 5 : Écrire une fonction prenant en argument un entier `n` et construisant la plus petite chaîne de caractères (selon l'ordre lexicographique) de taille `n` respectant les conditions du problème des perles de Dijkstra. On construira tous les tableaux de taille `n` comprenant des entiers de zéro à deux, successivement et dans l'ordre lexicographique, comme si on comptait en base 3. Le premier tableau qui convient sera converti en chaîne. On utilisera la fonction de l'exercice précédent.

Exercice 6 : Écrire une fonction prenant en argument un entier `n` et construisant la plus petite chaîne de caractères (selon l'ordre lexicographique) de taille `n` respectant les conditions du problème des perles de Dijkstra. Il s'agira de la remplir par retour sur trace sans se servir des exercices précédents.

Exercice supplémentaire : Calculer la complexité des trois fonctions de cette section.

Jeux de logique

Exercice 7 : Écrire une fonction sans argument permettant de remplir un carré magique normal de neuf cases. La méthode est au choix mais il faudra procéder à un moment par retour sur trace. En particulier, donner la solution à la main est exclu.

Un carré magique normal contient exactement un exemplaire de chacun des premiers entiers naturels non nuls et vérifie que la somme de toutes les lignes, de toutes les colonnes et des deux diagonales est toujours la même.

Exercice 8 : Écrire une fonction permettant de résoudre un sudoku. Même consigne.

La structure de données pour le sudoku est le tableau de tableaux d'entiers, le tout étant de taille neuf à chaque fois. La structure de données créée par la fonction de résolution, quant à elle, est laissée libre. Des suggestions pourront être données sur place.

Exercice 9 : Écrire une fonction permettant de résoudre un jeu des gratte-ciels. Même consigne.

Dans le jeu des gratte-ciels, il faut remplir une grille de `n` lignes par `n` colonnes avec les nombres de `1` à `n`, de sorte que chaque nombre apparaisse une fois et une seule dans chaque ligne et chaque colonne. Au début et à la fin de chaque ligne et chaque colonne, une indication précise le nombre de « gratte-ciels » visibles depuis le début / la fin de la ligne / colonne. Il s'agit de manière équivalente de la taille de la première (dans l'ordre lexicographique) sous-suite strictement croissante et non prolongeable de la suite des valeurs rencontrées de gauche à droite / de droite à gauche / de haut en bas / de bas en haut suivant le contexte.

TP 13 : Sérialisation, codage de Huffman

Dans ce TP, un premier exemple de sérialisation permettra de compléter l'optimisation de la compression de l'algorithme de Huffman en stockant également de manière aussi concise que possible les règles de compression, en tant qu'arbre ou autre structure, l'étude le dira...

On utilisera exclusivement le langage OCaml.

Algorithme de Huffman

La structure employée pour l'algorithme de Huffman est un arbre dont les feuilles sont étiquetées par des caractères et les nœuds internes ne sont pas étiquetés.

Le nombre d'occurrences est une information qui sera associée à toutes les parties en construction de l'arbre en tant que clés dans le tas-min qui permettra d'implémenter la file de priorité contenant ces parties en construction.

On utilisera donc les types suivants :

```
type arbre_huffman =
  Feuille of char | Noeud of arbre_huffman * arbre_huffman;;

type tas_huffman =
  V | N of (int * arbre_huffman) * tas_huffman * tas_huffman;;
```

Toutes les opérations élémentaires sur les arbres binaires et tas binaires peuvent être récupérées et adaptées.

Exercice 1 : ... le faire !

Exercice 2 : Écrire une fonction prenant en argument une chaîne de caractères et construisant une séquence (type au choix) de couples formés de chaque caractère et de son nombre d'occurrences. L'ordre n'a pas d'importance.

Exercice 3 : Écrire une fonction d'initialisation d'une file de priorité implémentée par un `tas_huffman` à partir de la séquence retournée par la fonction de la question précédente.

Exercice 4 : Écrire une fonction de mise à jour de la file de priorité de la fonction précédente en extrayant deux éléments de clés minimales et en les remplaçant par un nouvel élément obtenu en créant un nœud avec ces deux éléments comme fils, la clé étant la somme des clés des fils.

Exercice 5 : Écrire une fonction qui lance la fonction précédente dans une récursion jusqu'à arriver à un tas limité à sa racine, dont on retournera l'arbre en ignorant la clé.

Exercice 6 : Écrire une fonction qui prend en argument l'arbre ainsi obtenu et qui construit un dictionnaire donnant le code de chaque caractère qui y apparaît.

Exercice 7 : Écrire une fonction qui prend en argument le texte de départ et qui construit une chaîne de `'0'` et de `'1'` correspondant au codage de Huffman du texte. Renvoyer l'arbre de Huffman utilisé et le tableau de booléens correspondant à cette chaîne. On pourra aussi court-circuiter la construction de la chaîne mais la complexité doit être raisonnable dans ce cas.

Exercice 8 : Écrire une fonction qui prend en argument l'arbre de Huffman ainsi que le code en tant que tableau de booléens tels qu'obtenus par la fonction précédente et qui renvoie le texte de départ. Tester l'enchaînement des deux fonctions.

Sérialisation

Il s'agit désormais de stocker le résultat du codage de Huffman dans un fichier binaire. Il n'est pas question de consommer trop de place, par exemple en renseignant pour chaque caractère de l'arbre son code ASCII, puis le nombre de bits de son code de Huffman sur un octet, puis les bits en question en complétant l'octet pour simplifier.

La méthode classique revient à écrire l'arbre par un parcours en profondeur préfixe.

On commence à « donner le focus » à la racine. Pour signaler qu'on progresse dans l'arbre (donc en donnant le focus au nœud suivant dans le parcours préfixe, signalant par la même occasion que le nœud actuel était un nœud interne), on écrit par exemple un `'0'`. Pour signaler qu'on est sur une feuille, on écrit alors un `'1'` (rien n'empêche d'échanger ces deux chiffres, mais il faut que la sérialisation et la désérialisation s'accordent) suivi du code ASCII du caractère porté par la feuille, forcément sur huit bits. Ensuite on passe au nœud suivant automatiquement.

Il s'agit d'un encodage particulier (et non ambigu) : tout code est soit `'0'` soit un paquet de neuf bits démarrant par un `'1'`.

Une fois tous ces bits écrits (découpés en octets écrits en tant que le caractère ASCII correspondant), on enchaîne directement par l'écriture du texte codé.

Mais comment savoir où est la séparation ? En repensant aux propriétés d'un arbre binaire strict, dont l'arbre de Huffman est un exemple, À partir du moment où le nombre de codes `'1'` est strictement supérieur au nombre de codes `'0'`, c'est qu'il y a une feuille de plus que de nœuds internes, signalant que le parcours est terminé.

Exercice 9 : Écrire une fonction de sérialisation de l'arbre de Huffman, tout d'abord sous forme de tableau de booléens.

Terminons sur le souci de l'incomplétude du dernier octet avant de traiter la désérialisation de l'ensemble.

La méthode suggérée est de réserver les trois derniers bits pour une information signalant parmi les treize bits précédents combien sont à tenir en compte. Il s'agit de signaler qu'on en considère entre six et treize, en l'occurrence six de plus que la valeur brute des trois derniers bits. Suivant le cas, le dernier octet devient alors l'avant-dernier et on ajoute jusqu'à 7 bits à ignorer dont la valeur peut être arbitraire.

Une autre possibilité était de comptabiliser le zéro terminal du texte et de l'ajouter à l'arbre de Huffman. Rencontrer ce caractère dans la transcription du texte indique (en principe...) que le travail est fini. Cela coûte cependant a priori plus de place vu la taille du code de ce caractère qu'on ne rencontrera qu'une fois. Pour autant, on aurait une solution élégante au problème de la séparation entre l'arbre et le code en se débrouillant lors de la construction de l'arbre pour que le code du zéro terminal soit le plus grand dans l'ordre lexicographique. Rencontrer ce code signalerait qu'on enchaîne tout de suite avec le texte.

Exercice 10 : Incorporer cette méthode et écrire avec le résultat de la fonction précédente une fonction de sérialisation d'un arbre de Huffman et du texte encodé.

Exercice 11 : Écrire aussi la fonction de désérialisation associée pour l'ensemble des parties du texte encodé.

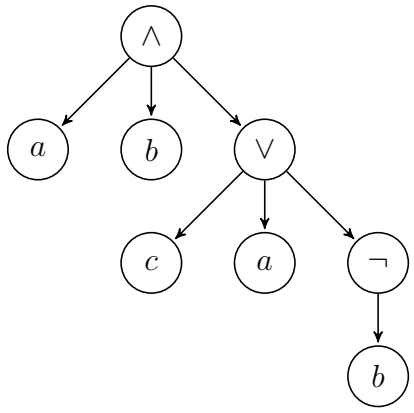
Exercice 12 : Utiliser des fonctions de lecture / écriture dans des fichiers pour encapsuler les fonctions précédentes dans des fonctions manipulant des fichiers.

Exercice 13 : Tester la fonction de sérialisation avec lecture dans un fichier sur des textes courts, moyens et très longs. Constater le taux de compression dans chaque cas.

TP 14 : Formules propositionnelles et arbres

Dans ce TP, dont tous les programmes seront une fois de plus en OCaml uniquement, nous allons représenter une formule de la logique propositionnelle par un arbre, dont les nœuds correspondront aux opérateurs et les feuilles représenteront les variables ou les constantes vrai et faux. En accord avec les chapitres sur la logique, on se limite aux opérateurs « ou », « et » et « non », mais l'arité des deux premiers ne sera pas limitée.

Par exemple, la formule a et b et (c ou a ou non b), dont le parenthésage a été allégé, s'écrit $a \wedge b \wedge (c \vee a \vee \neg b)$ et se représente ainsi :



Manipulations de base

On rappelle les priorités d'évaluation : d'abord les négations, puis les conjonctions (et), puis les disjonctions (ou).

Un type (somme) intuitif pour représenter en OCaml les formules booléennes sera :

```
type f_b = Vrai | Faux | Var of string | Non of f_b | Et of f_b list
| Ou of f_b list;;
```

et en se limitant à des opérateurs binaires on peut remplacer les listes par des couples [6].

6. Il est envisageable d'ajouter d'autres constructeurs pour introduire divers opérateurs.

En adaptant ce type aux arbres, on crée alors les types [7] :

```
type feuille_formule = V | F | Vb of string;;
type operateur = And | Or | Not;;
```

et on définit des arbres sans constructeur `Vide` pour distinguer les feuilles :

```
type arbre_formule = Feuille of feuille_formule
| Noeud of operateur * arbre_formule list;;
```

L'évaluation dépend d'une interprétation pour les variables, et un algorithme naïf ne détectera pas dans un premier temps que `x ou (non x)` s'évalue toujours à `vrai`, notamment quand `x` n'est pas instancié, c'est-à-dire non mentionné dans l'interprétation, ce qui causerait une erreur dans l'algorithme.

Pour représenter une interprétation, l'idéal serait de disposer d'un tabeau indexé par des noms de variables et leur associant leur valeur de vérité. Pour commencer, nous allons utiliser ici `n` variables identifiées par des chaînes de caractères contenant les nombres de `0` à `n-1`. On définit donc par renommage le type `interpretation` comme un `bool array`.

La fonction `int_of_string` permettant de convertir chaque identifiant en l'entier correspondant sera ici très utile.

L'évaluation d'une formule se fait de manière récursive, en déclenchant une erreur si un nœud avec l'opérateur `non` n'a pas exactement un fils [8] mais en assimilant une conjonction ou disjonction d'une seule formule avec la formule elle-même, une conjonction vide avec le vrai et une disjonction vide avec le faux [9]. On se protège de certains messages d'erreur grâce à l'évaluation paresseuse des booléens qui fait ignorer d'éventuelles branches litigeuses.

Exercice 1 : Écrire la formule et l'arbre donnés en exemple sous forme d'une expression en OCaml qui a les types créés dans le TP. On remplacera directement les variables propositionnelles par des chaînes contenant des entiers naturels.

7. ... en changeant les noms des constructeurs pour pouvoir faire coexister les types
8. Ce n'est a priori pas la peine de compliquer la définition du type pour que la structure empêche cela.
9. les éléments neutres des opérations en question dans le semi-anneau des booléens

Exercice 2 : Écrire la fonction `evalue v arbre` qui évalue la formule propositionnelle représentée par `arbre` avec le type défini ci-avant, suivant l'interprétation `v`. Écrire aussi une fonction d'évaluation pour le type `f_b`, tant qu'à faire.

Exercice 3 : Donner la complexité de la fonction précédente.

Exercice 4 : Décrire un algorithme naïf qui permet de déterminer si une formule booléenne est satisfaisable. Donner une signature possible pour une implémentation en OCaml et écrire la fonction associée.

Exercice 5 : Reprendre le TP avec une structure de dictionnaire pour les interprétations, en rétablissant les lettres pour les variables propositionnelles.

L'algorithme de Quine

Dans cette section, on prend une version d'arité limitée du premier type somme pour les formules considérées et une autre structure arborescente pour les formules transformées :

```
type f_b_b = True | False | Variable of string
| Neg of f_b_b | And2 of f_b_b * f_b_b | Or2 of f_b_b * f_b_b;;
type arbre_formule2 = Leaf of feuille_formule
| Node of string * arbre_formule2 * arbre_formule2;;
```

En pratique, le type `arbre_formule2` ressemblera beaucoup aux arbres de décision déjà rencontrés.

L'algorithme de Quine sert à déterminer si une formule est satisfaisable sans passer par l'énumération de toutes les interprétations, mais sa complexité reste exponentielle dans le pire des cas (vu qu'on traite un problème NP-complet...).

Il s'agit de commencer par transformer la formule en une formule équivalente qui est soit `True`, soit `False`, soit totalement privée de ces deux constructeurs. De plus, cette formule équivalente est censée être de taille moindre. On appellera cette transformation la *simplification* de la formule.

Exercice 6 : Trouver la relation de récurrence vérifiée par la simplification, à partir des règles de base de la logique. Écrire alors une fonction réalisant la simplification.

Une fois que l'on dispose d'une formule dont les feuilles ne sont que des variables, il s'agit d'écrire par élimination successive de toutes les variables une formule équivalente sous forme d'un arbre de décision.

Concrètement, l'algorithme construit à partir de la formule `f`, contenant au moins une variable, et de `v`, une variable apparaissant dans `f`, les formules `fbis` qui en est la simplification, puis `f0` qui est `fbis` dont toutes les occurrences de `v` sont remplacées par `False` et `f1` qui est `fbis` dont toutes les occurrences de `v` sont remplacées par `True`.

Si `fbis` est `True` ou `False`, c'est fini. Sinon, la formule `f` est satisfaisable si, et seulement si, la disjonction de `f0` et de `f1` l'est, donc on applique récursivement l'algorithme (y compris l'étape de simplification, bien entendu) à ces deux formules. Dès qu'un `True` est rencontré, c'est gagné. Si on est tombé à chaque fois sur `False`, alors la formule est insatisfaisable.

Pour une belle représentation, on pourra aussi construire l'arbre dont chaque nœud est soit une feuille `V`, soit une feuille `F`, soit un nœud étiqueté par la variable éliminée et ayant pour fils les arbres obtenus en convertissant `f0` et `f1` en arbres.

On pourra tenter de profiter de l'évaluation paresseuse des booléens, par ailleurs !

Concernant le choix de la variable, on pourra prendre la première rencontrée (version simple) ou la plus fréquente pour espérer accélérer les choses (version compliquée), en gérant les égalités comme on veut.

Exercice 7 : Écrire l'algorithme de Quine avec la version simple du choix de la variable.

Exercice 8 : Écrire l'algorithme de Quine avec la version compliquée du choix de la variable.

TP 15 : Une base de données sommaire

Dans ce TP, nous allons travailler sur une base de données présentée en cours, et effectuer des requêtes SQL, en plus de pouvoir procéder pour le travail chez soi à des manipulations à l'aide de la plate-forme PhpMyAdmin.

L'adresse de la base de données est `http://phpmyadmin.online.net`, et l'identifiant est db86896.

Le mot de passe sera communiqué pendant la séance exclusivement

La base de données peut être récupérée pour une utilisation individuelle sur un serveur personnel [10] (le TP est également donné en tronc commun en PC, d'où l'adresse).

La structure associée est la suivante :

— Table Etudiants, avec les attributs Id (clé primaire avec auto-incrémentation, entier naturel), Classe (chaîne de caractères), Nom (idem) et Prenom (idem). On peut considérer que (Nom, Prénom) est également une clé.
— Table Examens, avec les attributs Id (clé primaire avec auto-incrémentation, entier naturel), Date (chaîne de caractères) et Coeff (entier naturel).
— Table Notes, avec les attributs Etudiant (entier naturel), Examen (entier naturel) et Note (entier naturel). Le couple (Etudiant, Examen) est une clé.

Dans chacun de ces exercices, on se contentera d'écrire une requête correspondante.

Exercice 1 : Déterminer qui a la meilleure note à l'examen numéro 2 parmi les MPSI 2.

Exercice 2 : Déterminer la moyenne du meilleur étudiant.

Exercice 3 : Déterminer combien de MPSI 1 ont sous la moyenne.

Exercice 4 : Déterminer quelle classe a la meilleure moyenne.

Pour les deux exercices suivants, on considère que les étudiants de toutes les classes passent les mêmes examens.

10. `http://jdreichert.fr/Enseignement/CPGE/PC/bdd_td_1.sql`

Exercice 5 : Déterminer dans quelle classe est l'étudiant ayant majoré le premier examen.

Exercice 6 : Déterminer le nombre de majors par classe.

Exercice 7 : Déterminer la classe avec le plus grand pourcentage d'étudiants ayant la moyenne.

TP 16 : Une base de données avancée

Pour voir une mise en œuvre pratique des bases de données, rien de tel qu'un exemple concret et véritablement utilisé : la gestion de mini-tournois de skat sur mon site. [11]. Les entrées correspondent aux quatre premiers tournois de l'année 2023.

Ce TP se fait en totalité sur la plateforme phpMyAdmin mise à disposition sur `http://phpmyadmin.online.net`, l'identifiant est **db94381** et le mot de passe est communiqué au cours de la séance.

La base de données constituée au moment de préparer le TP a été exportée et elle peut être récupérée pour une utilisation individuelle sur un serveur personnel [12].

La structure de la base de données est la suivante :

— Table `Goetz_Minitournoi`, inutile pour le TP (données annexes sur un tournoi, seule la clé primaire compte).

— Table `Goetz_Joueurs`, également inutile pour le TP (données annexes sur les joueurs, dont l'identité est rappelée dans la table suivante sans utiliser à proprement parler de clé étrangère).

— Table `Goetz_Inscription`, avec les attributs `Id` (clé étrangère vers la table `Goetz_Minitournoi`), `Numero` (identifiant d'inscription pour le tournoi en cours), `Nom` (chaîne de caractères, identité du joueur) et `Parti` (entier indiquant le nombre de séries où le joueur était présent, 0 s'il a joué jusqu'au bout, inutile pour ce TP). Le couple `(Id, Numéro)` est une clé, le couple `(Id, Nom)` aussi.

— Table `Goetz_Serie`, avec les attributs `Id` (*idem supra*), `Serie` (numéro de la « série », c'est-à-dire la phase de trente-six ou quarante-huit donnes où les joueurs restent à la même table), `Numero` (relie à la table `Goetz_Inscription`), `Num_table` (numéro de la table où le joueur est pendant la série), `Place` (place entre un et quatre du joueur à sa table), `Resultat` (nombre de points marqués), `Gagnes` (nombre de donnes gagnées en tant que preneur), `Perdus` (nombre de donnes perdues en tant que preneur). Le triplet `(Id, Serie, Numero)` et le quadruplet `(Id, Serie, Num_table, Place)` sont des clés.

11. Pour tout dire, les requêtes effectuées sur le site sont souvent moins complexes, dans la mesure où les calculs et agrégations peuvent tout aussi bien être faites à l'aide du langage de scripts qui récupère le contenu de la base de données en exécutant les requêtes SQL.

12. `http://jdreichert.fr/Enseignement/CPGE/MP2I/bdd_tp_16.sql`

À noter : sur la plate-forme se situe une ancienne version de la base de données correspondant à un vieux TP. Elle est laissée telle quelle au cas où mais ne sera pas abordée dans ce TP.

Exercice 1 (*) : Écrire et exécuter une requête qui permet de récupérer le nombre de joueurs s'étant inscrits au tournoi d'identifiant `4`.

Exercice 2 (**) : Écrire et exécuter une requête qui permet de récupérer le nombre de joueurs s'étant inscrits au moins une fois à un tournoi.

Exercice 3 (****) : Écrire et exécuter une requête qui permet de déterminer le nombre de fois où un joueur était à la même table aux trois séries.

Exercice 4 (****) : Écrire et exécuter une requête qui permet de déterminer quel joueur a le plus grand nombre de points au total sur l'ensemble des tournois.

Exercice 5 (**) : Écrire et exécuter une requête qui permet de déterminer à quel tournoi le nombre d'inscrits était le plus grand et le nombre de ces inscrits.

Note : On ne peut pas totalement se fier au maximum de l'identifiant d'inscription, car il peut y avoir des annulations de participations...

Exercice 6 (**) : Écrire et exécuter une requête qui permet de déterminer à quel tournoi le meilleur score était le plus haut et ce meilleur score.

Exercice 7 (****) : Écrire et exécuter une requête qui permet de déterminer le nombre de fois où un joueur était à une place de même parité aux trois séries.

Exercice 8 (*****) : Écrire et exécuter une requête qui permet de déterminer le nombre de fois où deux joueurs, en les précisant, ont été à la même table (donc au cours de la même série d'un même tournoi), si ce nombre est au moins deux. Adapter cette requête pour déterminer le nombre de paires de joueurs qui ont déjà été au moins deux fois à la même table.

Un raffinement de cette dernière requête est de déterminer le nombre de paires de joueurs qui ont été n fois à la même table pour chaque n (parmi ceux où la réponse n'est pas 0).