Option informatique
première année

Julien Reichert

2020/2021

Table des matières

Première partie

Cours

Chapitre 1

Le langage Caml [1]

1.1 Introduction

1.1.1 Les données et leurs types

Les types de données, déjà vus avec Python [2], ont une utilisation plus rigoureuse en Caml, dans la mesure où le compilateur [3] ne va jamais faire de conversion lui-même en cas d'incompatibilité du type d'une expression avec le type attendu par son utilisation.

Ainsi, la plupart des fonctions attendent un certain nombre d'arguments d'un type précis et leur valeur de retour en a un aussi, ce qui se lit dans la signature de chaque fonction. [4]

Les types de base comprennent les entiers (`int`), les flottants (`float`) et les booléens (`bool`). Cette fois-ci, les booléens n'héritent pas de la structure d'entier et on ne peut

1. Remarque : Le cours d'algorithmique et programmation I d'informatique pour tous est un prérequis à ce cours.
2. Cependant, le système des types de données est très différent entre les deux langages.
3. Les expressions ne sont pas à proprement parler interprétées en Caml, mais ceci est une autre histoire ; *a contrario*, on n'a pas de compilateur en Python.
4. On parle de « type précis » car dans certains cas le type peut être variable, par exemple un test d'égalité impose seulement que les deux objets comparés soient de même type... en particulier, on n'a pas le droit de comparer `0` et `0.0`, ce qui est assez déstabilisant.

donc pas les assimiler à `0` et `1`.

Plus précisément, les opérations spécifiques sont :
- `+`, `-`, `*`, `/` et `mod` pour les entiers[5].
- `+.`, `-.`, `*.`, `/.` et `**` pour les flottants[6].
- `&&`, `||` et `not` pour les booléens[7].

Attention : Le point ne se met qu'après ces opérateurs (et on l'utilise évidemment aussi comme séparateur décimal), inutile de le mettre après les noms de fonctions agissant sur les booléens...

On produit des booléens à l'aide de comparaisons, les symboles étant $<$, $>$, $<=$, $>=$, $=$ (égalité[8], les affectations fonctionnant différemment comme nous le verrons plus tard) et $<>$ (différence[9]).

Pour additionner un entier et un flottant, on est obligé de procéder à une conversion manuelle, à l'aide de `float_of_int` (ajoute un point) et `int_of_float` (troncature à la manière de `int` en Python).

Les caractères (`char`) forment un type de base à part entière, ils sont entourés d'apostrophes simples sans second choix. Il est interdit de mettre un nombre différent de un de caractères entre les apostrophes (les caractères échappés du genre `'\n'` ne comptent que pour un).

Les chaînes de caractères sont un type que l'on considère encore comme simple, quand bien même ils ont une structure similaire à des types composés comme les tableaux (une remarque déjà faite pour Python).

On les produit en entourant un certain nombre de caractères par des guillemets,

---

5. L'opérateur `mod` remplace donc le `%` de Python. On notera que la division euclidienne a un comportement anormal sur les valeurs négatives, par exemple `-3/2` vaut `-1`.
6. Ceci est une première illustration de ce qui a été annoncé pour les types : même les opérateurs doivent être modifiés. En pratique, ce n'est simplement pas le cas pour la puissance car elle n'existe pas pour les entiers et s'applique donc à deux flottants. Il est également interdit d'omettre le `0` devant le point si un flottant a `0` pour partie entière.
7. L'évaluation est paresseuse, contrairement au cas des symboles simples, qu'on évitera.
8. Au passage, l'opérateur `==` existe et son utilisation est rare : il détermine si les deux objets comparés ont la même adresse mémoire.
9. La même remarque s'applique quant à `!=`.

également sans second choix, la chaîne vide étant alors `""`.

L'accès à un caractère se fait à l'aide de la syntaxe `t.[i]`, où `t` est une chaîne de caractères et `i` est un indice entre `0` et la longueur de `t` moins un, cette longueur s'obtenant à l'aide de la fonction `String.length`, puisque là aussi on ne dispose pas d'une unique fonction calculant la longueur pour des objets de types totalement différents.

Une chaîne de caractères était mutable (rappel : en Python non) jusque dans les années 2010, y compris en Caml light dont l'enseignement en CPGE s'est arrêté en 2017, et la modification se faisait à l'aide de l'opérateur `<-`, formé du symbole d'infériorité et du trait d'union. Désormais, on pourrait se servir (mais on ne le fera pas) du type `bytes` ou faire un tableau de caractères pour avoir la mutabilité.

On peut obtenir la concaténation de deux chaînes de caractères par l'opérateur `^` et extraire une sous-chaîne à l'aide de la fonction `String.sub`, prenant en arguments une chaîne, l'indice de départ et la longueur de la sous-chaîne à extraire, retournant une erreur en cas de débordement. Une chaîne de caractères peut également s'initialiser par la fonction `String.make`, prenant pour arguments la longueur et le caractère à répéter.

Les tableaux (`array`) sont une structure similaire aux listes de Python, mais avec plus de rigidité : ils ne peuvent contenir que des données du même type.

Ainsi, un tableau sera par exemple un `int array` s'il contient des entiers, un `string array array` (le type se lit dans ce cas de droite à gauche, en quelque sorte) s'il contient des tableaux, eux-mêmes contenant des chaînes de caractères, etc. [10]

On les produit en délimitant les données par des **points-virgules** et en les entourant par `[|` et `|]` [11], le tableau vide étant alors `[||]`.

L'accès à un élément du tableau se fait cette fois-ci à l'aide de la syntaxe `t.(i)` (analogue), la longueur du tableau s'obtenant à l'aide de la fonction `Array.length`.

La fusion de tableaux n'est pas prévue, ce qui veut dire que la taille n'est pas mo-

10. Un `'a array` contient des données dont le type n'est pas encore imposé, par exemple le tableau vide.

11. La barre verticale s'obtient à l'aide de AltGr + 6/- sur un clavier AZERTY de PC.

difiable. En revanche, on peut modifier les éléments à l'aide de l'opérateur `<-` et extraire un sous-tableau à l'aide de la fonction `Array.sub`.

Un tableau peut s'initialiser en le donnant tel quel ou par la fonction `Array.make` (syntaxe similaire à celle de `String.make`).

Attention : Les soucis rencontrés en Python lorsqu'on copiait une liste dans une autre sans faire de copie profonde ont leur pendant en Caml. Par exemple, définir un tableau `m` de dimension deux comme `Array.make 3 (Array.make 3 0)` fait que modifier `m.(0).(0)` entraîne la même modification de `m.(1).(0)` et `m.(2).(0)`. La solution est de définir des matrices à l'aide de `Array.make_matrix`, prenant pour arguments le nombre de lignes, le nombre de colonnes et l'élément à répéter.

Les listes [12] (`list`) sont une structure spéciale, dans la mesure où elles ont un lien fort avec la récursivité. Une manière de voir les choses est qu'une liste est soit vide, soit un élément associé à un pointeur vers une autre liste.

Ainsi, on ne peut pas accéder à un élément de la liste, sauf le premier, en une opération, comme on le ferait pour un tableau.

Une liste ne peut contenir que des données du même type, et on aura donc par exemple des `int list`, des `float array list`...

On les produit en délimitant les données par des **points-virgules**, mais cette fois-ci en les entourant par `[` et `]`.

On peut aussi effectuer un préfixage à l'aide de l'opérateur « conse » `::`, dont la partie gauche est nécessairement un seul élément, ou une fusion à l'aide de l'opérateur `@`, dont les deux parties sont des listes à fusionner.

Attention : Ne pas confondre `a::l` en Ocaml (nouvelle liste, pas de modification de `l`) et `l.append(a)` en Python (retourne None, `a` est ajouté à `l`.

La longueur d'une liste s'obtient à l'aide de la fonction `List.length` [13].

12. La structure algorithmique est celle d'une liste simplement chaînée. Le raccourci de liste est courant pour les langages fonctionnels.
13. Attention, la fonction parcourt la liste, donc son coût est linéaire.

Une liste en elle-même n'est pas modifiable (pour rebondir sur la mise en garde), cependant, et comme dit précédemment l'accès à ses éléments ne peut se faire que par filtrage, ce sur quoi nous reviendrons. On comprendra bien que les fonctions `List.sub` et `List.make` n'existent pas.

Les conversions entre types sont possibles (cf. `float_of_int`), et pour cela on dispose de multiples fonctions :

— `int_of_string` et `string_of_int`, de même en remplaçant `int` par `float`, `bool` ou `char`, à ceci près que `char_of_string` n'existe pas (sans intérêt), ni `string_of_char`.

— `int_of_char` et `char_of_int` procèdent à une conversion, l'entier étant la position du caractère dans la table ASCII étendue (256 caractères).

— `Array.of_list` et `Array.to_list`, de complexité linéaire en raison de la structure de liste.

Les n-uplets sont la structure de données de Caml la plus proche de leur pendant en Python : ils rassemblent des éléments de types quelconques, on les forme en donnant les éléments séparés par des virgules et entourés de parenthèses ou non au choix (attention cependant aux cas d'ambigüité), et ils peuvent être déconstruits.

Cependant, ils n'ont pas de type à part entière (ce qui a pour conséquence directe qu'on ne peut pas calculer leur longueur), mais leur type est le produit des types de leurs éléments ; ainsi, un couple formé par une chaîne de caractères et une liste d'entiers aura pour type `string * int list`[14].

Pour des couples uniquement, on dispose des fonctions `fst` et `snd` retournant respectivement le premier et le deuxième élément du couple.

Le type `unit` doit être mentionné ici par souci d'exhaustivité, mais il sera bien plus détaillé ultérieurement. On peut l'assimiler au type de `None` en Python, car c'est le type des objets vides.

Le dernier type présenté ici, avant de parler de types définis par l'utilisateur, est la référence, qu'on présente conjointement avec la notion de variable.

14. Alors que `["",1]` sera une `(string * int) list`, en notant bien qu'elle comporte un seul élément, qui est un couple.

1.1.2 Variables

Cette information peut faire un choc : en Caml, la notion de variable n'existe pas. En pratique, on peut tout à fait considérer que les références sont des variables, et c'est ce que nous allons faire par la suite.

Pour commencer, donnons enfin la syntaxe des affectations : `let objet = valeur;;` définit globalement `objet` comme étant `valeur` *ad vitam æternam*, en pratique jusqu'à la prochaine affectation (qui ne fait que masquer la précédente) `let objet = autre_valeur;;`. Un nom de variable doit commencer par une minuscule ou un `_`.

Une définition locale s'écrit `let objet = valeur in morceau_de_code;;`, auquel cas `objet` ne sera `valeur` que dans le morceau de code en question.

Une façon de voir les choses est que toutes les occurrences de `objet` (là où on l'a défini) sont remplacées par `valeur` (sauf présence d'effets de bord dans la définition).

Cependant, une affectation ne peut pas se faire n'importe comment, et on utilisera majoritairement des définitions locales.

Une référence en Caml permet de travailler sur une version mutable d'un objet immutable. En fait, le type d'une variable définie comme une référence est précisément... une référence du type de l'objet.

Pour fixer les idées, si on veut affecter à une variable `i` des valeurs entières dans un certain intervalle, on ne va pas affecter à `i` toutes les valeurs successivement et écrire `let i = i + 1` (ce qui provoque des erreurs de syntaxe en plein milieu d'une boucle, entre autres).

Au contraire, `i` sera une référence d'entier : on la créera par `let i = ref 1` (là aussi ce sera presque toujours localement), mais comme `i` sera de type `int ref`, il apparaît qu'on ne pourra pas faire de calculs dessus, ce qui implique de déréférencer `i` pour accéder à la valeur correspondante, en écrivant `!i` (qui sera donc de type `int`).

Modifier la valeur d'une référence se fait par l'opérateur `:=` dont le membre gauche est une référence d'un certain type et le membre droit une valeur du même type, par exemple `i := 42`, cette nouvelle valeur pouvant dépendre de l'ancienne (en déréférençant, donc).

Pour des références d'entiers, deux fonctions assez pratiques permettent d'additionner ou de soustraire 1 : ce sont respectivement `incr` et `decr` (pour **incr**émentation et **décr**émentation).

Bien entendu, on peut faire une référence d'à peu près tout (et même une référence de références), mais l'intérêt est limité (notamment en ce qui concerne des références de tableaux ou de chaînes de caractères, alors que les références de listes sont très utiles).

1.1.3 Instructions

En Caml, un morceau de code se termine usuellement par deux points-virgules, même lorsqu'on demande de calculer `2+2`. Dans les fichiers à compiler, ce n'est pas une nécessité absolue, mais c'est une habitude à prendre.

Séquence d'instructions

Il est évidemment possible d'enchaîner deux instructions au sein d'un même morceau de code, ce qui nécessite de les séparer par un point-virgule. Cependant, Caml déclenche un avertissement lorsqu'une instruction qui n'est pas la dernière possède une valeur, c'est-à-dire qu'elle n'est pas du type `unit` (pour le moment, les instructions de ce type parmi celles qu'on a vues sont les modifications d'éléments d'une chaîne de caractères ou d'un tableau et les modifications de références).

Ceci peut s'expliquer intuitivement par le fait qu'il n'y ait pas d'instruction `return` en Caml, et donc la valeur de retour est nécessairement la valeur obtenue en évaluant la dernière instruction, toutes les autres valeurs n'étant donc pas retournées, ce qui mérite d'être signalé par Caml si ce n'est pas le comportement attendu par l'utilisateur.

Plus précisément, si on souhaite provoquer un effet de bord en appelant une fonction retournant également une valeur à l'intérieur d'une séquence d'instructions, on peut stocker dans une sorte de variable poubelle le résultat, en écrivant `let _ = f(x) in suite` au lieu de `f(x); suite`.[15]

Au passage, le code `let x = 2; let x = 2*x;;` provoque une erreur de syntaxe,

15. Ocaml dispose de la fonction `ignore` permettant aussi d'écrire `ignore(f(x)); suite`, ce qui est sans doute plus agréable à lire.

et plus généralement faire suivre une définition d'un point-virgule simple cause des ennuis divers et variés. Une version acceptée s'obtient en remplaçant le point-virgule par `in` ou par deux points-virgules. [16]

Caml ne s'encombre pas de considérations quant à l'organisation du code en termes d'espaces, de tabulations et de sauts de lignes. Cependant, quelqu'un qui lit les programmes s'y intéresse, lui [17], et une bonne recommandation est d'indenter comme on le ferait en Python.

Disjonction de cas (if)

La syntaxe de la disjonction de cas est `if condition then bloc1 else bloc2`, là encore le code pouvant être espacé comme on le souhaite. Il est impératif que la condition s'évalue en un booléen et surtout que le type obtenu en évaluant les deux blocs soit le même.

En particulier, bien qu'il soit autorisé de ne pas écrire la partie avec `else`, ceci sera considéré comme `else ()`, qui est de type `unit`, et la première partie doit donc respecter ceci.

Il n'y a pas de raccourci de type `elif` ou `elseif`, il faut donc imbriquer les disjonctions le cas échéant, ce qui alourdit la notation et renforce le besoin de bien présenter le code.

Attention : Dans le code `if cond1 then if cond2 then bloc1 else bloc2`, Caml comprendra que le `else` correspond au `if` intérieur, et si ce n'est pas ce qu'on souhaite, il faut parenthéser (on n'utilise pas d'accolades comme dans d'autres langages).

Filtrage

Étant quasiment indissociable de la définition d'une fonction, le filtrage sera détaillé plus loin.

Boucle inconditionnelle (for)

16. Ne parlons pas de "shadowing", mais il faudrait garder à l'esprit que redéfinir une variable en fonction de son ancienne valeur sans passer par les références est une mauvaise idée.
17. Ou plutôt : souhaite ne pas avoir à s'y intéresser…

La boucle inconditionnelle s'écrit `for variable = debut to fin do bloc done` ou `for variable = debut downto fin do bloc done`, suivant qu'on veuille incrémenter ou décrémenter la variable de boucle à chaque étape. Bien entendu, les bornes sont incluses (il ne faut pas se laisser piéger par une mauvaise compréhension de la sémantique de `range` en Python).

Le type lors de l'évaluation de `bloc` doit toujours être `unit`, sous peine de recevoir un avertissement, tout comme dans les séquences d'instructions[18].

Si par exemple dans le premier cas `debut` est strictement supérieur à `fin`, la boucle n'est jamais exécutée. Ainsi, `for i = 3 to 2 do i / 0 done` ne provoquera pas d'erreur de division par zéro (mais un avertissement sur le type dans le bloc).

Comme en Python, on ne peut pas perturber une boucle (disons pas facilement en Python), par exemple la boucle `let n = ref 10 in for i = 1 to !n do decr n done;;` sera effectivement parcourue dix fois. La raison est que lors de l'évaluation de l'instruction `for` la valeur de `fin` est évaluée une fois pour toutes. En outre, puisqu'on crée une variable `i` en donnant son nom, on ne peut pas en faire une référence, donc elle augmentera (ou diminuera) forcément d'une unité par tour de boucle (malheureusement, on ne peut pas choisir d'autre pas, et il faut alors soit créer une variable qui en dépend par une relation affine, soit utiliser une boucle conditionnelle).

En revanche, la variable de boucle est locale à celle-ci et n'a donc aucune existence en-dehors si elle n'y est pas définie par ailleurs[19].

Boucle conditionnelle (while)

La boucle conditionnelle s'écrit `while condition do bloc done`, où `condition` doit là aussi être un booléen et le type lors de l'évaluation de `bloc` doit toujours être `unit`. Comme en Python, la condition est évaluée avant chaque tour dans la boucle.

18. Pire que cela : L'évaluation de `for i = 0 to 0 do 2+2 done;;` ne retournera même pas 4.
19. Et même dans ce cas, on ne fait que la masquer : on ne récupère pas la valeur de `i` à la dernière itération après la boucle mais la valeur de `i` en-dehors de la boucle.

1.1.4 Fonctions

Caml étant un langage fonctionnel, la notion de fonction est d'importance capitale.

Une fonction est une expression qui attend un ou plusieurs arguments[20] et qui retourne une valeur.

La définition d'une fonction peut donc faire intervenir, entre autres, des séquences ou des blocs d'instructions, et la valeur retournée est la dernière instruction rencontrée.

La définition la plus simple d'une fonction est la donnée du nom de celle-ci, puis de noms de variables pour ses arguments et de donner la séquence d'instructions après le symbole =, par exemple :

```
let fact n = let res = ref 1 in
    for i = 2 to n do res := !res * i done;
    !res;;
```

La signature d'une fonction est l'information sur les types des arguments et de la valeur de retour. Elle est donnée en Caml par la syntaxe :

```
nom : type_arg1 → type_arg2 → ... → type_retour
```

La signature de la fonction précédente est donc fact : int → int, que Caml précise par ailleurs en val fact : int → int = <fun>.

Une fonction peut aussi se définir par filtrage (exhaustif, sinon Caml déclenche un avertissement) des cas, et on retiendra uniquement la syntaxe suivante (les syntaxes fun et function seront abordées en deuxième année) : match expr with, où expr peut être n'importe quelle expression, habituellement un simple nom de variable de n'importe quel type ou un n-uplet de noms de variables.

Le filtrage se fait en introduisant chaque cas (sauf éventuellement le premier) par une

20. La syntaxe de la signature d'une fonction permet de constater qu'une fonction sans arguments est simplement une constante. Si une fonction ne doit dépendre de rien on lui donne en fait comme argument (), qui est de type unit. En particulier une constante est évaluée lors de sa définition : let a = Random.int 6;; affecte à a un nombre aléatoire entre 0 et 5, ce qui n'a rien à voir avec let b () = Random.int 6;; qui définit une fonction retournant à chaque appel un nouveau nombre aléatoire entre 0 et 5.

barre verticale et en le faisant suivre d'une flèche et du bloc d'instructions quand le cas est rencontré, en utilisant pour l'esthétique une représentation alignée verticalement.

On peut se servir du joker `_` pour signifier « tous les cas restants » en ayant bien à l'esprit que seul le premier cas favorable est retenu. Ainsi, un filtrage peut s'assimiler à un enchaînement de tests conditionnels.

Un exemple valant mieux qu'un long discours, voici une illustration de cette syntaxe pour définir des fonctions booléennes.

```
let et b1 b2 = match (b1, b2) with
|(true, true) -> true
|_    -> false (* (_,_) marche aussi
(et au passage ceci est notre premier commentaire)
(d'ailleurs ceux-ci peuvent être mis sur plusieurs lignes) *);;

let ou (b1, b2) = match (b1, b2) with
|(true, _) | (_, true) -> true
|_ -> false;;
```

Attention, les signatures dépendent de la façon dont les arguments sont présentés.

Ainsi, on aura `et : bool` → `bool` → `bool`, mais `ou : (bool * bool)` → `bool`; en effet, la signature dépend de la définition et non du filtrage.

Remarque : L'héritage du lambda-calcul fait que les fonctions en Caml sont dites **curryfiées** [21], c'est-à-dire dont les arguments peuvent être donnés partiellement (mais dans l'ordre, évidemment) afin de créer d'autres fonctions.

Par exemple, si on écrit `let add x y = x + y;;`, la fonction `add` est curryfiée, car `add 3` est une fonction de signature `int` → `int` qui étant donné un entier retourne trois de plus.

On notera qu'il est possible d'écrire en Caml une fonction pour curryfier et décurryfier des fonctions ayant le même nombre d'arguments.

21. du nom de l'informaticien Haskell Curry, d'où également le langage Haskell que quelques experts préfèrent à Caml[réf. nécessaire]

Ainsi, `let curry2 f a b = f (a,b);;` décrit une fonction dont la signature est `curry2 : ('a * 'b * → 'c) → 'a → 'b → 'c`, et si `f` est une fonction prenant un argument sous forme de couple, `curry2 f` est sa version curryfiée prenant deux arguments.

Par conséquent, on peut considérer en pratique qu'une fonction qui renvoie une fonction correspond à une fonction à deux arguments (ou plus).

Il apparaîtra très vite que les filtrages tels quels manqueraient de puissance, mais on peut les compléter par des conditions (remplaçant avantageusement des tests conditionnels après la flèche) introduites par le mot-clé `when` suivi d'une condition. Cette syntaxe ne sera pas étudiée ni utilisée en première année.

Attention, un nom de variable dans le filtrage est local et ne peut être utilisé qu'une fois dans un même cas de filtrage.

Par exemple, les deux premiers codes ci-après sont erronés (le premier donne une fonction incorrecte, le deuxième déclenche même une erreur), mais le suivant est correct (lourd au possible, mais correct) :

```
let mauvais_xor b1 b2 = match b2 with
|b1 -> false (* shadowing sur b1, vu comme une nouvelle variable *)
|_ -> true (* conséquence : ce cas n'est jamais rencontré *);;

let xor_qui_plante b1 b2 = match (b1, b2) with
|(b, b) -> false
|_ -> true;;

let bon_xor b1 b2 = match b2 with
|a -> if a = b1 then false else true;;

(* Sans filtrage : let xor a b = not (a = b);; *)
```

Remarque : Le filtrage est en fait une recherche de motifs, plutôt qu'un test d'égalité, et Caml affecte dans la foulée les variables correspondant aux arguments quand il trouve le motif. Le fait qu'un nom de variable soit local et ne puisse être utilisé qu'une fois est dû à un souci d'optimisation de la recherche de motifs, afin qu'elle reste de complexité linéaire en temps.

En outre, si la partie à droite de la flèche contient un nouveau filtrage, un conflit de syntaxe peut être déclenché, car un nouveau motif sera compris pour Caml comme correspondant au filtrage le plus intérieur, d'où l'intérêt de parenthéser les blocs trop gros.

Il s'avère également qu'on peut capturer plusieurs constantes à la fois dans un cas de filtrage (mais sans utiliser de noms de variables, sinon l'expression de droite peut être valide dans un cas de filtrage et non dans l'autre), il suffit de ne pas mettre de flèche à la suite des cas équivalents :

```
let voyelle carac = match int_of_char carac with
| 65 | 69 | 73 | 79 | 85 | 89
| 97 | 101 | 105 | 111 | 117 | 121 -> true
| c -> if c > 65 && c < 91 || c > 97 && c < 123 then false
       else failwith "Ceci est une façon de déclencher une erreur";;
```

1.1.5 Entrées et sorties

Les fonctions d'entrées et de sorties sont multiples, précisément parce qu'il en faut une par type de base.

Ainsi, pour imprimer un entier `x`, on écrira `print_int x` (de signature `int` $\to$ `unit`), mais si `x` est un flottant, on écrira `print_float x`.

Les autres fonctions d'impression classiques sont `print_char`, `print_string`, ainsi que `print_newline` (retour à la ligne, de signature `unit` $\to$ `unit`) et `print_endline` (imprime la chaîne en argument retourne à la ligne).

Remarques :

— Rien n'est prévu dans la bibliothèque standard pour les booléens, les listes, les tableaux, les n-uplets, et d'autres types exotiques.

— La fonction `print_string` imprime dans un buffer qui n'est affiché que lorsque les instructions sont terminées ou quand un saut de ligne est effectué. Voir par exemple le résultat de `print_string "plop"; print_string "\b";;` et celui de `print_string "plop\n!"; print_string "\b\b\b";;`, en signalant que le caractère spécial imprimé est le retour arrière (*backspace*).

En ce qui concerne la lecture, les fonctions suivantes attendent une saisie de l'utilisateur et les convertissent si possible : `read_int`, `read_float`, `read_line` (pour les chaînes de caractères).

Pour les fichiers, on ouvre un fichier (en tant que canal) en mode lecture ou écriture à l'aide de deux fonctions différentes : `open_in` et `open_out`, de signatures respectives `string` $\rightarrow$ `in_channel` et `string` $\rightarrow$ `out_channel`, l'argument étant le chemin vers le fichier à ouvrir. La fermeture se fait logiquement à l'aide de `close_in` et `close_out`.

Trois canaux sont toujours ouverts (et mieux vaut éviter de les fermer) : `stdin`, `stdout` et `stderr`, qui sont l'entrée standard, la sortie standard et le canal d'erreur (de sortie, donc) standard.

La lecture et l'écriture depuis et dans un fichier se fait à l'aide des fonctions de base (mais seuls les caractères et chaînes de caractères fonctionnent), en remplaçant `read` par `input` ou `print` par `output` et en précisant en premier argument le canal. Ainsi, la fonction `print_string` est équivalente à `output_string stdout`, par exemple.

1.2 Compléments

Attention, contrairement à la section correspondante du chapitre 2 d'informatique pour tous de première année, de nombreux passages de cette section sont essentiels et seront traités en cours.

1.2.1 Définitions simultanées

Les définitions peuvent aussi être simultanées, en se servant du mot-clé `and`.[22] Dans ce cas, si l'un des éléments dépend de l'autre, Caml provoquera une erreur.

Par conséquent, on peut écrire `let x = 2 in let y = 4 * x and z = 4 - x;;` mais pas `let x = 2 and y = x * x;;`.

Pire que cela, on peut l'écrire si `x` existait avant, et c'est l'ancienne valeur qui est prise en compte, d'où des confusions. Au passage, comparer les résultats des deux codes suivants :

```
let x = 42;;
let y = 19;;

let x = true and y = 12 in x;;
```

```
let x = 42;;
let y = 19;;

let x = true && y = 12 in x;;
```

1.2.2 Liaisons statiques

Une différence entre Python et Caml en ce qui concerne les définitions de fonctions est que Caml évalue le code au moment de la définition, alors que Python le fait au moment de l'appel.

La notion en jeu ici est celle de *liaisons statiques*, par opposition aux *liaisons dynamiques*.

Pour donner un exemple, en Python, le code suivant imprimera successivement `1` puis `2`, ce qui signifie que le code de la fonction `f` aura effectivement changé par la redéfinition de `g`, alors que le code transcrit en Caml imprimera deux fois `1` :

22. C'est utile dans le cas de fonctions mutuellement récursives.

```
def g():
    print(1)
def f():
    g()
f()
def g():
    print(2)
f()
```

```
let g () = print_int 1;;
let f () = g ();; (* ou let f = g *)
f();;
let g () = print_int 2;;
f();;
```

1.2.3 Fonctions locales, fonctions anonymes

Puisque Caml permet de faire des définitions locales, il est tout à fait envisageable de définir des fonctions dans des fonctions.[23]

Bien évidemment, une fonction locale n'existe que dans la fonction où elle est définie.

En outre, si on veut utiliser une fonction (si possible courte), `f` par exemple, que l'on définirait par `let f x = bloc`, on peut éviter de la définir en utilisant directement le bloc ainsi : `(fun x -> bloc) argument`.

En fait, `fun x -> bloc` a la signature que `f` aurait eue.

Ces fonctions anonymes, plus adaptées à Caml qu'à Python, se combinent bien avec des fonctions comme `map` ou `iter`, du module `List`, que l'on verra en TP.

1.2.4 Types somme et produit

En Caml, il est possible de créer ses propres types, et ce de trois façons. Le mot-clé pour la définition est `type`[24].

La première façon est simplement de renommer des types existants, éventuellement des types construits existants.

Par exemple, on peut considérer qu'un nombre complexe est la donnée de deux réels, ce qui se traduit par `type complexe = float * float;;`.

23. Là aussi, la récursivité fournira des motivations dans ce but.
24. et non `let`, bien que cela eût pu être envisageable

Bien entendu, Caml n'a aucune raison de considérer par défaut qu'un couple de flottants est un `complexe`. On peut cependant forcer un type, par exemple pour un argument de fonction, à l'aide d'une syntaxe apparaissant sur les exemples suivants : `let module (z:complexe) = let x, y = z in sqrt (x ** 2. +. y ** 2.);;` et `let i = ((0.,1.):complexe);;`.

La deuxième façon, donnant des types somme, est de donner des constructeurs, ce qui permet dans les cas simples d'obtenir une liste exhaustive des objets ayant le type défini, et dans les cas avancés de créer des types dont les objets peuvent être construits à partir de sous-objets d'un autre type (ou du même).

La syntaxe est dans ce cas `type montype = Elt1 | Elt2 | Elt3 of sontype | ...`, où les constructeurs `Elt1`, `Elt2` et `Elt3` doivent commencer par une majuscule, sinon Ocaml déclenche une erreur.

Attention : un constructeur ne doit jamais être utilisé deux fois pour deux types différents, car seule la dernière définition serait alors prise en compte.

Dans l'exemple ci-avant, il est tout à fait possible que `sontype` et `montype` soient les mêmes.

En outre, définir un type utilisant des constructeurs d'un autre type utilisant eux-mêmes des constructeurs du premiers nécessite une définition simultanée des types (voir aussi le chapitre sur la récursivité [25]).

Les éléments d'un type construit s'appellent naturellement en utilisant les constructeurs, avec le ou les arguments nécessaire(s).

Par exemple, on redéfinit les types `bool` et `'a list` : `type mybool = Vrai | Faux;;` et `type 'a mylist = Vide | Cons of ('a * 'a mylist);;`.

Plus complexe et récursif : `type boum = Pif | Paf | Pouf of general and general = Ratatata | Poum of boum;;` [26], dont un élément est `Pouf(Poum(Pif))`.

La troisième façon, donnant des types produit ou enregistrement (*record*), se rapproche de la programmation objet. En pratique, un type produit contient des « ru-

25. ... qui ne manquera pas de référer à ce paragraphe !
26. Vive la Grande-Duchesse !

briques »[27], utilisant elles-mêmes un type chacune.

Pour créer un type produit, on écrit

```
type monproduit = {rub1 : type1 ; rub2 : type2 ; ...};;
```

et pour un objet, on initialise les valeurs correspondant à chaque rubrique en écrivant (peu importe l'ordre pourvu que l'association soit correcte et complète)

```
let exemple = {rub1 = valeur1 ; rub2 = valeur2 ; ...};;
```

ce qui permet d'accéder à la valeur d'une rubrique par `exemple.rub1`.

Un objet d'un type produit peut avoir des rubriques mutables (dont le nom de rubrique est alors précédé du mot-clé `mutable`) et des rubriques non mutables ; la modification de la valeur d'une rubrique (à condition qu'elle soit mutable, donc) suit la syntaxe de la modification d'éléments d'un tableau. Un exemple ci-dessous :

```
type individu = {mutable nom : string; prenoms : string list;
mutable age : int; sexe : bool};;

let anonyme = {nom = "Martin"; prenoms = ["Camille";"Dominique"];
age = 42 ; sexe = false};;

let joyeuxanniversaire indiv = indiv.age <- indiv.age + 1;;
```

1.2.5 Exceptions

Les exceptions témoignent de comportements imprévus (pas forcément inattendus) du programme, par exemple une division par zéro, un accès à un élément inexistant d'un tableau, un accès à la tête d'une liste vide, etc.

Les erreurs de syntaxe et de type, entre autres, ne sont pas des exceptions, car elles se situent au niveau de l'analyse lexicale et sémantique d'un programme, et ne sont alors pas « rattrapables ».

Une exception a son type propre, noté `exn`, et il s'avère que déclencher une exception

27. pour reprendre le terme officiel de la documentation

permet... exceptionnellement que le type d'une expression puisse être différent, plus précisément une expression a un type quelconque prévu, et éventuellement le type exception.

Tout comme les objets des types standards, les exceptions peuvent être créées par l'utilisateur, à l'aide du mot-clé `exception`.

Comme dans le cas des types somme, les exceptions peuvent être constantes ou paramétrées, c'est-à-dire munies d'un type.

Pour déclencher une exception (la traduction littérale est « soulever »), on la précède du mot-clé `raise`[28].

L'intérêt majeur des exceptions est de pouvoir être rattrapées, permettant un filtrage suivant l'erreur déclenchée. Ceci sera une façon recommandée de quitter prématurément une boucle ou un morceau de code, entre autres.

La syntaxe est `try code with erreur1 -> code1 | erreur2 -> code2 | ...`.

La sémantique est la suivante : Caml évalue `code`, et retourne sa valeur finale ; si une exception est déclenchée, Caml va regarder si elle correspond successivement et dans l'ordre dans lequel elles sont énoncées aux erreurs situées après le mot-clé `with`, et la première erreur reconnue provoquera l'exécution du code associé, dans lequel cette fois-ci les exceptions qui seraient déclenchées ne sont pas rattrapées par le même filtrage (mais potentiellement par un filtrage extérieur).

Cette fois-ci, le filtrage n'a pas besoin d'être exhaustif, les erreurs non rattrapées étant alors forcément transmises.

Bien entendu, le type de l'expression obtenu par l'évaluation de `code`, de `code1`, de `code2` et de tous les autres blocs de code doit être le même[29].

Ainsi, la recherche de la première position d'un caractère dans une chaîne pourra utiliser une exception personnelle :

```
exception Trouve of int;;
```

28. Un équivalent de `failwith "paf"` est donc `raise (Failure "paf")`.
29. Ou, comme on l'a vu, certains peuvent être `exn`.

```
let cherche_chaine carac s =
  try
    for i = 0 to String.length s - 1 do
      if s.[i] = carac then raise (Trouve i)
    done; -1
  with
    |Trouve ind -> ind;;
```

Quelques exceptions très habituelles de Caml :

```
Uncaught exception: Division_by_zero (* 1/0 *)
Uncaught exception: Failure "hd" (* hd [] *)
Uncaught exception: Failure "tl" (* tl [] *)
Uncaught exception: Invalid_argument "index out of bounds"
 (* t.(-1), et donc on ne peut pas partir de la fin *)
```

L'exception qui sera utilisée de manière la plus classique est la `Failure`, déclenchée par le mot-clé `failwith` suivi d'une chaîne de caractères constituant le message d'erreur, que l'utilisateur adaptera au contexte.

1.2.6 Modules

Bien que la plupart des fonctions, types et autres objets utiles soient dans le module de base, Caml dispose de modules et bibliothèques complémentaires, dont la gestion n'est pas la même que celle de Python.

Ainsi, des modules de la bibliothèque standard sont déjà chargés et prêts à être ouverts, comme par exemple `Printf`, `Random` et `Sys`.

Dans ce cas, pour utiliser une fonction sans ouvrir le module, il faut la préfixer par le nom du module suivi d'un point [30].

L'ouverture du module par `open Nom_du_module;;` permet d'éviter ce préfixage, mais peut déclencher des conflits de noms (d'éventuelles définitions homonymes sont écrasées).

Des bibliothèques extérieures (comme la bibliothèque graphique `graphics`) doivent

30. On a déjà vu auparavant `Random.int`.

être chargées avant l'étape précédente : `#load "graphics.cma";;` par exemple (par chance, sous Windows, l'interpréteur classique s'en occupe par défaut, ce qui ne posera pas de problème).

Certains modules de Caml sont présentés en TP.

1.2.7 Formats

Les fonctions d'impression vues dans la section sur les entrées et sorties ont pu décevoir par leur faiblesse comparée à la flexibilité de la fonction `print` de Python3.

Pour pallier ce manque de fonctionnalités, Caml fournit comme beaucoup d'autres langages la possibilité d'utiliser des formats pour une impression fluide de chaînes de caractères contenant des valeurs paramétrées.

Ainsi, le module `Printf` [31] fournit (entre autres) les fonctions `printf`, `fprintf`, `eprintf`, imprimant respectivement sur la console, sur un canal de sortie en argument et sur la sortie d'erreurs standard une chaîne de caractères formatée selon la syntaxe présentée ci-après.

Une quatrième fonction utile, `sprintf`, retourne la chaîne formatée au lieu de l'imprimer.

Une chaîne de caractères formatée est une chaîne de caractères contenant un certain nombre d'occurrences du symbole de pourcentage associé à une lettre ou à lui-même (pour pouvoir tout de même produire le symbole), afin de produire une valeur paramétrée dont le type dépend de la lettre associée, les principales étant `d` pour un entier (également `i` pour un entier signé), `f` pour un flottant, `s` pour une chaîne de caractères, `c` pour un caractère et `b` pour un booléen.

Pour imprimer un format, il faut donner après ce format des valeurs correspondant à chaque combinaison dans le même ordre d'apparition.

Si des valeurs manquent, en accord avec les principes de Caml, on obtient une fonction qui attend les valeurs manquantes en tant qu'arguments.

Exemples :

31. issu du langage C

```
let table_multiplication m n =
(* table de n entre 0*n et m*n *)
  for i = 0 to m do Printf.printf "%d x %d = %d\n" i n (i*n) done;;

let date jour mois an = Printf.sprintf "%02d/%02d/%d" jour mois an;;
(* %nd, où n est un entier bien défini, signale que la taille
doit être au moins n, en complétant par des espaces,
et %0nd complète par des zéros. *)

let imprime_format canal n =
Printf.fprintf canal "Vous avez %d nouveaux messages." n;;
```

1.3 L'essentiel

Malgré le fait que le nouveau langage à apprendre en parallèle de Python diffère bien de celui-ci, il ne faut pas voir ceci comme un obstacle insurmontable. En pré-bac, il est courant de voir des élèves apprendre simultanément l'anglais et l'espagnol, soit deux langues très différentes, et pourtant peu de gens utilisent par erreur des mots d'espagnol dans une phrase en anglais.

En fait, le nom de langage de programmation ne fait pas penser aux langues naturelles par hasard. Ce qui compte pour l'apprentissage d'un langage, c'est de maîtriser sa syntaxe (informatiquement, il s'agit de la grammaire, c'est-à-dire comment organiser les mots-clés en un programme compréhensible) et sa sémantique (le vocabulaire, c'est-à-dire la liste des mot-clés et des fonctions avec leur spécification, donc leur sens, et leur type, qu'on peut voir en parallèle avec le fait de savoir si un mot est un verbe, un nom, etc.).

Pour apprendre la sémantique, il n'y a pas de secret, c'est essentiellement du par cœur ou de l'expérience, et une fois ceci maîtrisé, l'étude des programmes qu'on écrit revient à les décomposer en instructions élémentaires, à étudier en les évaluant selon les priorités exactement comme Ocaml ferait.

Les écueils classiques relevés au cours de ma carrière :
- écrire `a::l` ou `l@ll` (éventuellement avec une erreur de syntaxe) en croyant que cela modifie `l`, plus généralement un défaut de détection des moments où utiliser les références ;
- à ce propos, la syntaxe des références est également sujette à des confusions, on se souviendra donc que si `x` est une `'a ref`, la ligne qui permet de tout mémoriser de manière concise est `x := !x` (code évidemment inutile) ;
- comme il n'y a pas de `return` en Ocaml, on prendra garde à bien organiser les fonctions afin que la valeur retournée ne soit pas suivie d'une instruction (si elle est suivie d'un `else`, cela ne compte donc pas), et il faut faire un effort de présentation si cette valeur apparaît avant un `else` très long.
- ne **jamais** écrire un `let` suivi d'un simple point-virgule en milieu d'une fonction, d'ailleurs il ne faut pas non plus oublier les points-virgules qui séparent plusieurs instructions (si on veut copier la syntaxe de Python, on peut faire une instruction par ligne, et le réflexe de finir les lignes par des points-virgules de façon pertinente s'installera vite) ;
- une erreur qui ne cause pas de souci de syntaxe mais souvent de typage ou

des bugs : l'utilisation de virgules au lieu de points-virgules pour séparer des éléments d'une liste ou d'un tableau... le message d'erreur ou le résultat du test doit cependant faire découvrir rapidement le problème lorsqu'on est devant son ordinateur ;
— même s'il n'est pas nécessaire de mettre chaque argument d'une fonction entre parenthèses, mieux vaut le faire systématiquement pour éviter de déclencher une erreur de typage quand on les oublie alors qu'elles sont nécessaires ;
— et comme dit, on évitera de glisser des mots d'anglais dans une phrase en espagnol, donc `for i in range n` et autres sont à bannir.

1.4 Exercices

Exercice 1

Écrire une fonction qui détermine si une liste est un palindrome, c'est-à-dire si elle se lit de la même façon dans les deux sens.

Exercice 2

Écrire une fonction qui localise le plus petit élément d'un tableau.

Exercice 3

Écrire une fonction qui localise le deuxième plus petit élément d'un tableau.

Exercice 4

Écrire une fonction qui compte le nombre de voyelles d'une chaîne de caractères en minuscules.

Exercice 5

Écrire une fonction qui calcule l'écart maximal en valeur absolue entre deux éléments consécutifs d'un tableau d'entiers.

1.5 Correction des exercices

Exercice 1

```
let palindrome l = l = List.rev l;;
(* Il s’agit essentiellement de réviser la syntaxe de l’égalité,
et de comprendre les implications du fait qu’une liste soit un palindrome.
À ce stade, un algorithme serait pénible à implémenter sans récursion. *)
```

Exercice 2

```
let plus_petit tab =
  let ind = ref 0 in
    for i = 1 to Array.length tab - 1 do
      if tab.(i) < tab.(!ind) then ind := i
    done;
  !ind;;
```

Exercice 3

```
let deuxieme_plus_petit tab =
  let ind1 = ref (if tab.(0) < tab.(1) then 0 else 1) in
  let ind2 = ref (1 - !ind1) (* astuce *) in
    for i = 2 to Array.length tab - 1 do
      if tab.(i) < tab.(!ind1) then (ind2 := !ind1; ind1 := i)
      else if tab.(i) < tab.(!ind2) then ind2 := i
    done;
  !ind2;;
```

Exercice 4

```
let compte_voyelles s =
  let total = ref 0 in
    for i = 0 to String.length s - 1 do
      if List.mem s.[i] ['a'; 'e'; 'i'; 'o'; 'u'; 'y'] then incr total
    done;
  !total;;
```

Exercice 5

```
let ecart_maximal tab =
  let n = Array.length tab in
    if n < 2 then failwith "Tableau trop court";
    let maxi = ref (abs (tab.(1) - tab.(0))) in
      for i = 1 to n - 2 do
        let ecart = (abs (tab.(i+1) - tab.(i))) in
          if ecart > !maxi then maxi := ecart
      done;
    !maxi;;
```

Chapitre 2

Méthodes de programmation

2.1 Introduction [1]

En programmation, l'analyse descendante consiste à partir d'un problème pour lequel on est censé écrire un programme, à chercher à décomposer le problème en sous-problèmes jusqu'à tomber sur des problèmes élémentaires pour lesquels on sait écrire des programmes. Combiner les sous-programmes permet alors d'obtenir notre programme.

À titre d'exemple, imaginons qu'on ait à déterminer le point le plus proche de l'origine d'un plan parmi une liste de couples de flottants. La première chose à faire est de décomposer le problème en (i) « le plus … » donc une recherche d'optimum, (ii) « proche de l'origine d'un plan » donc un calcul de distance.

À ce moment, il est tout à fait avisé d'écrire à part une fonction de calcul de la distance réutilisable dans d'autres cas, et donc ne pas utiliser de fonction locale ou de formule toute faite (qui pour d'autres exemples peut paraître obscure au relecteur), c'est de la **programmation modulaire** [2].

1. Le lecteur se réfèrera préalablement au chapitre d'informatique pour tous sur l'algorithmique et la programmation, notamment les sections concernant les preuves de terminaison et de correction et la complexité, à voir indépendamment du langage.
2. et c'est le bien !

D'autres exemples élémentaires sont le calcul de la somme de deux fractions données en tant que couples d'entiers (on calcule d'abord le produit des dénominateurs, puis on additionne les numérateurs, puis on simplifie, faisant appel au calcul du PGCD), et le calcul du PPCM de deux entiers (on se sert d'une propriété arithmétique liant deux nombres, leur PGCD et leur PPCM, puis on réutilise la fonction PGCD de l'exemple précédent[3] OU on écrit une fonction annexe retournant la décomposition en facteurs premiers d'un nombre et on étudie les deux décompositions).

2.2 Récursivité

Écrire un programme récursif, c'est définir un objet (souvent une fonction, en pratique) en faisant, directement ou indirectement, appel à lui-même[4].

Pour illustrer ceci, on peut considérer que la factorielle de `n` (entier naturel) s'obtient en multipliant de manière itérative tous les entiers de `1` à `n` mais aussi que la factorielle de `0` est `1` et que la factorielle de `n` (entier naturel non nul) est `n` fois la factorielle de `n-1` (en notant qu'on introduit des cas de base assez intuitivement).

Comme on l'a déjà vu, utiliser en Caml un objet alors qu'il n'existe pas déclenche une erreur, et écrire une déclaration d'un objet dépendant de sa version précédente occulte celle-ci. Ainsi, le code suivant provoquera une erreur (Constater ceci de visu !) :

```
let fact n =
if n < 0 then failwith "On veut un nombre positif"
else if n = 0 then 1
else n * fact (n-1);;
```

En fait, il faut préciser à Caml que la fonction qu'on est en train de définir est récursive, ce qui se fait en ajoutant juste après `let` le mot-clé `rec`.

On notera que, comme annoncé en début de section, même si les fonctions sont majoritaires parmi les objets récursifs utilisés, Caml accepte tout à fait qu'on écrive `let rec l = 1::2::l;;` (attention à ne pas demander sa longueur).

La récursivité croisée (appels potentiellement indirects) nécessite une définition si-

3. Comme cela tombe bien !
4. récursivité (n.f.) : caractère de ce qui est récursif ; récursif (adj.) : présentant de la récursivité.

multanée des fonctions qui s'appellent mutuellement. Ainsi, on écrira :

```
let rec tic () = print_string "Tic !"; tac ()
    and tac () = print_string "Tac !"; tic ();;
```

Lorsqu'on appelle une fonction récursive, les instructions sont comme mises sur plusieurs niveaux (et on espère que chaque appel de la fonction procède à un seul appel récursif pour éviter l'explosion de la complexité), ce qui permet de définir la notion de pile d'appels.

Puisqu'en plein milieu d'un appel de fonction un nouvel appel de la fonction vient s'ajouter, le reste des instructions de la fonction doit être mis en attente, puisque tout le nouvel appel doit être exécuté prioritairement.

Tout se passe comme si les appels étaient mis dans une pile, le dernier arrivé étant traité le premier, ce qui nécessite un espace mémoire important dans la plupart des cas[5], alors même que la complexité en temps n'est parfois pas affectée.

On remarquera que la plupart du temps, le choix est difficile entre écrire un programme itératif et écrire un programme récursif, car bien que dans le premier cas on évite les coûts supplémentaires présentés ci-avant, la facilité d'écriture des programmes récursifs leur confère un intérêt certain.

Prouver la terminaison d'un programme récursif nécessite une induction permettant de garantir qu'un cas de base (donc sans appel récursif[6]) est forcément atteint.

La majorité des programmes récursifs permettent d'introduire l'équivalent des variants dans la mesure où il suffit d'exhiber une expression s'évaluant en un entier positif[7] dépendant des variables du programme et telle que les appels successifs (imbriqués) du programme se fassent avec des valeurs strictement décroissantes de l'expression.

5. La notion hors-programme de récursivité terminale indique dans quels cas on peut échapper à ce souci, sachant que Caml optimise des programmes dits récursifs terminaux (alors que Python ne le fait pas).

6. ... donc il en faut ! (Ou alors des mécanismes telles les exceptions permettent de ne plus faire d'appel, mais ce n'est pas souvent propre.)

7. Il suffit de disposer d'un ensemble bien fondé, en pratique.

Pareillement, les preuves de correction nécessitent d'utiliser des invariants, qui ne sont certes pas appelé invariants de boucle, mais dont le fonctionnement est similaire.

La complexité d'un programme récursif s'exprime ordinairement en nombre d'appels récursifs, ce qui permet de déduire la complexité asymptotique en tenant compte du coût de traitement de chaque appel (souvent indépendant des entrées ou qu'on peut ramener au pire cas sans changer la complexité asymptotique).

La méthode classique consiste à analyser la structure du programme et à établir une formule de récurrence sur les coûts, de la forme $c_n = a_1 c_{m_1} + \cdots + a_k c_{m_k} + O(f(n))$, où les m_i sont strictement inférieurs à n (histoire de ne pas avoir de souci de terminaison ou de formule chaotique auxquelles je doute que quiconque souhaite avoir affaire) et f est une fonction représentant le coût de traitement de chaque appel.

La forme simplifiée (elle aussi majoritaire) est $c_n = ac_{n-1} + f(n)$ (où $a \geq 1$ quasi systématiquement), qui s'apparente à une suite définie par une relation de récurrence... et qui permet par un raisonnement sur de telles suites de déduire c_n.

Pour l'exemple de la factorielle, on a donc $c_n = c_{n-1} + \mathcal{O}(1)$, d'où $c_n = \mathcal{O}(n)$.

Un exemple plus élaboré est le problème des tours de Hanoï, pour lequel on a cette fois $c_n = 2c_{n-1} + \mathcal{O}(1)$, donnant $c_n = \mathcal{O}(2^n)$ (complexité exponentielle).

Les fonctions récursives (ou faisant appel à des sous-fonctions récursives, ce qui est souvent plus agréable) sont particulièrement adaptées au traitement des listes, par exemple :

```
let maxilist l = match l with
|[] -> failwith "Liste vide"
|(a::q) -> begin let rec maxi aa ll = match ll with
    |[] -> aa
    |(bb::qq) -> maxi (max aa bb) qq
    in maxi a q end;;

let rec miroir l = match l with
|[] -> []
|(a::q) -> (miroir q)@[a];; (* BEURK ! *)
```

```
let miroir_mieux l =
    let rec miroir_aux accu ll = match ll with
    |[] -> accu
    |(a::q) -> miroir_aux (a::accu) q
    in miroir_aux [] l;;

let rec my_map f l = match l with
|[] -> []
|(a::q) -> f(a)::(my_map f q);;

let rec range (deb, fin, pas) =
    if pas = 0 then failwith "Pas nul"
    else if (deb >= fin) == (pas > 0) then []
    else deb::(range (deb+pas, fin, pas));;
```

On notera le lien avec l'analyse descendante : quand on écrit un programme récursif, on traite souvent les cas de base en premier et on cherche comment organiser les appels ensuite.

2.3 Diviser pour régner

Le paradigme « diviser pour régner » (DPR, en abrégé) permet d'écrire des programmes de complexité très bonne par rapport à leur équivalent naïf.

Il consiste à diviser un problème en sous-problèmes identiques [8], qu'on résout encore une fois en les divisant jusqu'à arriver à des problèmes élémentaires, puis on combine les solutions de sous-problèmes en une solution globale.

La dichotomie est une forme particulière de division pour régner : à chaque fois qu'on divise un intervalle de recherche en deux sous-intervalles, la résolution du problème pour l'un des intervalles de recherche est triviale : on ignore cet intervalle en ne se concentrant que sur l'autre.

Écrire un programme DPR permet donc en particulier de faire des appels récursifs (ce qui ne veut pas dire que le DPR ne s'applique que pour les programmes récursifs !)

8. À ne pas confondre avec l'analyse descendante où les sous-problèmes sont différents, sinon on les traiterait de la même façon.

sur des entrées de taille divisée au lieu de diminuée d'une constante, ce qui limite évidemment de manière drastique le nombre d'appels.

Proposition

Soit un programme dont la complexité est définie par une relation de la forme $c_n = ac_{\frac{n}{b}} + \mathcal{O}(n^\alpha)$*. Alors :*

— *Si* $\alpha < \log_b(a)$*, alors* $c_n = \Theta(n^{\log_b(a)})$.
— *Si* $\alpha > \log_b(a)$*, alors* $c_n = \Theta(n^\alpha)$.
— *Si* $\alpha = \log_b(a)$*, alors* $c_n = \Theta(n^\alpha \log_b(n))$.

Cette proposition se généralise aux relations de type $c_n = ac_{\frac{n}{b}} + f(n)$, où il s'agit de comparer $n^{\log_b(a)}$ et $f(n)$ en termes de domination.

Pour des exemples de problèmes et programmes faisant intervenir le DPR, voir le TP associé.

2.4 Programmation dynamique

Revenons sur la programmation récursive et ses aléas. Nous avons déjà signalé que la pile d'appels peut engendrer un coût en espace que la programmation itérative ne connaît pas (a priori seulement, car on peut très bien créer dans un programme itératif une pile de travaux encore à réaliser, mais quoi qu'il en soit ce qui est inévitable en itératif le sera en récursif à plus forte raison).

En pratique, un problème plus grave (car il concerne la complexité temporelle) peut se poser, c'est qu'un mauvais algorithme fait calculer plusieurs fois la même chose sans raison.

L'exemple classique est le calcul des termes de la suite de Fibonacci, dont la complexité peut varier de manière spectaculaire suivant l'algorithme employé.

```
let rec fibopourri n =
if n < 0 then invalid_arg "On veut n positif !"
else if n < 2 then n
else fibopourri (n-1) + fibopourri (n-2);;
```

Soit f_n la valeur calculée par `fibopourri n`. On constate pour de petites valeurs que f_4 nécessite de calculer f_3 et f_2, que f_3 nécessite de calculer f_2 et f_1 et que f_2 nécessite de calculer f_1 et f_0, le reste étant des cas de base. Or donc, on a mis deux fois dans la pile d'appels un calcul de f_2 (pour f_3 et f_4), et à chaque fois on a mis dans la pile d'appels un calcul de f_1 et de f_0 ; par ailleurs, on a aussi mis dans la pile d'appels un calcul de f_1 lorsqu'on a demandé la valeur de f_3.

Ainsi donc, la complexité c_n en nombre d'additions du calcul de f_n est la suivante : $c_n = c_{n-1} + c_{n-2} + 1$, avec $c_1 = c_0 = 0$, et la complexité t_n en nombre d'appels récursifs est la suivante : $t_n = t_{n-1} + t_{n-2} + 2$, avec aussi $t_1 = t_0 = 0$, soit dans les deux cas un nombre exponentiel (de l'ordre de f_n, en pratique).

Une version plus pertinente stocke dans une liste ou dans un tableau les données calculées. Le principe dit de **mémoïzation** consiste à retenir les valeurs calculées et n'en calculer de nouvelles que si elles ne sont pas disponibles.

Pour se faciliter la vie et ne pas avoir à écrire un programme qui vérifie la disponibilité, non seulement on retiendra toutes les valeurs utiles par défaut, mais de plus on pourra les calculer dans un ordre pertinent, souvent à partir du cas de base et de façon monotone (approches dites *top-down* et *bottom-up* en anglais, non traduites de manière officielle en français).

Ainsi, les deux optimisations suivantes sont en temps linéaire et en espace respectivement linéaire et constant : [9]

```
let fibolin n =
  let rec fiboaux accu nn = match nn with
  |0 -> List.hd (List.tl accu)
  |i -> fiboaux ((List.hd accu + List.hd (List.tl accu))::accu) (i-1)
  in fiboaux [1;0] n;;

let fibocstt n =
  let rec fiboaux moinsun moinsdeux nn = match nn with
  |0 -> moinsdeux
  |i -> fiboaux (moinsun+moinsdeux) moinsun (i-1)
  in fiboaux 1 0 n;;
```

9. Pour être complet sur le sujet, il est possible de calculer f_n en temps logarithmique en n à l'aide de l'exponentiation rapide de matrices, mais ceci relève plutôt de la section précédente.

Un autre exemple plus élaboré est le calcul d'un coefficient binomial sans utiliser la factorielle. On peut se reposer sur une des deux formules suivantes :
$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ et $\binom{n}{k} = \frac{n}{k} \times \binom{n-1}{k-1}$.

Pour la première formule, le cas de base est $\binom{p}{0} = \binom{p}{p} = 1$ pour tout p. En effet, dès qu'on cherche $\binom{n}{k}$ pour $k < n$, on va utiliser le résultat de $\binom{k}{k}$ et celui de $\binom{n-k}{0}$ dans les chemins extrêmes de calcul.

Pour la deuxième formule, le cas de base est $\binom{p}{0} = 1$ pour tout p. En effet, les deux paramètres sont diminués de 1 à chaque étape, et le premier est normalement supérieur au deuxième.

Pour garantir la terminaison et puisque par convention $\binom{n}{k} = 0$ si l'un des paramètres est négatif ou si $n < k$, on ajoute ceci aux cas de base.

Voici donc une version mauvaise en termes de complexité, puis une version écrite en se servant de la programmation dynamique :

```
let rec newton_add_pourri n k =
if n < 0 || k < 0 || n < k then 0 else if k = 0 || k = n then 1
else newton_add_pourri (n-1) k + newton_add_pourri (n-1) (k-1);;

let newton_add n k =
if n < 0 || k < 0 || n < k then 0
else if k = 0 || k = n then 1 (* gagnons du temps *)
else let ligne = Array.make (k+1) 1 in
  let rec newton_rec m =
    for i = min (m-1) k downto 1 do (* ou une autre récursion *)
      ligne.(i) <- ligne.(i) + ligne.(i-1) done;
    if m = n then ligne.(k) else newton_rec (m+1)
  in newton_rec 1;;
```

On note que contrairement au cas où on souhaiterait toutes les valeurs de $\binom{n}{k}$ pour $0 \leq k \leq n$[10], la formule « multiplicative » est plus efficace ici car elle demande de calculer un nombre linéaire de valeurs et d'en mémoriser une seule, tandis que

10. Là, il faut calculer tout le triangle jusqu'à la ligne en question et additionner est plus rapide que multiplier.

la formule « additive » nécessite de calculer des valeurs formant un « rectangle » dans le triangle de Pascal, soit un nombre quadratique de valeurs à calculer (et un nombre linéaire à mémoriser, car une fois une ligne calculée, les lignes précédentes sont inutiles).

Pour tout dire, selon certains la formule multiplicative ne relève pas tout à fait de la programmation dynamique, car chaque appel récursif n'en engendre à chaque fois qu'un autre.

```
let newton_mul n k =
if n < 0 || k < 0 || n < k then 0
else if k = 0 || k = n then 1
else let rec newton_rec accu m =
  if m = n+1 then accu else newton_rec (accu * m / (m-(n-k))) (m+1)
  in newton_rec 1 (n-k+1);;
```

La programmation dynamique peut se formaliser ainsi : on considère un problème qu'on assimile au calcul de l'image par une fonction f de paramètres $p_1, \ldots, p_n$ (de n'importe quel type).

1. On cherche à établir une **formule de récurrence** (éventuellement donnée) liant $f(p_1, \ldots, p_n)$ à une ou plusieurs (toujours plusieurs en pratique) images par f d'autres paramètres **en garantissant la terminaison** (donc il doit s'agir de sous-problèmes).
2. On résout ces sous-problèmes par **mémoïzation** (donc en stockant tout ce qui peut être utile à la volée, car les sous-problèmes sont indépendants).
3. On **recombine les solutions**.

Une belle illustration de la mémoïzation revient à refaire deux versions « jumelles » en espace linéaire de Fibonacci à l'aide d'un tableau géré comme une variable globale aux appels récursifs et locale à la fonction principale (raffinement : globale tout court, afin de pouvoir calculer plusieurs termes de la suite, mais il faudrait alors potentiellement redimensionner le tableau).

```
let fibodyn n =
  let tab = Array.make (n+1) (-1) in
    tab.(0) <- 0; tab.(1) <- 1;
    let rec aux i =
```

```
      if tab.(i) = -1
        then (aux (i-2); aux(i-1); tab.(i) <- tab.(i-2) + tab.(i-1))
    in aux n; tab.(n);;

let fibodyn2 n =
  let tab = Array.make (n+1) (-1) in
    tab.(0) <- 0; tab.(1) <- 1;
    let rec aux i =
      if tab.(i) = -1
        then (let v = aux(i-2) + aux(i-1) in tab.(i) <- v; v)
      else tab.(i)
    in aux n; tab.(n);;
```

Suivant le même principe, on peut calculer les coefficients binomiaux avec mémoïzation :

```
let coeffbin n k =
  let memoire = Array.make_matrix (n+1) (k+1) (-1) in
  for i = 0 to n do memoire.(i).(0) <- 1 done;
  for j = 1 to k do memoire.(0).(j) <- 0 done;
  let rec aux nn kk =
    if memoire.(nn).(kk) = -1 then (* en particulier nn et kk sont > 0 *)
    (
      aux (nn-1) kk;
      aux (nn-1) (kk-1);
      memoire.(nn).(kk) <- memoire.(nn-1).(kk) + memoire.(nn-1).(kk-1)
    )
  in aux n k; memoire.(n).(k);;
```

Voyons un exemple supplémentaire : le problème du rendu de monnaie. On cherche à utiliser le moins de pièces (ou billets) possibles dans un système monétaire donné pour produire une somme donnée.

Les paramètres sont la somme en question et la liste des valeurs de pièces et billets possibles (sous forme de liste ou en tant qu'arguments individuels), et la fonction f retourne le minimum du nombre de pièces possibles.

Pour les euros [11], on a donc la fonction f, en écrivant l la liste $[0.01, 0.02, \dots, 200, 500]$, donnée par la formule de récurrence : $f(s,l) = 1 + \min\{f(s-v,l) \mid s-v \geq 0, v \in l\}$, avec $f(0,l) = 0$.

En mémorisant (malheureusement) toutes les valeurs, on ne les calcule qu'une fois et la complexité est de l'ordre du produit de la valeur à rendre par le nombre de pièces et billets.

Peut-on faire mieux ? Oui, et cela nous amène à parler d'algorithmes gloutons [12]. Instinctivement, pour rendre une somme du genre 1912 €, on va utiliser autant que possible les plus grosses unités d'abord, soit $500+500+500+200+200+10+2$ pour un total de 7 pièces et billets. Il s'avère que c'est optimal, mais en choisissant un autre système le principe aurait pu être mis en échec : imaginons un pays farfelu où les pièces ont pour valeur 1, 42 et 73. Pour payer 84, si on décide d'utiliser d'abord une pièce de 73, on doit compléter avec onze pièces de 1, alors que deux pièces de 42 faisaient l'affaire. [13]

Ce même principe qu'on utilise pour ce problème est un algorithme dit glouton. Il choisit à chaque étape d'explorer un seul des sous-problèmes, qu'il considère comme optimal (qu'il le soit ou non, mais la tendance est qu'on en approche bien) et il donne la solution en conséquence, sans jamais revenir en arrière.

Pour aller plus loin, un théorème donne une condition suffisante pour qu'un algorithme glouton donne toujours une solution optimale : c'est de travailler sur ce qu'on appelle un matroïde [14], c'est-à-dire un couple (S, I) où S est un ensemble fini et I est un sous-ensemble non vide de 2^S ayant les deux propriétés suivantes :

- Tout sous-ensemble d'un élément de I est dans I
- Tout élément X de I de taille strictement inférieure à un élément Y de I peut se compléter en un élément $X \cup \{y\}$ de I pour au moins un $y \in Y \setminus X$. [15]

11. Lecteur du futur : en 2018, c'était notre monnaie, et on avait des pièces de $0,01$ €, $0,02$ €, $0,05$ €, $0,10$ €, $0,20$ €, $0,50$ €, 1 € et 2 € ainsi que des billets de 5 €, 10 €, 20 €, 50 €, 100 €, 200 € et 500 €.
12. pas vraiment au programme, mais tellement utiles
13. Histoire d'être complet sur le sujet et pour ranimer de bons souvenirs sur ma thèse, le problème de Frobenius (datant de la fin du dix-neuvième siècle) et remis au goût du jour en tant que nombres de McNuggets demande, à unité monétaire farfelue fixée, quelle est la plus grande somme entière qui ne peut pas être payée.
14. Promis, ce n'est pas moi qui ai inventé tous ces termes !
15. En mettant de côté le caractère fini de S, on peut faire un parallèle avec l'algèbre linéaire, où

Ici donc, à s fixé, le matroïde est le couple (S, I) où S est un ensemble formé de $\lfloor s/v \rfloor$ copies de chaque élément v de l (copies pour qu'elles puissent apparaître plusieurs fois) et I est l'ensemble des ensembles de pièces/billets dont la valeur est $\leq s$ et de taille inférieure ou égale à la taille optimale recherchée.

La difficulté demeure dans la détermination des systèmes pour lesquels I vérifie la deuxième propriété du matroïde, et même dans la vérification que c'est bien le cas pour le système monétaire européen actuel… [16]

---

les éléments de I peuvent être assimilées à des familles libres dans un espace vectoriel.

16. Voir aussi le paragraphe « Systèmes canoniques » de l'article « Problème du rendu de monnaie » sur Wikipédia, entre autres.

2.5 Exercices

Exercice 1

Écrire une fonction `dernier l` qui détermine le dernier élément d'une liste.

Exercice 2

Écrire une fonction `puiss2 n` qui détermine la liste des puissances de deux dans l'ordre croissant depuis `1` jusqu'à deux puissance `n`.

Exercice 3

Écrire une fonction `fusion l1 l2` qui prend en entrée deux listes croissantes `l1` et `l2` et qui retourne la fusion croissante de ces deux listes. Cette fonction est le cœur de l'algorithme de tri fusion.

Exercice 4

Écrire une fonction `facteurs n` qui détermine la liste des facteurs premiers de `n` dans l'ordre croissant et avec autant d'occurrences de chaque facteur premier que l'ordre de multiplicité associé.

Exercice 5

Écrire une fonction `fibolog n` qui calcule avec un nombre logarithmique d'opérations arithmétiques sur les entiers le terme d'indice `n` de la suite de Fibonacci.

2.6 Correction des exercices

Exercice 1

```
let rec dernier l = match l with
| [] -> failwith "Liste vide" (* jamais rencontré sauf si l est vide au début *)
| [elt] -> elt
| _::q -> dernier q;;
```

Exercice 2

```
let puiss2 n =
  let rec aux i deuxpuissi l =
    if i = n then List.rev l else aux (i+1) (2*deuxpuissi) (deuxpuissi::l)
  in aux 0 2 [1];;
```

Exercice 3

```
let rec fusion l1 l2 = match l1, l2 with
| [], l2 -> l2
| l1, [] -> l1
| a::q, b::qq -> if a < b then a::(fusion q l2) else b::(fusion l1 qq);;
```

Exercice 4

```
let facteurs n =
  let rec facteurs accu i n =
    if i > n then List.rev accu
    else if n mod i = 0 then facteurs (i::accu) i (n/i)
    else facteurs accu (i+1) n
  in facteurs [] 2 n;;
```

Exercice 5

```
let multmat22 a b =
  let reponse = Array.make_matrix 2 2 0 in
    for i = 0 to 1 do
```

```
      for j = 0 to 1 do
        for k = 0 to 1 do
          reponse.(i).(j) <- reponse.(i).(j) + a.(i).(k) * b.(k).(j)
        done
      done
    done;
  reponse;;

let fibolog n =
  let mat_puissance = [|[|1; 1|]; [|1; 0|]|]
  and mat_travail = [|[|1; 0|]; [|0; 1|]|] and i = ref n in
    while !i > 0 do
      if !i mod 2 = 1 then
      (
        let nouv_mat = multmat22 mat_travail mat_puissance in
          for i = 0 to 1 do
            for j = 0 to 1 do
              mat_travail.(i).(j) <- nouv_mat.(i).(j) (* éviter le shadowing *)
            done
          done
      );
      let mat_puissance_carre = multmat22 mat_puissance mat_puissance in
        for i = 0 to 1 do
          for j = 0 to 1 do
            mat_puissance.(i).(j) <- mat_puissance_carre.(i).(j)
          done
        done;
      i := !i / 2
    done; mat_travail.(0).(1);;
```

Chapitre 3

Structures de données et algorithmes

3.1 Introduction

Une structure de données abstraite, en informatique, se résume à un type muni d'opérations. En quelque sorte, il s'agit d'un modèle prévu pour rassembler des données avec des spécifications prévues par celui qui la conçoit (et par celui qui choisit de l'utiliser), ce qui est comparable à un cahier des charges.

L'épithète « abstraite » pour qualifier ce modèle signifie en particulier qu'on peut imaginer toutes sortes de fonctionnalités pour la structure de données, mais il faut tout de même qu'on puisse effectivement se servir du modèle qu'on conçoit.

À titre de comparaison, une machine de Turing est un ordinateur idéal, mais qui ne s'éloigne de la réalité que par le fait qu'on suppose sa mémoire illimitée, et en physique, assimiler une balle lancée à un point qui se déplace sans frottements (ou avec des frottements calculés selon une formule simplifiée) donne tout de même un aperçu de la trajectoire réelle.

Ainsi, une structure de données abstraite efficace est une structure de données abstraite à laquelle on trouve une application rentabilisant le fait de laisser de côté les structures déjà existantes et plus simples, et pour laquelle on trouve une implémentation en une structure de données concrète.

Par exemple, considérons comme structures des ensembles de données issues d'un ensemble ordonné (nombres, n-uplets de nombres muni d'un ordre quelconque, chaînes de caractères, ...), avec comme opérations l'accès immédiat au n-ième plus grand élément, l'insertion, la suppression et la modification.

Cette structure s'implémente en un tableau trié (par exemple). On pourrait imposer dans la structure de données abstraite une complexité pour les opérations, au risque de rendre difficile voire impossible de réaliser une implémentation (par exemple forcer les opérations à avoir une complexité temporelle constante est peine perdue).

On distingue les structures de données persistantes (ou immutables) des structures de données impératives (ou modifiables). Appeler les premières « immutables » ne doit pas créer de confusion : on rappelle par exemple que les entiers, les chaînes de caractères et d'autres objets sont immutables en Python ; de même, en Caml, la notion de variable étant abusive, les structures immutables sont nombreuses.

En pratique, une structure immutable va conserver une sorte d'historique de ses valeurs (on peut mentionner le cas des variables entières dans de nombreux langages qui, lorsqu'elles sont modifiées, sont simplement redirigées vers une case mémoire contenant leur nouvelle valeur, la case précédente gardant l'ancienne valeur).

Cela s'imagine assez aisément dans le cas des listes en Caml (dites listes [simplement] chaînées, puisqu'une liste est vide ou un élément qui s'enchaîne avec une autre liste [1]), car si on modifie ou retire l'élément de tête, cet ancien élément peut encore s'enchaîner avec le reste de la liste, mais plus rien ne pointe sur lui-même [2].

Quant aux structures impératives, comme leur nom l'indique, ce sont aussi celles qui sont les plus adaptées à la programmation impérative : autant le répéter, ce sont par exemple les tableaux (et les chaînes de caractères aussi, même en retirant le caractère mutable) en Caml.

Bien entendu, une structure de données abstraite peut être implémentée en diverses structures de données concrètes (ce qu'on verra sur les deux années d'enseignement de l'option informatique), en utilisant éventuellement des structures de données déjà existantes avec quelques aménagements.

1. Les listes doublement chaînées, quant à elles, ont leurs éléments qui s'enchaînent avec les listes des prédécesseurs et des successeurs

2. Et le ramasse-miettes menace !

3.2 Exemples de structures de données abstraites

Les listes et tableaux (ainsi que les chaînes de caractères, qui ne sont moralement que des tableaux dont les éléments sont nécessairement des caractères, ce qui permet d'ajouter des opérations spécifiques) sont déjà plus ou moins connus, au point de ne pas considérer leur version abstraite, et on s'attardera donc sur des structures un peu plus complexes.

L'objectif de cette section, et plus généralement de ce chapitre, est de présenter différents outils pour une meilleure programmation : il s'avère que le choix de la structure de données à employer peut être décisif pour l'aisance avec laquelle on réalisera des programmes, et donc ce choix a tout intérêt à être fait, et à être fait au début.

3.2.1 Les piles

Une pile est une structure de données que l'on qualifie de *LIFO* (pour `Last In, First Out`, c'est-à-dire que le dernier élément ajouté sera le premier retiré). Cette notion d'accès au dernier élément (le sommet de la pile) se retrouve dans les listes chaînées, c'est donc sans surprise que ces dernières constituent la première implémentation qui vient à l'esprit.

Les opérations élémentaires sur une pile sont la création d'une pile vide, l'empilement d'un élément au sommet, le dépilement du sommet (classiquement, la fonction renvoie le sommet et le retire de la pile par effet de bord), l'accès au sommet sans dépilement et le calcul de la taille de la pile. Cette dernière opération peut être remplacée par un test de vacuité de la pile (ceci simplifie l'implémentation si on impose chacune de ces opérations à être en temps constant).

Quelques applications usuelles des piles sont données en TP.

3.2.2 Les files

En parallèle de la pile, la file est une structure de données *FIFO* (pour `First In, First Out`, le premier arrivé est le premier servi).

Les opérations sont sensiblement les mêmes, à ceci près que l'e**nf**ilement se fait en queue de pile et le dé**f**ilement se fait en tête.

Les files s'utilisent notamment lorsqu'on souhaite qu'une tâche ajoutée à la « liste » aient une chance d'être effectuées en temps raisonnable, dans la mesure où on interdit que d'autres tâches prioritaires arrivent en continu.

Deux exemples seront rencontrés en deuxième année où les files sont utilisées pour un traitement sur des structures qui seront vues à ce moment-là.

3.2.3 Les dictionnaires

Un dictionnaire est un ensemble d'associations entre des clés et des valeurs (on peut voir cela comme une fonction d'un ensemble fini dans un peu n'importe quoi, car rien n'est imposé quant aux valeurs). Son nom rappelle les dictionnaires classiques qui associent à des mots (des chaînes de caractères, en quelque sorte) leur(s) définition(s).

Il est important que les clés soient uniques, pour éviter toute redondance, et ceci se fait usuellement par des tables de hachage. L'accès en temps constant à chaque clé y est requis pour que la structure soit utile, et c'est presque le cas en pratique (on parle de coût amorti constant), mais la gestion des conflits (vérifier l'unicité des clés, justement) peut avoir un coût linéaire de temps à autres.

Un tableau est une forme optimisée de dictionnaire, pour laquelle les clés sont les entiers de 0 à un certain n (la taille plus un). La liste des clés se déduit alors de la taille, ce qui simplifie évidemment beaucoup les choses.

Les opérations sur la structure abstraite de dictionnaire sont l'ajout d'un couple (clé, valeur), sa suppression, la mise à jour de la valeur (pour la clé, l'intérêt est moindre, on fera plutôt une suppression et un ajout) et bien entendu la recherche.

Certains langages ont naturellement une structure de tableau associatif, qui implémente efficacement les dictionnaires, par exemple PHP et Java.

3.2.4 Les files de priorité

Une file de priorité est une file dans laquelle les éléments sont munis d'une clé. Ce n'est alors pas l'élément ajouté en premier qui est retiré en premier, mais celui qui a la plus grande priorité, en considérant que par défaut enfiler un élément revient à lui donner une priorité moindre que les éléments de la file.

Pour rendre la structure pertinente, on a donc besoin d'une opération de plus : la modification de la priorité d'un élément (on peut restreindre à l'augmentation, voire à la diminution, suivant les besoins en termes de puissance et de complexité).

3.3 Exemples d'implémentations

3.3.1 Faire une pile avec une liste

Commençons par une implémentation maison des piles, similaire aux listes.

```
type 'a pile = PileVide | Empilement of 'a * 'a pile;;

let creer_pile () = PileVide;;

let est_vide p = p = PileVide;;

let empiler p e = Empilement(e,p);;

let depiler pile = match pile with
|PileVide -> invalid_arg "Pile vide"
|Empilement(e,p) -> (e,p);;
(* Problème de structure, cf. remarques ultérieures *)

let rec taille pile = match pile with
|PileVide -> 0
|Empilement(_,p) -> 1 + taille p;;

let sommet pile = match pile with
|PileVide -> invalid_arg "Pile vide"
|Empilement(e,p) -> e;;
```

La structure de pile étant fondamentalement modifiable, et cette implémentation se prêtant plutôt à la récursion (comme les listes), il y a un souci dans les fonctions `empiler` et `depiler` : il n'y a pas d'effet de bord, alors que c'est ce que l'on souhaiterait.

On peut contourner ce problème en travaillant sur des références de piles (ou alternativement en considérant que les opérations en question travaillent elles-mêmes sur de telles références), le code se brancherait alors ainsi :

```
...
   (* p est ici une référence de liste *)
   p := empiler(e,!p);
...
   let (ee,pp) = depiler(!p) in p := pp; (* utiliser ee *)
...
```

Cependant, en voyant la définition du type, on se rend compte que l'opération d'empilement est plus ou moins une concaténation, d'où :

```
let creer_pile () = [];;

let est_vide p = p = [];;

let taille = List.length;;

let empiler p e = e::p;;

let depiler l = (List.hd l, List.tl l);;

let sommet = List.hd;;
```

La même remarque s'applique à l'utilisation d'une structure persistante pour simuler une structure plutôt impérative.

3.3.2 Faire une pile avec un tableau

Au vu de ce qui précède, la structure de tableau semble plus adaptée pour simuler une pile. Malheureusement, les tableaux ne sont pas redimensionnables en Caml, ce qui nous limitera aux piles de taille majorée (on parle de « capacité » pour qualifier la borne supérieure de cette taille).

Le principe de l'implémentation est le suivant : le premier élément du tableau représentant la pile décrira la taille de celle-ci, de sorte que seuls les éléments suivant dans cette limite seront considérés (l'élément d'indice `1` est alors au fond de la pile, et le sommet, comme on le verra, est à l'indice donné par l'élément d'indice `0`).

On note qu'il serait également possible d'utiliser un couple (entier ou référence d'entier, tableau).

```
let creer_pile c = Array.make (c+1) 0;; (* c est la capacité *)

let est_vide p = p.(0) = 0;;

let empiler p e =
    if p.(0) = Array.length p - 1
    then invalid_arg "Dépassement de capacité"
    else p.(0) <- p.(0) + 1; p.(p.(0)) <- e;;

let depiler p =
    if p.(0) = 0
    then invalid_arg "Pile vide"
    else p.(0) <- p.(0) - 1; p.(p.(0) + 1);;

let taille p = p.(0);;

let sommet p = p.(p.(0));;
```

3.3.3 Faire une file avec deux listes

Comme on l'a vu précédemment, en Caml, on peut réaliser une structure (persistante) de pile avec une liste.

Ainsi, bien qu'algorithmiquement il soit plus concevable de faire une file à l'aide de deux piles, nous allons utiliser ici deux listes. La structure sera évidemment persistante.

Le principe est simple : puisqu'on se limite à la concaténation, l'enfilement sera naturel, mais le défilement nécessite d'accéder au fond de notre liste principale.

Ceci va nous obliger à utiliser une liste d'appui dans laquelle on videra quand il le faudra (exactement aux moments où on a besoin de défiler alors que la liste d'appui est vide) la liste principale. Cette opération ne peut se faire qu'en concaténant les sommets successifs de la liste principale, ce qui revient à inverser l'ordre des éléments.

```
let creer_file () = ([], []);;

let est_file_vide (lp, la) = lp = [] && la = [];;

let enfiler (lp, la) e = (e::lp, la);;

let defiler (lp, la) =
    if est_file_vide (lp, la) then invalid_arg "File vide"
    else if la <> [] then (List.hd la, (lp, List.tl la))
    else let la2 = List.rev lp in (List.hd la2, ([], List.tl la2));;
(* pas d'effet de bord ici non plus donc la signature change *)

let taille (lp, la) = List.length lp + List.length la;;

let tete (lp, la) =
    if est_file_vide (lp, la) then invalid_arg "File vide"
    else if la <> [] then List.hd la
    else List.hd (List.rev lp);;
```

3.3.4 Faire une file avec un tableau

De façon similaire à la réalisation d'une structure de pile avec un tableau, nous allons réaliser une file, en tant que structure modifiable mais de capacité limitée.

Le tableau utilisé ici est circulaire, c'est-à-dire qu'une fois le dernier élément rempli, sous réserve de ne pas dépasser la capacité, on continuera au début du tableau (ceci rend encore plus intéressant le remplacement de la structure par un couple, que nous ne ferons cependant pas ici).

La raison en est simple : enfilement et défilement ne se font pas au même endroit, et donc certaines cases en début de tableau seront « libérées » une fois que leur contenu correspondra à un élément défilé.

Il est cependant hors de question de décaler tous les éléments du tableau à chaque défilement, l'opération serait en temps linéaire et on ne peut pas sacrifier la complexité constante pour limiter la difficulté (très relative) de la programmation.

Cette fois-ci, il faudra donc une information supplémentaire, car la seule taille ne suffit pas : il faut aussi mémoriser la position de la tête ou de la queue de la file, et comme en pratique deux des trois informations permettent de déduire la dernière, on aura le choix de celles qu'on mémorise, ici taille puis queue.

```
let creer_file c = Array.make (c+2) 0;; (* c est la capacité,
au début la taille est 0 et la queue en position 0,
qui est la position impossible *)

let est_file_vide f = f.(0) = 0;;

let taille f = f.(0);;

let enfiler f e =
    if f.(0) = Array.length f - 2
    then invalid_arg "Dépassement de capacité"
    else f.(0) <- f.(0) + 1; f.(1) <- f.(1)+1;
    if f.(1) = 1 || f.(1) = Array.length f
    then f.(1) <- 2;
    f.(f.(1)) <- e;;

let position_tete f =
    let pos = f.(1) - f.(0) + 1 in
    if pos >= 2
    then pos
    else Array.length f - 2 + pos;;
(* cf. transfert, fonction d'appui *)

let defiler f =
    if f.(0) = 0
    then invalid_arg "File vide"
    else let tete = f.(position_tete f) in f.(0) <- f.(0) - 1; tete;;

let queue f = f.(f.(1));;
```

3.3.5 Faire un dictionnaire avec un tableau

Sans entrer dans les détails de la structure de dictionnaire utilisée en pratique dans la plupart des langages, structure qui est optimisée à l'aide de fonctions de hachage, nous allons présenter une structure simple faisant intervenir des tableaux de couples (clé, valeur).[3]

Supposons qu'on choisisse des tableaux triés. Le but ici est de garantir un temps logarithmique pour l'accès (avec ou sans modification de la valeur) et la recherche de la position où insérer une entrée.

La suppression peut alors poser un problème : si on laisse un trou dans le tableau, on ne peut pas se permettre de faire une recherche dichotomique, puisque la direction où poursuivre la recherche serait inconnue à chaque fois qu'on rencontre un trou et nécessiterait un nombre non contrôlé de tests sur des cases voisines.

Puisque de toute façon un décalage est nécessaire lorsqu'on ajoute une entrée, des décalages seront également effectués pour supprimer une entrée.

Reste le problème de la taille fixée d'un tableau en Caml : il faudra là aussi partir sur une capacité limitée, en effaçant ou ignorant les entrées dépassant du nombre d'entrées, également mémorisé pour éviter que, lorsque ce nombre est petit par rapport à la capacité, la recherche se fasse sur une plage indûment trop grande.

Par conséquent, la structure de tableau trié n'est à privilégier que si on n'est presque jamais amené à faire des insertions et suppressions, car alors quitte à avoir un coût linéaire presque à tout moment (il suffit d'insérer ou de supprimer dans la première moitié pour décaler au moins l'autre moitié...), il y a peu d'intérêt à faire une recherche logarithmique.

En effet, si on ne trie pas les données, l'insertion se fait en un coût constant sous certaines conditions (mémoriser la position du dernier élément et considérer un élément supprimé comme définitivement perdu), en s'autorisant à recycler les positions vides en temps linéaire une fois toutes les cases occupées.

La suppression se fait aussi en un coût constant à condition de savoir à quelle position la clé doit être supprimée, avec une solution alternative (que nous retiendrons) : la suppression avec recherche de clé étant linéaire, on peut l'accompagner d'un décalage, rendant l'insertion constante quoi qu'il arrive.[4]

3. En deuxième année, une structure plus efficace faisant intervenir des arbres particuliers sera également présentée.

4. Il est déraisonnable de ne pas mémoriser l'emplacement de la dernière entrée, qui nécessiterait

Les deux choix conduisent aux implémentations suivantes :

```
let cree_dict_trie c = let dict = Array.make (c+1) ("", "")
  in dict.(0) <- ("taille :", "0"); dict;;

exception Trouve of int;;
let position_trie dict cle = let taille = int_of_string (snd dict.(0)) in
  let deb = ref 1 and fin = ref taille in
    try
      while !deb < !fin do
        let milieu = (!deb + !fin) / 2 in
          if fst dict.(milieu) = cle then raise (Trouve milieu)
          else if fst dict.(milieu) > cle then fin := milieu
(* pas - 1 pour des raisons pratiques, mais fin décroît strictement *)
          else deb := milieu + 1
      done; (dict.(!deb) = cle, !deb)
    with Trouve i -> (true, i);;
(* Cette fonction va être utile pour les opérations sur un dictionnaire trié,
cela économisera un nombre conséquent de lignes de code *)

let recherche_trie dict cle =
  let (trouve, pos) = position_trie dict cle in
    if trouve then snd dict.(pos) else failwith "Introuvable";;

let modifie_trie dict cle valeur =
  let (trouve, pos) = position_trie dict cle in
    if trouve then dict.(pos) <- (cle, valeur) else failwith "Introuvable";;

let insere_trie dict (cle, valeur) = let taille = int_of_string (snd dict.(0)) in
  if taille = Array.length dict - 1 then failwith "Capacité dépassée";
  let (trouve, pos) = position_trie dict cle in
    if not trouve then begin
      for i = taille+1 downto pos do dict.(i) <- dict.(i-1) done;
      dict.(pos) <- (cle, valeur);
      dict.(0) <- ("taille :", string_of_int (taille + 1)) end
    else failwith "La clé existe déjà";;

let supprime_trie dict cle = let taille = int_of_string (snd dict.(0)) in
  let (trouve, pos) = position_trie dict cle in
    if trouve then begin
      for i = pos to taille-1 do dict.(i) <- dict.(i+1) done;
(* Si on veut, on ajoute dict.(taille) <- ("", ""). *)
      dict.(0) <- ("taille :", string_of_int (taille - 1)) end
    else failwith "Introuvable";;

let cree_dict c = let dict = Array.make (c+1) ("","")
  in dict.(0) <- ("taille :", "0"); dict;;
```

---

sinon au moins une recherche dichotomique pour être obtenu.

```
let position dict cle =
  let i = ref 1 and trouve = ref false in
    while not !trouve && !i < Array.length dict do incr i done;
    if !i = Array.length dict then failwith "Introuvable" else !i;;

let recherche dict cle = snd dict.(position dict cle);;
(* Une éventuelle erreur est remontée. *)

let modifie dict cle valeur = dict.(position dict cle) <- (cle,valeur);; (* idem *)

let insere dict (cle,valeur) = let taille = int_of_string (snd dict.(0)) in
  if taille = Array.length dict - 1 then failwith "Capacité dépassée";
  dict.(taille+1) <- (cle,valeur);
  dict.(0) <- ("taille :",string_of_int (taille + 1));;
(* Malheureusement, on ne vérifie pas l'absence de doublons. *)

let supprime dict cle = let taille = int_of_string (snd dict.(0)) in
  let pos = position dict cle in
  dict.(pos) <- dict.(taille); dict.(taille) <- ("", "");
(* pas besoin de tout décaler, ce n'est pas trié *)
  dict.(0) <- ("taille :",string_of_int (taille - 1));;
```

3.4 La structure d'arbre

On a vu qu'une liste pouvait être considérée comme un élément muni d'un pointeur sur son successeur dans la liste. Que se passe-t-il s'il peut y avoir plusieurs successeurs menant à des sous-listes bien distinctes ? Ce branchement donne, comme le vocabulaire peut le laisser deviner… un arbre.

Les arbres vus en informatique ne partagent pas beaucoup de caractéristiques avec leurs homonymes bien connus, en particulier ils sont dessinés à l'envers, la racine (unique) étant tout en haut et les feuilles en bas.

Commençons par du vocabulaire, justement : un **nœud** de l'arbre est un de ses éléments, avec les fameux pointeurs vers d'autres nœuds appelés ses fils (et dans certaines implémentations un pointeur d'un nœud `x` vers son **père**, c'est-à-dire l'unique nœud ayant `x` parmi ses fils) ; un nœud sans fils est appelé une **feuille**, mais cette distinction n'est pas obligatoire, car on peut considérer qu'une feuille a pour fils un ou plusieurs arbres vides, notamment quand on impose l'**arité** d'un arbre, c'est-à-dire le nombre exact de fils (éventuellement avec des éléments signalant le vide) de chaque nœud interne.

La **racine** est l'unique nœud sans père, en insistant sur le fait qu'il est impossible qu'un nœud ait plusieurs pères, de même qu'il est impossible que l'arbre soit non connexe [5], c'est-à-dire qu'il soit formé de deux arbres distincts sans que le moindre nœud de l'un ait pour père un nœud de l'autre et vice-versa.

Une **branche** est une suite de nœuds telle que chaque nœud soit un fils du précédent, la plupart des conventions imposant que le premier nœud soit la racine et le dernier une feuille ou au moins un nœud ayant un fils vide.

La **taille** d'un arbre est le nombre de ses nœuds (feuilles incluses), la **profondeur** d'un nœud est la longueur (en nombre de nœuds ou ce nombre moins un, suivant les conventions) de la plus grande branche finissant sur ce nœud (et partant évidemment de la racine, mais en n'obligeant évidemment pas qu'une branche finisse sur une feuille pour que la définition ait toujours un sens) et la **hauteur** d'un arbre est la profondeur maximale d'un nœud, c'est-à-dire la longueur maximale d'une branche.

Les arbres les plus classiques sont les arbres **binaires**, c'est-à-dire dont chaque nœud a au plus deux fils. Signalons que les nœuds des arbres véhiculent une information (il faut bien que la structure ait un intérêt), qui prend la forme d'un objet d'un certain type. Dans certains cas [6], les feuilles véhiculent une autre information, et l'objet peut alors être d'un type différent, ce qui amène aux définitions suivantes des arbres (binaires ou non) en Caml :

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

Quelques arbres particuliers : un **peigne droit** (gauche analogue) est un arbre binaire dont le fils gauche de tous les nœuds est l'arbre vide. [7] Si un fils indifférent est vide pour chaque nœud, l'arbre est simplement une branche.

Un arbre binaire **complet** est un arbre binaire de hauteur notée n dont tous les

5. On retrouvera ce terme plus en détail dans le chapitre sur les graphes en deuxième année.
6. voir par exemple le codage de Huffman
7. Selon certains, un peigne droit est soit vide soit un arbre binaire dont la racine a un fils gauche sans fils et un fils droit racine d'un peigne droit.

nœuds de profondeur inférieure ou égale à $n-1$ n'ont pas de fils vide. Évidemment, la définition implique que tous les nœuds de profondeur n n'ont que des fils vides.

De nombreux autres arbres particuliers existent dans la littérature, mais aucun n'est au programme de première année...

Proposition

La taille d'un arbre binaire complet de hauteur n est $2^n - 1$.

La preuve se fait par une récurrence simple : le nombre de nœuds de profondeur k pour $1 \leq k \leq n$ est 2^{k-1} car il y a deux fils par nœud à la profondeur $k-1$ (si $k > 1$) et un seul nœud à la profondeur 1 : la racine.

Proposition

Soit un arbre binaire dont chaque nœud a aucun ou deux fils vides (on appelle cela un arbre binaire strict). Le nombre de feuilles de cet arbre est le nombre de ses nœuds internes plus un.

La preuve se fait par induction sur la structure de l'arbre : c'est vrai sur l'arbre restreint à sa racine (qui est une feuille), et si c'est vrai pour deux arbres g (ayant x_g nœuds internes et x_g+1 feuilles) et d (analogue), c'est vrai pour un arbre formé d'une racine ayant g pour fils gauche et d pour fils droit, car alors le nombre de nœuds internes sera $x_g + x_d + 1$ (+1 pour la racine) et le nombre de feuilles $x_g + 1 + x_d + 1$.

3.5 Induction structurelle

Le principe de cette section est de fournir un outil de preuve en informatique plus puissant que les récurrences, et qui les généralise.

Principe : Soit une propriété que l'on souhaite prouver pour tous les objets d'un type défini récursivement à partir de cas de base et de constructeurs faisant intervenir un ou plusieurs sous-objet(s) du même type. Alors il suffit de prouver que la propriété est vérifiée pour tous les cas de base et pour toutes les constructions. C'est ce qu'on appelle faire une induction structurelle.

Exemples :

- La récurrence est une induction sur la structure suivante pour $\mathbb{N}$: un entier naturel est 0 ou le successeur d'un entier naturel.
- La récurrence d'ordre k est une induction sur la structure suivante pour $\mathbb{N}$: un entier naturel est l'un des nombres de 0 à $k-1$ ou le successeur du plus grand élément d'une suite de k entiers consécutifs.
- La récurrence forte s'obtient de la même façon, et on peut même considérer une structure dont l'élément de base est 0 et d'autres éléments s'obtiennent en prenant un objet et en ajoutant son maximum plus un (un objet contient donc tous les nombres entre 0 et son maximum... par induction structurelle !).
- Une preuve par induction structurelle sur une liste chaînée se fait en traitant le cas des listes vides et en prouvant que si une propriété est vraie pour une liste elle le reste lorsqu'on ajoute un élément quelconque.
- Une preuve par induction structurelle sur un arbre binaire se fait en traitant le cas des arbres vides ou des feuilles, suivant la structure, et en prouvant que si une propriété est vraie pour deux arbres binaires elle le reste pour un arbre construit à partir d'une valeur quelconque mise à la racine d'un arbre dont les fils sont les deux arbres binaires considérés.

Bien entendu, les inductions structurelles sont particulièrement adaptées aux preuves de programmes récursifs.

Un choix alternatif moins esthétique à l'induction structurelle peut être de faire une récurrence sur la taille des objets.

3.6 Exercices

Exercice 1

Écrire une fonction déterminant la somme des étiquettes d'un arbre binaire d'entiers.

Exercice 2

Implémenter une structure de pile de capacité limitée en tant que couple (référence d'entier, tableau), la référence d'entier précisant la taille de la pile à tout moment. Pour la création de la pile, on pourra donner un élément de base qui permettra d'avoir un type quelconque (autre choix : utiliser un type `option`).

Exercice 3

Implémenter une structure de liste doublement chaînée, avec les opérations d'ajout et de retrait d'un élément aux deux bouts.

Exercice 4

Implémenter une structure de dictionnaire en tant que type `(string * string list) array`, le tableau devant être trié selon les clés (avec un tri croissant selon l'ordre lexicographique), avec les opérations d'ajout d'une entrée tel que si la clé existe déjà, une nouvelle définition est ajoutée, de consultation des définitions d'une entrée et de suppression d'une entrée entière ou seulement d'une définition dont l'indice dans la liste est précisé.

3.7 Correction des exercices

Exercice 1

```
type intarbin = Vide | Noeud of intarbin * int * intarbin;;
(* Version sans feuille ici, qui sera souvent privilégiée
quand on pourra se le permettre. *)

let rec somme a = match a with
| Vide -> 0
| Noeud(g, n, d) -> somme g + n + somme d;;
```

Exercice 2

```
let creer_pile c base = (ref 0, Array.make c base);;

let est_vide (t, _) = !t = 0;;

let empiler (t, p) e =
    if !t = Array.length p - 1
      then invalid_arg "Dépassement de capacité";
    incr t; p.(!t) <- e;;

let depiler (t, p) =
    if !t = 0
      then invalid_arg "Pile vide";
    decr t; p.(!t + 1);;

let taille (t, p) = !t;;

let sommet (t, p) = p.(!t);;
```

Exercice 3

Une information redondante va être fournie ici, afin d'avoir un accès immédiat à la taille et d'essayer de faire le moins souvent possible des renversements.

```
type 'a ldc = {mutable taille : int;
```

```
mutable debut : 'a list; mutable fin : 'a list};;

let cree_ldc () = {taille = 0; debut = []; fin = []};;

let remettre taille reste debut_ou_fin =
  let rec separation nombre accu l =
    if nombre = 0 then List.rev accu, List.rev l
    else separation (nombre - 1) (List.hd l::accu) (List.tl l)
  in let l1, l2 = separation (taille / 2) [] reste
  in if debut_ou_fin then l1, l2 else l2, l1;;

let ajouter_tete e l = l.debut <- e::l.debut; l.taille <- l.taille + 1;;

let ajouter_queue e l = l.fin <- e::l.fin; l.taille <- l.taille + 1;;

let retirer_tete l =
  let t = l.taille in
  if t = 0 then failwith "Liste vide";
  l.taille <- t - 1;
  match l.debut with
  | a::q -> l.debut <- q; a
  | _ -> let d, f = remettre t l.fin false in
    l.debut <- List.tl d; l.fin <- f; List.hd d;;

let retirer_queue l =
  let t = l.taille in
  if t = 0 then failwith "Liste vide";
  l.taille <- t - 1;
  match l.fin with
  | a::q -> l.fin <- q; a
  | _ -> let d, f = remettre t l.debut true in
    l.debut <- d; l.fin <- List.tl f; List.hd f;;
```

Une version non retenue revient à utiliser une liste qu'on renverse dès qu'on a besoin d'accéder à l'élément au fond, ce qui occasionne trop souvent des coûts linéaires.

Exercice 4

```
let cree_dict_trie c = let dict = Array.make (c+1) ("", [])
  in dict.(0) <- ("0", []); dict;;
(* On aura compris que le premier est la taille, donc autant simplifier *)

exception Trouve of int;;
let position_trie dict cle = let taille = int_of_string (fst dict.(0)) in
  let deb = ref 1 and fin = ref taille in
    try
      while !deb < !fin do
        let milieu = (!deb + !fin) / 2 in
          if fst dict.(milieu) = cle then raise (Trouve milieu)
          else if fst dict.(milieu) > cle then fin := milieu
          else deb := milieu + 1
      done; (dict.(!deb) = cle, !deb)
    with Trouve i -> (true, i);;

let recherche_trie dict cle =
  let (trouve, pos) = position_trie dict cle in
    if trouve then snd dict.(pos) else failwith "Introuvable";;

let insere_trie dict (cle, valeur) =
  let taille = int_of_string (fst dict.(0)) in
    if taille = Array.length dict - 1 then failwith "Capacité dépassée";
    let (trouve, pos) = position_trie dict cle in
    if not trouve then begin
      for i = taille+1 downto pos do dict.(i) <- dict.(i-1) done;
      dict.(pos) <- (cle, [valeur]);
      dict.(0) <- (string_of_int (taille + 1), []) end
    else dict.(pos) <- (cle, valeur::(snd dict.(pos)));;
(* Deuxième cas : une définition s'ajoute,
c'est une différence par rapport à la version du cours. *)

let supprime_trie dict cle num =
(* spécification : num = -1 -> supprime tout,
num = 0 -> retire la tête, puis ainsi de suite *)
  let taille = int_of_string (fst dict.(0)) in
```

```
let (trouve, pos) = position_trie dict cle in
  if not trouve then failwith "Introuvable";
  if num = -1 || List.length (snd dict.(pos)) = 1 then begin
    for i = pos to taille-1 do dict.(i) <- dict.(i+1) done;
    dict.(0) <- (string_of_int (taille - 1), []) end
  else let rec nouvelles_def indice liste =
    if indice = 0 then List.tl liste
    else (List.hd liste)::(nouvelles_def (indice-1) (List.tl liste))
  in dict.(pos) <- (cle, nouvelles_def num (snd dict.(pos)));;
```

Deuxième partie

Travaux pratiques

TP 1 : Prise en main basique de Caml

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

Chaque ligne se termine par deux points-virgules, et l'éditeur de base de Caml 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.

Les commentaires à la suite du code sont les passages entourés de parenthèses étoilées.

```
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 *)

[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 *)
```

```
[|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 *)

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

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;; (* 1764, inoubliable ! *)
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;;
```

```
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;;
```

Terminons par quelques structures de contrôle simples, avant de les combiner avec d'autres éléments de syntaxe dans le TP suivant.

```
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 42 else 4.2;; (* interdit *)

if true then false;; (* interdit pour la même raison *)

if true then print_string "Autorisé";;

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;;
```

Exercice 1 : Créer un tableau de taille 42 dont les éléments sont les premiers entiers naturels.

Exercice 2 : Créer une chaîne de caractères quelconque de taille au moins 10. Remplacer tous les caractères d'indice pair par des espaces. Si cela ne marche pas, créer le tableau de caractères correspondant à la chaîne, faire le remplacement et récupérer la chaîne correspondante.

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

Exercice 4 : Créer une chaîne de caractères quelconque de taille au moins 10. La modifier de sorte que le premier caractère soit envoyé à la fin, tout le reste subissant donc un décalage. Même adaptation qu'à l'exercice 2 si cela ne marche pas.

TP 2 : Prise en main avancée de Caml

Commençons par des fonctions.

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

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

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

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

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 ? *)
```

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

```
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 taille_liste l =
  let taille = ref 0 and liste = ref l in
    while !liste <> [] do
      taille := !taille + 1; (* ou incr taille *)
      liste := List.tl !liste
    done; !taille;;
(* à ne plus faire une fois les récursions étudiées *)
```

```
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;;
```

Terminons par des fonctions récursives.

```
let somme_tableau_rec tab =
  let rec somme_tableau_aux i =
    if i = Array.length tab then 0
    else tab.(i) + somme_tableau_aux (i+1)
  in somme_tableau_aux 0;;

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

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

```
let rec part_ent_log2_rec nombre =
  if nombre <= 0 then failwith "Boum !";
  if nombre = 1 then 0
  else 1 + (part_ent_log2_rec (nombre / 2));;

let est_dans_tableau_rec element tab =
  let rec aux i =
    if i = Array.length tab then false
    else tab.(i) = element || aux (i+1)
  in aux 0;;
```

Et histoire d'aller un peu plus loin, ces fonctions récursives peuvent être écrites de façon plus agréable :

```
let somme_tableau_rec tab =
  let rec somme_tableau_aux buff i =
    if i = Array.length tab then buff
    else somme_tableau_aux (buff + tab.(i)) (i + 1)
  in somme_tableau_aux 0 0;;

let taille_liste_rec l =
  let rec taille_liste_rec_aux buff liste = match liste with
  | [] -> buff
  | _::q -> taille_liste_rec_aux (buff + 1) q
  in taille_liste_rec_aux 0 l;;

let pow_rec valeur exposant =
  let rec pow_rec_aux buff expo = match expo with
  | 0 -> buff
  | i -> pow_rec_aux (buff * valeur) (i - 1)
  in pow_rec_aux 1 exposant;;

let part_ent_log2_rec nombre =
  if nombre <= 0 then failwith "Boum !";
  let rec part_ent_log2_rec_aux buff n =
    if n > nombre then buff - 1
    else part_ent_log2_rec_aux (buff + 1) (n * 2)
  in part_ent_log2_rec_aux 0 1;;
```

```
let est_dans_tableau_rec element tab =
  let rec est_dans_tableau_rec_aux buff i =
    if i = Array.length tab then buff
    else est_dans_tableau_rec_aux (buff || tab.(i) = element) (i+1)
  in est_dans_tableau_rec_aux false 0;;
```

Exercice 1 : 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 2 : Écrire une fonction qui prend en entrée un tableau et qui retourne son plus grand élément. La fonction peut être itérative ou récursive.

Exercice 3 : Écrire une fonction qui prend en entrée une liste de flottants et qui retourne la somme de ses éléments. La fonction peut être itérative ou récursive, mais cette dernière version est préférable. Vérifier que la signature est correcte et faire attention aux erreurs de typage.

Exercice 4 : Écrire une fonction qui prend en entrée un tableau et qui détermine si ses éléments sont dans l'ordre croissant. La fonction peut être itérative ou récursive.

Exercice 5 : Écrire une fonction qui prend en entrée un tableau 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`). La fonction peut être itérative ou récursive.

Exercice 6 : Écrire une version récursive de `print_int_array`.

TP 3 : Types construits

L'un des domaines où les étudiants rencontrent le plus de difficultés est la gestion de types construits. La syntaxe est effectivement assez exotique, et la rigueur habituelle de Caml 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 de raison que Caml s'alarme *)
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, cf. l'ordre lexicographique *)
Fraction(2, 8) = Fraction(1, 4);; (* faux, Caml n'a pas à comprendre le sens caché *)
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[8].

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

8. le retour du TP 8 d'IPT !

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 };;
(* intéressant : le mot-clé val est réservé *)
```

Une main sera considérée comme un tableau de cartes (ne donnant donc pas lieu à la création d'un 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 *)
```

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 4 : Fonctions sur les listes

Le module `List`, issu de la bibliothèque principale (*core library*), contient des fonctions déjà présentées en cours, à savoir `hd`, `tl` et `length`, ainsi que l'opérateur de concaténation `@`.

D'autres fonctions de ce module font l'objet de ce TP :

— `rev` renvoie la version retournée de la liste ;
— `for_all` et `exists` vérifient si tous les éléments (resp. au moins un élément) de la liste en deuxième argument satisfont (resp. satisfait) un prédicat, c'est-à-dire une fonction prenant en argument des objets du type commun des éléments de la liste, en premier argument, et retournant un booléen ;
— `mem` vérifie si le premier argument est un élément de la liste en deuxième argument, et `index` retourne la position de la première occurrence, les indices commençant à zéro, avec l'exception `Not_found` en cas d'échec ;

On notera que les fonctions `union`, `intersect` et `except`, au comportement parfois douteux, ne figurent plus en tant qu'opérations de listes mais en tant qu'opérations sur un type qui implémente la structure d'ensemble.

Exercice 1 : Réécrire toutes ces fonctions (en utilisant la récursivité).

Une importance particulière est accordée aux itérations de fonctions [9] :

— `map : ('a -> 'b) -> 'a list -> 'b list` applique son premier argument aux éléments de son deuxième argument dans l'ordre pour former une liste ;
— `iter : ('a -> unit) -> 'a list -> unit` fait la même chose, mais le premier argument est une fonction renvoyant un `unit` ;
— `fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` applique son premier argument de manière imbriquée à tous les éléments de son deuxième argument conjointement à son troisième argument [10] ;
— `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` est similaire, mais l'ordre est inversé [11].

Exercice 2 : Réécrire également ces quatre fonctions.

9. Malgré le nom, on peut tout de même faire ce genre de fonctions de manière récursive.
10. En clair, `fold_right f [a1; ...; an] b` correspond à `f a1 (f a2 (...(f an b) ...))`.
11. De même, `fold_left f b [a1; ...; an]` correspond à `f (...(f (f b a1) a2) ...) an`.

Exercice 3 : Écrire à l'aide d'itérateurs de listes des fonctions `print_type_list` pour différents types.

Exercice 4 : Écrire les fonctions `for_all`, `exists` et `mem` à l'aide de `fold_left` ou `iter`.

Exercice 5 : Écrire la factorielle à l'aide de `fold_left` ou `iter` en créant la liste sur laquelle faire une itération [12].

12. conseil : ne jamais réécrire la factorielle ainsi

TP 5 : Alea camelus est

ATTENTION : Ce TP ne fera pas l'objet d'une séance. Il peut cependant être fait en guise d'entraînement.

Le module `Random`, issu de la bibliothèque standard (*standard library*), comme signalé, est chargé de base, et le préfixage est bien entendu nécessaire par défaut.

Le générateur aléatoire d'Ocaml a le même inconvénient que celui de Caml light : sans initialisation, chaque session retournera les mêmes valeurs si les mêmes bornes sont demandées.

Ce module contient un peu plus de fonctions que son équivalent de Caml light, mais on se contentera de présenter les principales :
- `int` renvoie un entier pseudo-aléatoire [13] entre `0` (inclus) et son paramètre (exclu). Si le paramètre est strictement négatif, l'entier sera négatif ou nul, et si le paramètre est nul, une exception `Division_by_zero` est soulevée.
- `float` renvoie un flottant pseudo-aléatoire entre `0.` (inclus... pour ce que ça change) et son paramètre (exclu).
- `init` initialise le générateur aléatoire avec pour « graine » (*seed*) son argument (entier), `full_init` fait de même avec un tableau et `self_init` le fait à l'aide de paramètres dépendant du système, donc son argument est `()`.

Exercice 1 : Constater le besoin d'initialiser le générateur pseudo-aléatoire. Comment procéder de préférence dans un programme ? [14]

Exercice 2 : Se renseigner sur internet sur le fonctionnement des générateurs pseudo-aléatoires. En écrire un soi-même.

Exercice 3 : À l'aide de la fonction `Random.float` et d'une bijection d'un intervalle semi-ouvert fini dans un intervalle semi-ouvert infini, écrire une fonction sans argument qui retourne un entier positif ou nul aléatoire. Quels sont les problèmes d'une telle fonction ?

---

13. De toute façon, il n'existe pas de vrai générateur aléatoire, et parler d'informatique quantique à ce stade est à la fois prématuré, non prématuré et $\frac{1}{\sqrt{2}}$ (prématuré + non prématuré).

14. Attention : ne **surtout** pas initialiser le générateur avec un nombre obtenu à l'aide de `Random.int`, pour une raison évidente...

Exercice 4 (assez long) : Faire un programme pour générer des marches aléatoires en dimension 1 ou 2 (voire 3). Si le TP suivant a été fait, réaliser une implémentation graphique.

Exercice 5 (long) : Écrire un programme simulant un jeu de hasard (par exemple la roulette ou le black jack) à la manière d'un casino. Perdre toutes ses économies en le testant.

TP 6 : Graphismes de base

Attention : Ce TP ne fera pas l'objet d'une séance. Il peut cependant être fait en guise d'entraînement.

Dans ce TP, nous aborderons le module graphique d'Ocaml. Ce TP a surtout une vocation ludique, dans la mesure où, même pour un TP des ENS, il est peu probable qu'un sujet demande de prendre du temps pour faire de jolis dessins plutôt que des algorithmes complexes.

Le module `graphics` n'est pas dans la librairie standard, il faut donc le charger, soit pour les unixiens en écrivant dans un terminal `ocaml graphics.cma` au lieu de `ocaml` quand on ouvre une session, soit pour ceux qui utilisent l'éditeur officiel en profitant du fait que cet éditeur fait automatiquement une manipulation équivalente.

La première instruction avant toute utilisation de fonctions graphiques doit dans les deux cas être `Open Graphics` pour éviter de préfixer tous les noms de fonctions par `Graphics.`. Les fonctions ci-après sont issues du module et le préfixage a été retiré pour la lisibilité.

Pour commencer, il faut ouvrir une fenêtre graphique, ce qui se fait par la fonction `open_graph`, qui prend en argument une chaîne de caractères. Cet argument peut être `""` afin d'ouvrir une fenêtre par défaut, ou `" 800x600"` pour ouvrir une fenêtre de taille 800 par 600. Attention à bien mettre une espace avant, qui est en fait un délimiteur, car on peut choisir l'écran où ouvrir la fenêtre graphique en le précisant avant l'espace (je ne l'ai jamais fait...).

La fenêtre ne peut être ouverte qu'une fois, donc tenter d'en ouvrir une autre écrasera la précédente. Pour la réinitialiser, on utilise `clear_graph ()`, pour la redimensionner `resize_window largeur hauteur`, pour changer son titre `set_window_title titre` et pour la fermer `close_graph ()`.

Il est bon de savoir que fermer à l'aide de la croix une fenêtre graphique déclenche une erreur, pouvant aller jusqu'à la fermeture de la session Caml quand on travaille dans un terminal.

Le type couleur est un entier, qui est égal à 256^2 `R + 256 G + B` selon le code RGB. Il existe d'ailleurs une fonction `rgb` qui à trois entiers associe la couleur correspondante. Les couleurs principales sont prédéfinies en OCaml. La fonction `set_color` permet de changer la couleur du pinceau.

Ensuite, les fonctions de dessin prennent le relais [15] :

- `plot x y` imprime le point `(x,y)` ;
- `moveto x y` déplace le pointeur à la position `(x,y)`, `rmoveto x y` déplace le pointeur selon le vecteur `(x,y)` ;
- `lineto x y` trace une ligne depuis le pointeur, qu'il déplace, jusqu'à `(x,y)`, `rlineto x y` trace une ligne depuis le pointeur, qu'il déplace, selon `(x,y)` ;
- `draw_rect x_coin_bg y_coin_bg lrg htr` dessine un rectangle sans modifier la position du pointeur ;
- `draw_poly_line` dessine les lignes joignant deux points consécutifs du tableau en argument, `draw_poly` fait de même et ferme le polynôme formé ;
- `draw_segments` dessine les segments définis par le tableau de quadruplets d'entiers en argument ;
- `draw_arc x_centre y_centre rayon_x rayon_y angle_dp angle_ar` dessine un arc, `draw_ellipse x_centre y_centre rayon_`x `rayon_y` (cas particulier) dessine une ellipse, `draw_circle x_centre y_centre rayon` (cas particulier) dessine un cercle ;
- Pour les suffixes `rect`, `poly`, `ellipse`, `circle` et `arc`, remplacer `draw` par `fill` remplit la zone au lieu de dessiner une figure ; dans le dernier cas, il s'agit de la zone définie par arc et corde.

Il est également possible d'imprimer du texte dans une fenêtre graphique, ce qui se fait par les fonctions `draw_char` et `draw_string`. Le caractère ou la chaîne a son coin en bas à gauche au niveau du pointeur, qui est déplacé au coin en bas à droite [16]. La taille des caractères, qui dépend aussi de l'implémentation, se modifie par `set_text_size`, la fonte (idem) par `set_font`. Afin de savoir la taille du rectangle dans lequel, avec les paramètres actuels, le texte `str` serait affiché, on dispose de `text_size str`.

Certaines informations peuvent aussi se récupérer : `point_color x y` retourne la couleur du point en `(x,y)`, en déclenchant une erreur `BadMatch` si ce point est hors de la fenêtre graphique, `current_x ()`, `current_y ()` et `current_point ()`, retourne les coordonnées de la position courante sous forme d'un couple d'entiers, `mouse_pos ()` retourne les coordonnées de la souris, `button_down ()` retourne un

15. pour toutes ces fonctions, en cas de débordement, aucune erreur n'est déclenchée
16. donc pensez à ajouter des espaces si vous utilisez plusieurs fois les fonctions

booléen indiquant si un bouton est enfoncé, et `read_key ()` attend qu'une touche soit enfoncée **alors que la fenêtre graphique a le focus** et retourne le caractère correspondant. Les flèches sont malheureusement ignorées (gros défaut du module).

Les événements ajoutent de l'interactivité, plus ou moins en temps réel. Il y en a cinq : `Button_down`, `Button_up`, `Key_pressed`, `Mouse_motion` et `Poll`, ce dernier se produisant à tout moment. Le statut est un type produit contenant les informations suivantes :
`type status = { mouse_x : int; mouse_y : int; button : bool;`
`keypressed : bool; key : char; }`.

Ceci est utilisé par la fonction `wait_next_event`, qui attend qu'au moins un événement de la liste donnée en argument se produise, et retourne le statut à ce moment-là. Si la souris est hors de la fenêtre graphique, les valeurs `mouse_x` et `mouse_y` peuvent ne pas être exploitables par d'autres fonctions sans déclencher d'erreur.

Tout ceci n'est pas sans rappeler l'interactivité de pages avec JavaScript, ou bien ?

Exercice 1 : « S'il vous plaît, dessine-moi un mouton ! »

Exercice 2 : Écrire une fonction qui dessine un carré dans une fenêtre graphique et sur lequel certaines touches du clavier auront des effets divers, notamment le déplacer, le rétrécir, l'agrandir, le tourner, le remplir, le vider.

Exercice 3 : Écrire une fonction qui affiche dans une fenêtre graphique le texte que l'utilisateur saisit. Trouvez notamment comment effacer un caractère.

Pour ce dernier exercice, une solution peu élégante (mais utile pour d'autres cas) pour effacer une zone de texte est de redessiner en blanc dessus.

TP 7 : Débug

Attention : Ce TP ne fera pas l'objet d'une séance. Il peut cependant être fait en guise d'entraînement.

Ce TP est à lire en complément du TP 4 d'IPT SUP, en utilisant les mêmes principes transposés à Caml.

En plus des classiques erreurs de syntaxe, Caml fait la part belle aux erreurs de type. Sans entrer dans les détails du typage, Caml considèrera toujours que dans les cas de filtrage ou les disjonctions de cas, puisqu'une expression ne peut avoir qu'un type (sauf à utiliser des exceptions), les contraintes de type se renforcent de haut en bas à la lecture du programme [17], et en cas d'incohérence une erreur est déclenchée, sous la forme de « cette expression est de type <blabla>, mais doit avoir le type <blibli> ».

Ainsi, en écrivant :

```
let fail x = match x with
|0 -> failwith "nul"
|1 -> 0
|_ -> true
```

Caml ne déduira rien de l'exception (écrire `let f () = failwith "erreur";;` donnera d'ailleurs une expression de type `unit -> 'a`), puis considèrera que le type est `int`, puis rencontrera une incompatibilité en tombant sur un objet de type `bool`.

L'éditeur de Caml, le terminal et le compilateur permettent de baliser les erreurs de syntaxe, avec cependant de temps en temps des problèmes de localisation (en même temps, quand on voit la gestion des retours à la ligne et l'incompatibilité avec ne serait-ce que le format TXT. . .).

La technique donnée dans le TP correspondant en Python et consistant à imprimer des informations en cours d'exécution se transpose facilement à Caml, en utilisant éventuellement l'impression formatée (voir la section 2.7 du chapitre 1).

17. Une formulation plus explicite mais moins précise voire erronée est de dire que c'est le premier type qui compte, mais si le premier type est `'a list` car on a une liste vide et plus loin on voit une liste d'entiers, c'est bien une précision qui est apportée, sans déclencher de contradiction.

En revanche, les éditeurs connus ne disposent pas de débogueur. Il reste la possibilité (laborieuse et parfois trop obscure pour être utile) de « tracer » les fonctions, en voyant les appels récursifs. Les directives `#trace` et `#untrace` permettent d'activer ou de désactiver cette option, en précisant dans les deux cas le nom de la fonction à tracer et sans oublier le croisillon supplémentaire.

Exercice : Tracer un maximum de fonctions récursives écrites dans les autres TP et constater le résultat. Faire ceci en particulier pour toutes les versions de la factorielle.

TP 8 : Récursivité [18]

ATTENTION : Ce TP ne fera pas l'objet d'une séance en raison de la redondance. Il peut cependant être fait en guise d'entraînement.

Exercice 1 : Écrire une fonction imprimant les instructions pour résoudre le problème des tours de Hanoï.

Remarque : Ce problème consiste à déplacer une pile de n (en argument) anneaux de taille croissante d'un tas (matérialisé par un piquet) à un autre (parmi trois), les opérations élémentaires étant le déplacement d'un anneau du haut d'une pile sur le haut d'une autre pile, à condition qu'il soit plus petit que l'ancien sommet de la pile d'arrivée.

Le nombre optimal d'opérations élémentaires est $2^n - 1$.

Exercice 2 : Écrire une fonction comptant le nombre de façons de tracer une ligne de longueur `n` (en argument) avec des segments de longueur deux ou trois (l'ordre est important).

Exercice 3 : Adapter le code de l'exercice précédent pour retourner la liste des façons de tracer la ligne en question.

Exercice 4 : Déterminer ce que retourne la fonction de McCarthy.

```
let rec mccarthy n = if n > 100 then n-10 else mccarthy (mccarthy (n+11));;
```

Exercice 5 : Recopier le code de la fonction d'Ackermann et l'appeler pour certaines valeurs des arguments.

```
let rec ackermann m n =
    if m = 0 then n+1
    else if n = 0 then ackermann (m-1) 1
    else ackermann (m-1) (ackermann m (n-1));;
```

18. Ce TP correspond au troisième TP d'IPT de deuxième année, sauf les exercices relevant d'autres sections du chapitre 2 d'option de première année. Là aussi, toutes les fonctions demandées devront être récursives.

Exercice 6 : Écrire une fonction qui calcule le PGCD de deux entiers à l'aide de l'algorithme d'Euclide.

Exercice 7 : Écrire une fonction qui détermine si un entier relatif ne comporte que des 0 et des 1 dans son écriture en base 3.

Terminons ce TP par quelques compléments. La fonction de Morris est un cas particulier de fonction récursive, dont la terminaison se prouve à l'aide d'un variant... et qui pourtant ne termine pas.

En pratique, c'est dû au fait que Python, comme la plupart des langages, évalue d'abord les arguments d'une fonction avant de procéder à son appel.

Ainsi, dans le code suivant :

```
let rec morris m n =
    if m = 0 then 1
    else morris (m-1) (morris m n);;
```

... appeler `morris(1,0)` provoquera un dépassement de la pile d'exécution, car Python calculera `morris(0,morris(0,... morris(0,morris(1,0))...))`, qui vaut bien entendu `1`, mais cette valeur ne sera jamais obtenue.

Ceci incite à la prudence lors de l'écriture de preuves de terminaison.

Le dernier exercice met surtout en œuvre la structure de pile, du point de vue algorithmique, mais rien n'empêche de chercher une solution en tant que fonction récursive.

Exercice 8 : Écrire des programmes qui résolvent des problèmes classiques de « passage de rivière », notamment celui du loup, de la chèvre et du chou.

TP 9 : Diviser pour régner

Dans ce TP, des applications du principe « diviser pour régner » sont présentées, associées à des exercices de programmation en Caml. Les formules de récurrence pour le calcul de complexité ainsi que les complexités sont à donner avec chaque programme.

Travail à faire pendant la séance

Exercice 1 : Écrire une fonction de recherche d'un élément dans un tableau trié.

Exercice 2 : La multiplication du paysan revient à calculer un produit en n'utilisant que des multiplications et divisions (euclidiennes) par 2 et des additions. Le principe est d'écrire un des facteurs, noté x, en binaire et d'additionner les $2^i y$, où y est l'autre facteur, tels que le bit correspondant à 2^i soit à 1 dans x. Concrètement, on regarde x modulo 2 et s'il vaut 1 on ajoute y à un accumulateur, puis on multiplie y par 2 et on divise x par 2 jusqu'à ce que x soit nul. Écrire ceci sous la forme d'une fonction.

Exercice 3 : Reprendre ce principe pour calculer a^b (exponentiation rapide).

Exercice 4 : Écrire les algorithmes de tri DPR sur des listes (le tri fusion y est plus adapté) et des tableaux (le tri rapide y est plus adapté).

Exercice 5 : (Méthode de Karatsuba pour la multiplication de deux polynômes [19]) : Pour calculer $P \times Q$, où P et Q sont deux polynômes de degré $2n$ (au plus), on écrit $P = P_1 + X^n P_2$ et $Q = Q_1 + X^n Q_2$, où les quatre nouveaux polynômes sont de degré au plus n. On sait que $PQ = P_1Q_1 + X^n(P_1Q_2 + P_2Q_1) + X^{2n}P_2Q_2$, cependant on peut se limiter à trois multiplications en calculant $(P_1 + P_2)(Q_1 + Q_2)$. Prouver qu'on obtient bien le produit souhaité en trouvant les deux autres produits à faire, et donner la complexité de l'algorithme (pas besoin de l'écrire pour une fois). [20] Consulter le web pour y découvrir l'algorithme de Strassen permettant la multiplication de deux matrices en un temps meilleur que cubique, précisément en $\mathcal{O}(n^{\log_2(7)})$.

---

19. L'algorithme naïf fonctionne en temps quadratique.
20. Pour aller plus loin, un principe encore plus efficace et utilisant aussi le DPR est de passer par la transformée de Fourier rapide, la complexité en temps sera alors un $\mathcal{O}(n \log(n))$.

Exercices pour réfléchir après la séance

Exercice 6 (*) : Écrire un programme DPR pour déterminer l'enveloppe convexe d'un nuage de points [21].

Le principe est de calculer l'enveloppe convexe des deux moitiés du nuage contenant les points les plus à gauche et les points les plus à droite (par exemple), puis de rassembler ces enveloppes en ajoutant exactement deux lignes (et en en supprimant un certain nombre).

Pour aller plus loin, ce problème a fait l'objet du concours blanc d'option première année en 2018 (et du sujet d'informatique B au concours X-ENS de 2015, avec deux algorithmes DPR différents dans les deux sujets).

Exercice 7 (*) : Écrire un programme DPR qui calcule le nombre d'inversions dans un tableau ou dans une liste (au choix).

Une inversion dans `t` est un couple `(i,j)` tel que `i < j` et `t.(i) > t.(j)`.

Le principe est de faire un tri fusion et de compter les inversions au moment de fusionner : les inversions d'un tableau sont les inversions de sa moitié gauche, les inversions de sa moitié droite et pour chaque élément de la moitié droite le nombre d'éléments supérieurs dans la moitié gauche, nombre qu'on peut déterminer en temps constant par le principe annoncé.

Pour aller plus loin, ce problème a fait l'objet du concours blanc d'option première année en 2019.

Exercice 8 (**) : Écrire un programme DPR qui détermine les deux points les plus proches dans un nuage de points du plan.

Il s'agit de disposer d'informations suffisantes sur le nuage de points, on créera donc deux tableaux : l'un trié par abscisses croissantes (et par ordonnées croissantes à même abscisse) et l'autre trié par ordonnées croissantes.

On peut séparer le tableau trié en abscisses en deux moitiés de taille égale sur lesquelles on applique récursivement ce principe jusqu'à ce que la taille du tableau soit

21. L'algorithme du paquet cadeau est plus intuitif, mais quadratique.

assez petite pour qu'un algorithme naïf suffise (moins de 6 points par exemple).

Si on dispose d'un couple de points minimisant la distance pour chaque moitié (disons que `d` est la plus petite des deux distances obtenues), il faut vérifier qu'aucun couple dont les éléments sont chacun dans une moitié ne donne une distance inférieure à `d`, mais si c'était le cas ils seraient dans une bande d'abscisse de largeur inférieure ou égale à `2*d`, et le nombre de points du nuage dans cette bande est assez faible (car ils sont espacés d'au moins `d`).

En extrayant les points dont l'abscisse est dans la bande du tableau trié par ordonnées croissantes, on peut déterminer en temps linéaire s'il y a une distance inférieure à `d` dans le tableau obtenu, car l'étroitesse de la bande permet de limiter la recherche à une zone constante du tableau (faire un dessin pour le prouver).

TP 10 : Programmation dynamique

Dans ce TP, des applications de la programmation dynamique sont présentées, ainsi que des exercices impliquant des algorithmes gloutons.

Travail à faire pendant la séance

Exercice 1 : Écrire un programme dynamique déterminant le nombre minimal de multiplications de scalaires à faire pour multiplier n matrices de dimensions variées.

Par exemple, pour calculer $M_1M_2M_3M_4$, où les dimensions respectives des matrices sont $(4,6)$, $(6,2)$, $(2,10)$ et $(10,3)$, les multiplications peuvent être faites ainsi :
- $M_1(M_2(M_3M_4))$ ($2\times 10\times 3+6\times 2\times 3+4\times 6\times 3$ soit 168 multiplications) ;
- $(M_1M_2)(M_3M_4)$ ($4\times 6\times 2+2\times 10\times 3+4\times 2\times 3$ soit 132 multiplications) ;
- $M_1((M_2M_3)M_4)$ ($6\times 2\times 10+6\times 10\times 3+4\times 6\times 3$ soit 372 multiplications) ;
- $((M_1M_2)M_3))M_4$ ($4\times 6\times 2+4\times 2\times 10+4\times 10\times 3$ soit 248 multiplications) ;
- $(M_1(M_2M_3))M_4$ ($6\times 2\times 10+4\times 6\times 10+4\times 10\times 3$ soit 480 multiplications).

Pour information, le nombre de façons d'organiser les multiplications de $n+1$ matrices est le n-ième nombre de Catalan, donné par les formules équivalentes

$$C_n = \frac{\binom{2n}{n}}{n+1} = \frac{(2n)!}{n!(n+1)!} = \binom{2n}{n} - \binom{2n}{n-1}.$$

Le nombre de Catalan se retrouve très souvent en combinatoire avancée.

Exercice 2 : Écrire un programme glouton pour le problème du rendu de monnaie avec des billets en euros pour une somme multiple de 5.

Exercice 3 : Écrire un programme dynamique pour le problème du rendu de monnaie dans le cas général.

Exercices pour réfléchir après la séance

Exercice 4 : Ordonnancement de tâches pondérées (*weighted interval scheduling*) : écrire un algorithme dynamique déterminant la valeur maximale d'un ensemble de tâches effectuables, les tâches étant définies par leur heure de début, leur heure de fin et leur valeur, et une seule tâche pouvant être accomplie à la fois.

Conseil : oublier tout de suite l'algorithme glouton.

Exercice 5 : Distance d'édition [22] : Soient deux chaînes de caractères s_1 et s_2. La distance d'édition de s_1 à s_2 correspond au coût minimal pour passer de s_1 à s_2 (coût qui n'est pas symétrique a priori) en nombre d'insertions de caractères, de modifications de caractères et de suppressions de caractères (n'importe où dans les trois cas). On associe à chacune de ces opérations un coût constant, mais pas nécessairement égal entre les opérations (on considère dans le cas simple que les coûts sont tous d'un). Écrire un programme dynamique qui calcule la distance d'édition entre deux chaînes de caractères dans le cas simple puis dans le cas général.

Exercice 6 : Écrire un programme dynamique pour le problème du sac à dos. Écrire ensuite un programme glouton et constater qu'il n'est pas optimal dans certains cas.

Le problème du sac à dos est le suivant : étant donné un ensemble d'objets d'un certain poids et d'une certaine valeur, comment remplir un sac à dos avec un ou plusieurs exemplaire(s) (certaines variantes excluent cependant d'en prendre plusieurs) de certains objets de sorte que la valeur transportée soit maximale et le poids soit inférieur à un seuil donné ? Le problème demande normalement de renvoyer la valeur maximale sans détailler les objets à transporter, mais le programme devra retourner les deux (ou au moins la liste des objets, dont poids et valeur seront déduits).

22. À partir de maintenant, quand vous effacerez beaucoup de texte parce que vous avez vu \b\b\b\b\b\b\baurez vu une erreur en début de ligne, vous penserez à moi !

TP 11 : Piles et applications [23]

ATTENTION : Ce TP ne fera pas l'objet d'une séance en raison de la redondance. Il peut cependant être fait en guise d'entraînement.

1 Implémentation d'une pile

Exercice 1 : Écrire une fonction qui accède au i-ième élément d'une pile (naturellement et à l'aide des opérations de base sur les piles).

Exercice 2 : Écrire une fonction qui échange les deux éléments au sommet d'une pile (idem).

Exercice 3 : Écrire une fonction qui met l'élément du sommet au fond de la pile (idem).

Exercice 4 : Écrire une fonction qui inverse l'ordre des éléments de la pile (idem).

Il peut être intéressant de chercher à provoquer une erreur de dépassement de pile [24], aussi appelée *stack overflow*.

Une remarque pratique : dans un éditeur tel que Libreoffice, la pile des modifications est de capacité relativement courte [25]. Ici, il s'agit d'une pile dont le fond est progressivement effacé plutôt que de provoquer une erreur quand on tente d'empiler trop d'éléments.

Exercice 5 : Réaliser une implémentation d'une pile adéquate à l'aide d'un tableau circulaire et réécrire les fonctions de ce TP.

23. Ce TP correspond aux deux premiers TP d'IPT de deuxième année.
24. quand on programme, elles arrivent vite si on fait un oubli bête...
25. On voit vite sa limite quand on résout un Su-Doku sur un tableur et qu'on émet des hypothèses...

2 Applications des piles

2.1 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. Dans la mesure où tout ce qui n'est pas une parenthèse n'a pas de pertinence dans cette étude, on considèrera que les expressions sont des mots n'utilisant que les caractères `'('` et `')'`.

Exercice 5 : Donner une condition nécessaire et suffisante pour qu'un mot soit bien parenthésé.

Exercice 6 : Prouver que le nombre de mots bien parenthésés de taille $2n$ est $\frac{1}{n+1}\binom{2n}{n}$ [26].

Exercice 7 : Écrire un programme qui vérifie si un mot est bien parenthésé. [27]

Exercice 8 : Écrire un programme 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.

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 une liste d'entiers non nuls [28]

Exercice 9 : Écrire un programme qui vérifie si une telle liste est bien parenthésée.

2.2 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.

---

26. le n-ième nombre de Catalan
27. L'utilisation des piles n'est pas obligatoire ici.
28. On pourrait assimiler le zéro à autre chose qu'une parenthèse.

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 : un pour le signe (s'appliquant à un opérande) et un pour la soustraction (s'appliquant à deux opérandes) [29].

Exercice 10 : Écrire un programme qui évalue une expression arithmétique qui est donnée en notation polonaise inversée, par exemple sous la forme d'une liste de chaînes de caractères [30]. L'expression utilisera seulement des entiers, des flottants et les opérateurs usuels sur ces nombres.

2.3 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 [31]. Cette méthode consiste simplement à faire un parcours en profondeur.

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 11 : Écrire un programme 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.

29. De nombreuses calculatrices ont utilisé cette notation, ce qui explique l'emploi historique des deux boutons.
30. ou, pour les plus sportifs, d'une chaîne qu'il faudra éclater suivant les espaces
31. Attention aux éraflures !

Exercice 12 : Écrire un programme correspondant.

Bonus pour les plus motivés : 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, à partir de tirages aléatoires et de piles. N'hésitez pas à le tenter.

TP 12 : Tris [32]

Attention : Ce TP ne fera pas l'objet d'une séance en raison de la redondance. Il peut cependant être fait en guise d'entraînement.

Exercice 1 : Écrire une version non en place des tris par insertion et par sélection. La fonction devra alors avoir une valeur de retour, puisque la liste en argument ne sera pas modifiée.

Le tri à bulles pèche par sa complexité, et seule sa compréhension relativement facile et son originalité font qu'il est enseigné [33].

Ainsi donc, rentabilisons-le en faisant quelques calculs dessus. Vérifier si la liste est déjà triée peut se faire de manière très pratique dans le tri à bulles : on utilise un booléen déterminant si un échange a été effectué dans un parcours de la boucle principale. Si le booléen reste à `False`, c'est que la liste est déjà triée.

L'utilité est de ne pas perdre de temps quand la liste de départ est presque triée, dans la mesure où par exemple quelques éléments parmi les plus grands sont au début de la liste.

Le principe du tri à bulles fait que de tels éléments seront vite envoyés à leur place [34], mais le souci est qu'au contraire des éléments parmi les plus petits sont peut-être à la fin de la liste [35], et ils devront être échangés au cours d'un nombre considérable [36] de parcours de la boucle principale.

Le tri cocktail pallie ce souci en faisant des parcours de la liste dans les deux sens, afin qu'il n'y ait pas à proprement parler de « tortue ». Après chaque (double) parcours dans le tri cocktail, un élément supplémentaire est à la bonne place dans les deux extrémités.

Exercice 2 : Implémenter le tri cocktail.

---

32. Ce TP correspond au quatrième TP d'IPT de deuxième année.
Travail préliminaire : Lire le chapitre sur les tris issus du cours d'IPT deuxième année. Adapter en Caml tous les algorithmes de tris qui y figurent.
33. Si mon cours était sur Wikipédia, ce passage subjectif serait irrémédiablement retiré…
34. On utilise le terme imagé de « lièvres ».
35. Et ceux-là sont appelés des « tortues ».
36. … de lièvre ?

Exercice 3 : Faire les preuves de terminaison et correction du tri fusion et du tri rapide.

Exercice 4 : Écrire une fonction `split2` utilisant le principe de la médiane des médianes. Prouver que la recherche de la médiane est alors en temps linéaire.

Exercice 5 : Faire tourner « à la main » un tri au choix du cours sur une entrée de taille 16.

Exercice 6 : Implémenter une fonction de tri qui applique un algorithme le plus efficace possible dans l'hypothèse où la liste en entrée est obtenue à partir d'une liste triée après un nombre très faible d'insertions de nouveaux éléments

Exercice 7 : Implémenter le tri par dénombrement.

Exercice 8 : Implémenter le tri par base pour des chaînes de caractères.

Exercice 9 : Implémenter un algorithme de tri le plus efficace possible pour une liste correspondant à une permutation quelconque dans $\mathcal{S}_n$.

TP 13 : Arbres binaires

Dans ce TP, nous allons faire des calculs sur les arbres binaires.

Pour simplifier, les étiquettes seront simplement des entiers. Le type employé ici est alors `type arbin = Vide | Noeud of arbin * int * arbin`.

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 liste des étiquettes (quel que soit l'ordre) 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), par programmation dynamique [37].

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.

37. Un exercice de combinatoire de niveau master revient à donner la formule.