Notes de cours d'Informatique MP2I

Julien Reichert

2023/2024

Ces notes de cours reprennent le programme officiel de l'enseignement d'informatique en MP2I, et correspondent à une version alternative par rapport au cours réalisé en direct, potentiellement d'une manière différente afin de donner deux occasions d'assimiler les notions abordées.

Le découpage est chronologique, avec une précision de la partie correspondante dans le programme officiel, sachant que les parties ne sont pas forcément abordées d'une traite, et certaines sont couvertes sur les deux années.

La première partie consiste en l'apprentissage des langages au programme, ainsi qu'une révision de Python à toutes fins utiles. Cette partie sera abordée au fur et à mesure des besoins ou occasions au cours de l'année.

La page d'introduction de chaque chapitre s'inspire du style de Mme Le Blanc, enseignant les mathématiques en MP2I au lycée Kléber, et que je remercie.

Mis à part ces passages, du texte orange dans ces notes de cours marque les informations mises pour la culture, mais absentes du programme et non évaluées.

Table des matières

Première partie

Langages de programmation du programme

Chapitre A)

Le langage Python

Ce chapitre est à prendre comme une fiche mémo, en particulier peu de phrases explicatives y figurent.

L'utilité de Python est restreinte en MP2I, notamment pour le parcours majoritaire. Ce langage reste néanmoins un bon outil pour l'usage quotidien de l'informatique, permettant rapidement de faire faire des calculs qu'on ne souhaite pas faire à la main.

Les données et leurs types

Types de base

Types simples de base : entiers, flottants, *booléens*. Séquences de base : *n-uplets*, *listes*, *chaînes de caractères* (pas de type « caractère simple » en Python).

Typage *dynamique* (le type de chaque expression est déterminé lors de l'exécution d'un programme, et non pas lors de l'analyse du code).

Entiers

Opérations : `+`, `-` (soustraction et opposé), `*` (**ne peut pas être implicité**), `//` (**quotient de la division euclidienne**), `%` (le reste de la division euclidienne) `**` (puissance).

Attention : ne pas utiliser pour la puissance l'opérateur `^`.

Flottants

Séparateur décimal : un point (convention anglo-saxonne).

Si à gauche ou à droite de la virgule, la valeur est zéro, on peut ne pas l'écrire (mais zéro ne peut pas s'écrire simplement `.`, ce sera `0.` ou `.0`).

Opérations : `+`, `-`, `*` et `**` comme pour les entiers, `/` pour la division (retourne un flottant même si elle concerne deux entiers), `//` et `%` fonctionnent, avec un comportement intuitif, mais sans intérêt a priori.

Conversions : `int()` (troncature), `float()` (conversion d'un entier en lui-même virgule zéro), `math.floor()` (arrondi par défaut, donc la partie entière) et `math.ceil()` (arrondi par excès), `round()` (arrondi flottant vers entier si un argument, flottant vers flottant si deuxième argument pour le nombre de chiffres après la virgule).

Priorités opératoires comme en mathématiques. Parenthèses avec imbrications, ne pas utiliser de crochets comme en mathématiques.

Booléens

Deux éléments : `True` et `False`. **Attention aux capitales**.

Opérations : `and`, `or` et `not`.

La négation est prioritaire sur la conjonction qui est prioritaire sur la disjonction : `True or not True and False` est en fait `True or ((not True) and False)`.

Évaluation paresseuse : quand l'expression `a and b` est évaluée (avec `a` et `b` deux expressions), si `a` se révèle fausse, `b` n'est même pas évaluée.

Produire des booléens : comparaisons `<`, `>`, `<=`, `>=`, `==` (test d'égalité, à ne pas utiliser avec un seul égal) et `!=` (test de différence). Enchaîner les comparaisons (ne pas abuser) : le résultat étant vrai si toutes les comparaisons isolées sont vraies. Les comparaisons sont prioritaires sur les opérations booléennes.

Quelques résultats à connaître au cas où :

- `42 == 42.0` est vrai, de même pour tous les entiers.
- D'ailleurs, `True == 1` est vrai, de même que `False == 0`, `True == 1.0` et `False == 0.0`.
- Du coup, `True + True` vaut 2 et `False < True` est vrai.
- On peut (mais on ne doit pas) utiliser par exemple une chaîne de caractères dans les tests. Bien que `False == ""` soit faux, la chaîne vide est la seule qui s'assimile à `False` si on l'utilise dans un test. Quoi qu'il en soit, il faut éviter de confier à Python le soin de comprendre quelque chose qui n'est pas un booléen comme un booléen.

Opérateurs `&`, `|` et `^` équivalents bit-à-bit pour des entiers de `and`, `or` et du test de différence ("xor" en anglais, « ou exclusif » en français), respectivement. Ils ne sont pas à connaître et peuvent aussi être utilisés pour les booléens. Pas paresseux, et prioritaires sur les comparaisons.

Variables

Nom composé de lettres (qu'on évitera d'accentuer), de chiffres, à condition que ce ne soit pas par un chiffre que le nom commence, et de caractères de soulignement.

Affectation ou réaffectation (le type peut changer) d'une variable : `nom = valeur`.

Réaffectation à partir d'une ancienne valeur : `+=`, `-=`, etc. (à peu près tous les symboles marchent). Pas d'incrémentation ou de décrémentation avec `++` ou `-`.

Séquences

Introduction

Séquences : objets qui possèdent une *taille* (`sized`), sont *itérables* (*iterable*, on peut les parcourir), *indexables* (*container*, on peut accéder à chacun de leurs éléments au moyen d'un indice), et *tranchables* (*sliceable*, tentative personnelle de traduction).

Voir `http://sametmax2.com/les-trucmuchables-en-python/` pour plus d'informations sur ce vocabulaire.

Opérations communes aux séquences

Fonction `len()` pour la taille.

Accès au `i`-ième élément `t` par `t[i]`, indices commençant à `0`, erreur si `i >= len(t)`.

Accès à partir de la fin : `t[-1]` dernier élément, ainsi de suite jusqu'à `t[-len(t)]` premier élément (indice strictement inférieur : erreur).

Accès à une tranche (*slice*) : `s[i:j]`, donne une séquence du même type formée de `s[i]`, `s[i+1]`, ..., `s[j-1]`, en remplaçant tout nombre négatif par lui-même plus la taille de `s`, à la manière décrite à la phrase précédente. Vide si `t[i]` est à droite de `t[j-1]`, tronquée aux extrémités de `t` sans provoquer d'erreur si i ou j débordent des indices autorisés. Il est même possible de préciser le pas dans la tranche, permettant notamment de lire à l'envers. La syntaxe devient `t[i:j:pas]`, vide suivant l'ordre entre `i` et `j` et le signe de `pas`, avec `pas` forcément non nul.

Test d'appartenance : `x in seq`.

Fusion : `seq1 + seq2`.

Répétition : `seq * nombre` ou `nombre * seq`, vide si `nombre <= 0`.

n-uplets

En anglais : *tuple*, collection d'éléments séparés par des virgules, éventuellement entourée de parenthèses.

Le 0-uplet : `()`. Un 1-uplet : `(x, )`.

On ne peut accéder aux éléments d'un n-uplet qu'en lecture, il est juste possible de réaffecter un n-uplet dans son ensemble si on veut le modifier.

Peuvent être déconstruits : on écrira `(a, b, c) = (3, 2, 1)`, parenthèses optionnelles des deux côtés s'il n'y a pas de risque de confusion, affecte d'un coup trois variables (attention à la lisibilité).

Attention aux parenthèses pour la fusion : `1, 2 + 3, 4` donne `(1, 5, 4)` au contraire de `(1, 2) + (3, 4)`, et `3 * 1, 2` donne `(3, 2)`, au contraire de `3 * (1, 2)`.

Chaînes de caractères

Caractères rassemblés entre deux délimiteurs (apostrophes ou guillemets, simples ou triples), l'utilisation de délimiteurs triples permettant d'aller à la ligne en écrivant la chaîne.

Impossible de modifier un caractère isolé d'une chaîne.

Caractères échappés avec antislash (délimiteur, saut de ligne, tabulation, antislash lui-même).

Test d'appartenance avec `in` utilisables pour des sous-chaînes (cas particulier).

Comparer : `"23" in "123456"`, `"24" in "123456"`, `(2, 3) in (1, 2, 3, 4, 5, 6)` et `(2, 3) in (1, (2, 3), 4, (5, 6))`. Au passage, voir l'effet en retirant les parenthèses à gauche de `in`.

Listes

Remarque : Il faut éviter de dire « tableau », car en Python, contrairement à d'autres langages, on appelle « tableau » une structure, introduite dans la bibliothèque `numpy`, ressemblant à une liste mais contenant préférablement uniquement des entiers ou uniquement des flottants.

Liste : collection d'éléments, comme un n-uplet, mais les éléments peuvent être modifiés individuellement.

Création : donner les éléments séparés par des virgules, entourer le tout de crochets.

Impossible d'ajouter un ou plusieurs éléments en donnant un indice au-delà de la taille.

Pour ajouter un élément : `l.append(x)`.

Pour ajouter autant d'éléments qu'on veut : `l1 += l2` ou `l1.extend(l2)`

Pour retirer le dernier élément : `l.pop()` (retourne l'élément retiré). Possibilité d'ajouter un argument (indice du retrait), coût en fonction de la taille de la liste et de l'indice.

Création d'une liste *en compréhension*, (cf. description d'ensembles en mathématiques). La liste `[expr for x in seq]` est celle des valeurs de l'expression `expr`, dépendant a priori d'une variable nommée `x`, pour toutes les valeurs de la séquence `seq`, stockées dans `x`, dans l'ordre de cette séquence. Possibilité d'imbriquer des boucles ou de glisser des tests.

Conversions

Fonctions intuitives : `list(a)`, `tuple(a)`.

Fonction pas intuitive : `str(a)`, donne la représentation telle qu'on verrait à l'impression.

Convertir une liste en chaîne : méthode `join` par exemple.

Séparer une chaîne en liste : méthode `split`.

Instructions composées

Séquence d'instructions

Enchaîner deux instructions : aller à la ligne entre les deux. Ne pas utiliser de point-virgule, c'est moche.

Disjonction de cas (if)

À NE SURTOUT PAS QUALIFIER DE BOUCLE !

Introduction par `if`, puis la condition (interprétée comme un booléen), puis **double-point en fin de ligne**.

Contenu délimité par **l'indentation** du bloc (toutes les lignes dans le corps du test doivent commencer par le même nombre, d'espaces par rapport à la ligne d'introduction).

Partie optionnelle pour « sinon » : s'introduit par `else`, suivi d'un **double-point**, aussi avec indentation spécifique pour le corps.

Partie optionnelle pour « sinon si » (avant) : s'introduit par `elif`, suivi d'une autre condition et d'un **double-point**, toujours avec une indentation spécifique.

Un bloc avec `elif` est exécuté si aucune des conditions précédentes n'est remplie alors que la condition actuelle l'est ; un bloc avec `else`, nécessairement en toute fin s'il existe, est exécuté si aucune des conditions n'est remplie.

Boucle inconditionnelle (for)

En Python : c'est toujours un parcours d'itérable. Une variable, créée pour l'occasion et qu'on n'a pas besoin d'initialiser par ailleurs, vaudra successivement chacun des éléments dans le corps de boucle.

Syntaxe : `for i in a:`, où `i`. Corps de boucle indenté ensuite.

Séquence classique dans de telles boucles : un « objet-range » (qui n'est pas une liste). Fonction `range(debut, fin, pas)`, avec `debut` optionnel (défaut zéro) et `pas` optionnel (défaut un).

Parcourir une liste `l` : au choix `for x in l:` ou `for i in range(len(l)):` en commençant le bloc par `x = l[i]`. Utiliser en fonction du besoin de connaître les indices ou non. Éventuellement utiliser la fonction `enumerate`.

On ne peut pas perturber une boucle inconditionnelle par modification de la variable, et il faut éviter de la perturber par mutation de la séquence.

Boucle conditionnelle (while)

Introduction : `while cond:`, où `cond` est la condition, on n'oublie toujours pas le **double-point** ni l'indentation dans le corps de la boucle, qui est exécuté tant que la condition est remplie. Condition testée entre la fin de chaque passage dans la boucle et le début de l'éventuel passage suivant (mais pas pendant).

Possibilité d'imbriquer boucles et tests l'un(e) dans l'autre à un niveau de profondeur arbitraire.

Fonctions

Définition par `def f(arg1, arg2, ...):`, les arguments doivent être des noms (pas de déconstruction). Corps indenté.

Valeur de retour obtenu par `return` (interrompt éventuellement prématurément la fonction).

Appeler une fonction : `f(a, b, ...)`, les expressions sont évaluées en priorité (doivent pouvoir l'être).

Entrées et sorties

Prompteur pour récupérer une saisie utilisateur : `input(message)` (imprime `message` et renvoie ce que l'utilisateur a écrit), la saisie est forcément une chaîne de caractères qu'il faut convertir soi-même en cas de besoin (par exemple avec `eval`).

Ouvrir un fichier : `fichier = open(nom_du_fichier, mode)`, crée un « objet-fichier », où `nom_du_fichier` est une chaîne de caractères (avec l'extension). Argument optionnel `mode` : `"r"` par défaut pour lecture seule ("read"), donc il fichier non modifié. Autres modes ci-après.

Fermer un fichier après traitement (instruction `fichier.close()`, où `fichier` est la variable définie précédemment) !

Récupérer le contenu (toujours des chaînes de caractères) d'un fichier : `fichier.read()`, totalité du fichier à partir de la position du curseur de lecture, initialement au début). Ajouter un argument : définit le nombre maximal de caractères à lire. Utiliser `readline` au lieu de `read` : s'arrête en fin de la ligne actuelle, en incluant le caractère de fin de ligne. Utiliser `readlines` : retourne la liste des lignes restantes.

Pointeur de lecture : `fichier.seek(position, mode)` le place au `position`-ième caractère (si `mode` vaut `0`), `position` caractères plus loin (si `mode` vaut `1`) ou au `position`-ième caractère en partant de la fin (si `mode` vaut `2`).

Imprimer : `print(x)` quel que soit le type de `x`. Ensuite retour à la ligne (on peut finir par autre chose en ajoutant à la fin un argument `end=y`).

Imprimer plusieurs choses à la fois, avec des types totalement arbitraires, en mettant plusieurs arguments. Séparation systématique par une espace (on peut également modifier en ajoutant, également à la fin, un argument `sep=z`.

Imprimer dans un fichier : si ouvert en écriture (mode `"w"`, contenu original définitivement supprimé) ou en ajout (mode `"a"`, ce qu'on imprime est mis à la suite). On utilise `fichier.write(contenu)`, valeur de retour : nombre de caractères écrits.

Divers

Variables locales, globales, etc.

Quand une fonction manipule une variable, elle tente en priorité de la reconnaître comme locale (dont un argument), puis globale.

Mots-clés pour changer la portée : `global` et `nonlocal`.

Liaisons dynamiques

Définir une fonction `f` qui appelle une fonction `g` ne lie pas `f` à la version actuelle de `g`, mais fait exécuter `g` par `f` au moment de l'appel.

Fonctions locales, fonctions anonymes

Fonction locale : fonction définie dans une autre fonction. Elle n'existe pas en dehors.

Fonction anonyme : utilisable seulement pour l'expression en cours. Syntaxe : `lambda x_1, ..., x_n : resultat`, où les `x_i` sont les arguments.

Exceptions

Interrompre l'exécution du code manuellement ou suite à un comportement anormal prédéfini par l'interpréteur (par exemple diviser par zéro, déborder d'une séquence, utiliser une variable non instanciée, ...).

Exception au programme du tronc commun : assertion. Syntaxe : `assert condition`. Si la condition est fausse, erreur, sinon on passe à la suite.

Autres possibilités : `raise` puis un nom d'erreur.

Version classique : `raise ValueError("Pas content !")`.

Rattrapage d'exception, version complète :

```
try: # on se met dans un contexte où chaque exception est rattrapée
    code_avec_exceptions_possibles # bloc donc indentation et double-point
# le code ci-dessous est exécuté si une exception d'un des types est soulevée
except (un_type_d_exception, un_autre):
    autre_code_sans_que_les_nouvelles_exceptions_soient_rattrapees
# à l'instar du elif, ce code est exécuté si rien de ce qui précède ne l'est
# mais qu'une exception de ce nouveau type est soulevée
except un_autre_type:
    autre_code_idem
# toutes les autres exceptions sont rattrapées ici
except:
    encore_un_autre_code_idem
# si aucune exception n'est déclenchée, ce code est exécuté
else:
    code_pas_idem
# ce code est exécuté à la fin dans tous les cas
finally:
    code_final
# étonnamment, un return dans le finally remplacerait tout autre return
```

Seuls les blocs introduits par `try` et par `except` sont obligatoires.

Bibliothèques

Module (ou `bibliothèque`, autant dire que c'est la même chose) : ensemble d'instructions qu'on va pouvoir importer. Tout fichier Python en est un, si on veut.

Syntaxes possibles :

```
import mon_module # tout le module est disponible
mon_module.ma_fonction(mon_module.ma_variable)
```

```
import mon_module as momo
# momo sera un alias, et mon_module sera inconnu
momo.ma_fonction(momo.ma_variable)

from mon_module import ma_fonction, ma_variable
# le reste du module est inaccessible
ma_fonction(ma_variable)

from mon_module import * # tout est importé
ma_fonction(ma_variable)
```

Des bibliothèques et modules peuvent contenir eux-mêmes des sous-modules, qu'il faut parfois importer même si la bibliothèque ou le module l'a été.

Chapitre B)

Le langage OCaml

Introduction

Les données et leurs types

Les types de données, découverts par exemple en Python [1], ont une utilisation plus rigoureuse en OCaml, dans la mesure où le compilateur [2] 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.

Le typage en OCaml est dit *statique*, au contraire du typage *dynamique* d'autres langages. La différence réside dans le moment où un type est associé à chaque expression : respectivement avant ou pendant l'exécution. On retrouvera aussi le typage statique en C dans la déclaration du type (le `int` de `int x = 42;`) au moment de créer une variable, comme on le verra au moment d'aborder ce langage.

1. Cependant, le système des types de données est très différent entre les deux langages.
2. Les expressions ne sont pas à proprement parler interprétées en OCaml, mais ceci est une autre histoire ; *a contrario*, on n'a pas de compilateur en Python.

Pour autant, le type d'une expression n'est pas forcément précis. Il est déduit en OCaml par les opérateurs et fonctions, entre autres, impliqués dans l'expression, c'est ce qu'on appelle *l'inférence de type*, ce qui laisse parfois un doute. Par exemple, la fonction identité sera de type « fonction qui à une expression de type noté `'a` associe un objet de type `'a` » (on parle de *polymorphisme*). De même, l'opérateur de comparaison nécessite deux opérandes du même type, mais celui-ci peut être quelconque. En particulier, on n'a pas le droit de comparer `0` et `0.0`, ce qui risque d'être assez déstabilisant.

La gestion de la mémoire en OCaml est automatique. Inutile donc de s'inquiéter avec les allocations. En outre, il existe un ramasse-miettes pour libérer la mémoire qui n'est plus utilisée.

Types de base

Les types de base comprennent les entiers (`int`), les flottants (`float`) et les booléens (`bool`). Attention, les booléens, notés `true` et `false`, n'héritent pas de la structure d'entier et on ne peut donc pas les assimiler à `0` et `1`.

Les entiers sont encodés, du moins sur les ordinateurs habituels, sur 63 bits. Des dépassements arithmétiques peuvent donc intervenir autour de 10^{18}, justifiant l'existence d'un module pour gérer des entiers de précision arbitraire (hors programme). Les flottants sont évidemment aussi limités en précision et sujets aux dépassements et soupassements arithmétiques.

Plus précisément, les opérations spécifiques sont :
- `+`, `-`, `*`, `/` et `mod` pour les entiers [3].
- `+.`, `-.`, `*.`, `/.` et `**` pour les flottants [4].
- `&&`, `||` et `not` pour les booléens.

3. L'opérateur `mod` remplace donc le `%` de beaucoup d'autres langages. On pourra s'apercevoir que la division euclidienne a un comportement anormal sur les valeurs négatives, par exemple `-3/2` vaut `-1`, mais il n'est pas nécessaire de le savoir, et en cas de bug faire le test suffira à comprendre l'origine du problème. Quant au modulo, il est bon de savoir que `a mod (-b)`, en n'oubliant pas les parenthèses, équivaut à `a mod b`, mais que cette valeur est comprise entre `0` et `b-1` inclus si `a` est positif, et entre `-b+1` et `0` inclus si `a` est négatif.

4. 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.

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 [5], les affectations fonctionnant différemment comme nous le verrons plus tard) et <> (différence [6]).

L'évaluation des opérations booléennes est paresseuse. Pour rappel, cela signifie que l'évaluation de l'expression `b1 && b2` ignore `b2` si `b1` s'évalue à `false`, n'occasionnant ni les éventuels effets de bord ni les éventuelles erreurs. De même pour `b1 || b2` si `b1` s'évalue à `true`.

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, pas la partie entière).

Les caractères (`char`) forment un type de base à part entière, ils sont entourés d'apostrophes simples sans second choix. Il est interdit que le nombre de caractères entre les apostrophes ne soit pas exactement un (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, 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.

---

5. Au passage, l'opérateur `==` existe et son utilisation est rare : il détermine si les deux objets comparés ont la même adresse mémoire.

6. La même remarque s'applique quant à `!=`.

Une chaîne de caractères n'est plus mutable dans les dernières versions du langage. Pour simuler une chaîne de caractères mutable, les deux possibilités sont le type `bytes` ou, bien plus simplement, l'utilisation d'un tableau de caractères.

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.

Caractères et chaînes de caractères supportent la comparaison, aussi bien en ce qui concerne le test d'égalité que les tests d'infériorité. L'ordre total sur les caractères correspond à la comparaison de leur position dans la table de caractères utilisée, et l'ordre total sur les chaînes de caractères, s'appuyant sur l'ordre précédent, est l'ordre lexicographique, reprenant les règles de l'ordre alphabétique sans se limiter aux lettres. Ainsi, une chaîne `s` sera inférieure à une chaîne `t` si et seulement si `t` commence par `s` en entier ou s'il existe un entier naturel `k` tel que tous les caractères de `s` et de `t` jusqu'au `k`-ième inclus sont les mêmes et le `k+1`-ième caractère de `s` est strictement inférieur au `k+1`-ième caractère de `t`.

Séquences de base

La structure de *tableau* (`array`) fournit des séquences indexables d'éléments, avec une certaine rigidité : un tableau ne peut contenir que des expressions 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.[7]

On produit un tableau en délimitant les données par des **points-virgules** et en les entourant par `[|` et `|]`[8], 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)`

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

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

(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 modifiable. 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 classiques lorsqu'on déclare, dans un langage de programmation habituel, une variable de même fonctionnement qu'un tableau comme étant égale à une autre telle variable, sans faire de copie, existent en OCaml. De manière analogue, définir par exemple 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[9] (`list`) sont une structure spéciale, dans la mesure où elles ont un lien fort avec la récursivité.

Le plus simple est de donner une définition récursive : une liste est soit vide, soit la donnée d'un élément et d'une autre liste.[10]

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 `@`,

9. La structure algorithmique est celle d'une liste simplement chaînée. Le raccourci de liste est courant pour les langages fonctionnels.
10. À ce stade, on peut imaginer des listes infinies, et en fait... elles existent en OCaml.

dont les deux parties sont des listes à fusionner. Il faut cependant avoir à l'esprit que la fusion occasionne un coût linéaire en la taille de l'opérande de gauche.

Attention : L'expression `a::l` produit une nouvelle liste sans modifier `l`.

La longueur d'une liste s'obtient à l'aide de la fonction `List.length`[11].

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 se fera que par filtrage, ce sur quoi nous reviendrons. Il existe une fonction pour accéder au premier élément d'une liste non vide, dont on évitera l'utilisation : `List.hd`, et de même pour accéder au reste : `List.tl`. On comprendra bien que les fonctions `List.sub` et `List.make` n'existent pas.

Conversions

Les conversions entre types sont possibles (cf. `float_of_int`), et pour cela on dispose de multiples fonctions. Il est bon de les connaître, mais elles sont censées être rappelées en cas de besoin[12] :

- `int_of_string` et `string_of_int`, de même en remplaçant `int` par `float` ou `bool`.
- `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 en OCaml 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`, alors que `["",1]` sera une `(string * int) list`, en notant bien qu'elle comporte un seul élément, qui est un couple.

11. Attention, la fonction parcourt la liste, donc son coût est linéaire.
12. ... besoin qui n'est pas à la hauteur de l'abus de telles fonctions !

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

Types avancés

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`, `NULL` ou `nil` d'autres langages, car c'est le type des objets vides.

En parlant de `None`, cette expression existe aussi en OCaml, et correspond à un type composé, appelé `option`. Un type `'a option` propose une alternative : avoir quelque chose (constructeur `Some` suivi d'une expression de type `'a`) ou ne rien avoir (constructeur `None`).

Il est bien entendu possible de se passer totalement des types `option`, mais parfois cela arrange de les utiliser, et en tout état de cause il faut les connaître car ils sont désormais officiellement au programme.

Le dernier type prédéfini au programme est la *référence*, qu'on présentera au moment de mentionner la programmation impérative.

Variables

Cette information peut faire un choc : en OCaml, 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;;`. Comme dans la plupart des langages de programmation, un nom de variable doit commencer par une minuscule ou un `_` et contenir uniquement des chiffres, minuscules, majuscules et `_`.

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.

Programmation fonctionnelle en OCaml

Introduction

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

Une fonction est une expression qui attend un ou plusieurs arguments[13] et qui retourne une valeur.

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 l'expression (ou la séquence d'instructions, comme nous le verrons plus tard) après le symbole =, par exemple :

```
let module_carre (x, y) = x *. x +. y *. y;;
```

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

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

La fonction ci-avant a pour signature `module_carre : float * float → float`, et la console affiche `val module_carre : float * float → float = <fun>`.

Bien entendu, une fonction peut prendre une autre fonction en argument, nous en verrons plusieurs cas et en créerons également. On parle alors de *fonction d'ordre supérieur*.

Les arguments d'une fonction sont évalués avant que le code de la fonction ne s'exécute. En outre, ils sont passés *par valeur*, c'est-à-dire que la fonction travaille sur une copie de la mémoire d'où les arguments proviennent. Cela n'empêche cependant

13. 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`.

pas que les mutations d'un objet mutable passé en argument d'une fonction ne se répercute sur la mémoire. On pourra comparer le passage par valeur, le passage *par référence* et le passage *par affectation* pour en savoir plus.

Curryfication

Ce qu'il faut comprendre, dans la syntaxe précédente de la signature d'une fonction, c'est qu'en fait la fonction `nom` n'attend qu'un argument et retourne une autre fonction de signature `nom1 : type_arg2 → ... → type_retour`, pour laquelle le principe est le même. Il est donc tout à fait possible de définir une fonction partielle à partir d'une telle fonction en renseignant successivement des arguments.

Tester à ce sujet le code suivant :

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

let add3 = add 3;;
add3 4;;
```

Ceci ne serait par ailleurs pas possible avec une fonction qui prend en entrée un n-uplet d'arguments. Pour des raisons pratiques, on utilisera donc moins souvent ce dernier type de fonctions. On dit qu'une fonction attendant *en quelque sorte* plusieurs arguments est *curryfiée*[14].

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

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.

Disjonction de cas (if)

La disjonction de cas peut être utilisée en programmation fonctionnelle comme en programmation impérative, on en aura besoin très rapidement et on l'introduit ici.

14. du nom de l'informaticien Haskell Curry

La syntaxe de la disjonction de cas est `if condition then expr1 else expr2`, 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 expressions (pouvant se résumer à des instructions) soit le même.

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

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

Au passage, si les parenthèses offrent un souci de lisibilité, on peut les remplacer par `begin` et `end`. Ces mots-clés sont équivalents en tout point aux parenthèses, mais on ne peut pas apparier l'un d'entre eux à une parenthèse.

Filtrage

Une fonction peut aussi se définir par filtrage (exhaustif, sinon OCaml déclenche un avertissement) des cas, et on retiendra pour commencer la syntaxe suivante (une syntaxe alternative, non explicitement au programme, sera introduite au fur et à mesure sur des exemples) : `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 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.

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 *);;
```

```
let ou (b1, b2) = match (b1, b2) with (* cas avec un seul argument *)
|(true, _) -> true
|(_, true) -> true
|_ -> false;;
```

Attention, les signatures dépendent de la façon dont les arguments sont présentés. On retrouve la notion de fonction curryfiée, sachant que dans le filtrage même une fonction curryfiée nécessitera de rassembler les arguments en un n-uplet.

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

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 n'est pas mentionnée dans le programme, mais vu son utilité elle sera également introduite progressivement.**

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 (mais inutilement lourd, il s'agit surtout d'un premier exemple illustrant la syntaxe avec `when`) :

```
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 (* Interdiction d'utiliser deux fois un nom *)
|_ -> true;;

let bon_xor b1 b2 = match b2 with
|a when a = b1 -> false
|_ -> 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 OCaml affecte dans la foulée les variables correspondant aux arguments quand il trouve le motif. On peut voir cela comme une version plus puissante du `switch` de certains langages.

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 OCaml comme correspondant au filtrage le plus intérieur ; un parenthésage pourra s'imposer.

Il s'avère également qu'on peut capturer plusieurs constantes à la fois dans un cas de filtrage (mais avec une gestion compliquée si des variables interviennent), 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";;
```

Programmation impérative en OCaml

Attention : En pratique, la notion d'instruction n'existe pas vraiment en OCaml. Elle sera utilisée ici par abus, mais il est bon de garder à l'esprit que tout est expression, et ce qu'on assimilera à une instruction est à considérer comme une expression impure.

En OCaml, 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.

Avant de commencer, les références

Une référence en OCaml 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` (ce sera souvent localement), mais comme `i` sera de type `int ref`, on ne pourra pas faire de calculs sur `i` directement, il faudra déréférencer pour accéder à la valeur correspondante, en écrivant `!i` (qui sera cette fois 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 type référencé, par exemple `i := 42`, cette nouvelle valeur pouvant dépendre de l'ancienne.

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 s'imaginent le temps de maîtriser suffisamment la programmation fonctionnelle).

Séquence d'instructions

Il est possible d'enchaîner deux instructions au sein d'un même morceau de code, ce qui nécessite de les séparer par un point-virgule. Cependant, OCaml 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 déjà mentionnées sont les modifications de références et d'éléments d'un tableau).

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

Plus précisément, si on souhaite provoquer un effet de bord en se servant d'une

fonction retournant 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, 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.

OCaml 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[16], et une bonne recommandation est **d'indenter comme si, à l'instar de Python, le fonctionnement du programme en dépendait**.

Complément sur la disjonction de cas

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 être de type `unit` dans ce cas.

Cela correspond à « si cette condition est vérifiée alors faire ceci, sinon ne rien faire ».

Boucle inconditionnelle (for)

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. **Les bornes sont incluses.**

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[17].

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 print_int (i / 0) done` ne provoquera pas d'erreur de division par zéro.

15. OCaml dispose de la fonction `ignore` permettant aussi d'écrire `ignore(f(x)); suite`, ce qui est sans doute plus agréable à lire.
16. Ou plutôt : souhaite ne pas avoir à s'y intéresser...
17. Pire que cela : l'évaluation de `for i = 0 to 0 do 2+2 done;;` ne donnera même pas `4`.

Puisque les seuils sont évalués une fois pour toutes avant d'entrer dans une boucle inconditionnelle, on ne peut pas perturber une boucle, par exemple la boucle `let n = ref 10 in for i = 1 to !n do decr n done;;` sera effectivement parcourue dix fois (et la valeur de `!n` sera bien `0` après la boucle).

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 [18].

Il convient de signaler à présent que la définition d'une fonction peut faire intervenir, entre autres, des séquences ou des blocs d'instructions, et la valeur retournée est la dernière instruction rencontrée. Par exemple :

```
let fact n = let res = ref 1 in
    for i = 2 to n do res := !res * i
    (* Ici le corps de boucle, rien ne peut être retourné *)
    done;
    !res;; (* Ici la valeur de retour *)
```

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`. La condition est évaluée avant chaque tour dans la boucle.

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` $\rightarrow$ `unit`),

18. Et même dans ce cas, on ne fait que la masquer (notion de "shadowing") : 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.

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` → `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` → `in_channel` et `string` → `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.

Compléments

Attention : De nombreux passages de cette section sont essentiels et seront traités en cours, il ne s'agit pas d'une collection de notions juste pour aller plus loin.

Définitions simultanées

Les définitions peuvent aussi être simultanées, en se servant du mot-clé `and`.[19] Dans ce cas, si l'un des éléments dépend de l'autre, OCaml 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;;
```

Liaisons statiques

Une différence entre Python (entre autres) et OCaml en ce qui concerne les définitions de fonctions est que le code est évalué en OCaml 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 OCaml imprimera deux fois `1` :

19. 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();;
```

Ainsi, lorsqu'une définition utilise une variable globale, c'est la valeur de cette variable au moment de la définition qui est prise en compte.

Fonctions locales, fonctions anonymes

Puisque le langage OCaml permet de faire des définitions locales, il est tout à fait envisageable de définir des fonctions dans des fonctions.[20]

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 OCaml qu'à Python, se combinent bien avec des fonctions comme `map` ou `iter`, du module `List`, que l'on verra en TD.

Types somme et produit

En OCaml, il est possible de créer ses propres types, et nous verrons trois façons bien distinctes. Le mot-clé pour la définition est `type`[21].

Renommage

La première façon est simplement de renommer des types déjà existants.

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

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

Bien entendu, OCaml 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);;`.

Type somme

La deuxième façon, donnant des *types sommes*, 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).

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);;`.

La mention `'a`, normalement déjà aperçue sur des fonctions ou types préexistants, témoigne du polymorphisme (notion déjà abordée) du type ainsi construit.

Plus complexe et récursif : `type t1 = A | B of t2 and t2 = C | D of t1;;`, dont un élément est `B(D(A))`.

Les types `mybool` et `mylist` laissent deviner que les types somme s'associent harmonieusement au filtrage.

On écrira donc par exemple :

```
let mytete maliste = match maliste with
| Vide -> failwith "Tete"
| Cons(t, q) -> t;;
```

Type produit

La troisième façon, donnant des *types produits* ou *enregistrements* (*record*), se rapproche de la programmation objet. En pratique, un type produit contient des « rubriques » [22], 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 ; ...};;
```

On accède alors 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; mutable prenoms : string list;
mutable age : int; mutable sexe : bool};;
```

22. pour reprendre le terme officiel de la documentation

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

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

Exceptions

Introduction

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 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.

Création d'exceptions

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.

Un cas simple dans lequel on souhaite créer une exception paramétrée est la recherche de la position d'un élément particulier dans un tableau. Le nom de l'exception sera alors `Trouve`, en notant la majuscule obligatoire là aussi, et le paramètre sera l'indice où l'élément sera trouvé, d'où la syntaxe `exception Trouve of int;;`.

Pour déclencher une exception (la traduction littérale est « soulever »), on la précède du mot-clé `raise` [23].

23. Un équivalent de `failwith "paf"` est donc `raise (Failure "paf")`.

Rattrapage d'exceptions

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 : OCaml évalue `code`, et retourne sa valeur finale ; si une exception est déclenchée, OCaml 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 [24].

Ainsi, la recherche de la première position d'un caractère dans une chaîne pourra utiliser l'exception personnelle mentionnée ci-avant :

```
exception Trouve of int;;

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

C'est à peu près ainsi qu'on pourra simuler le mot-clé `return`.

24. Ou, comme on l'a vu, certains peuvent être `exn`.

Quelques exceptions très habituelles en OCaml :

```
Uncaught exception: Division_by_zero (* 1/0 *)
Uncaught exception: Failure "hd" (* List.hd [] *)
Uncaught exception: Failure "tl" (* List.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. Ce sera par ailleurs la seule exception à connaître absolument.

Modules

Bien que la plupart des fonctions, types et autres objets utiles soient dans le module de base, OCaml 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`. On notera que les noms de module commencent eux aussi nécessairement par une majuscule.

Dans ce cas, pour utiliser une fonction (ou quoi que ce soit d'autre) sans ouvrir le module, il faut la préfixer par le nom du module suivi d'un point [25].

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 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 du langage OCaml sont présentés en TP.

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

Formats

Il n'est pas nécessaire de maîtriser cette section hors programme, mais le besoin d'imprimer de manière fluide pour déboguer justifie que la notion soit abordée.

Les fonctions d'impression vues dans la section sur les entrées et sorties ont pu décevoir par leur faiblesse par rapport à l'impression dans d'autres langages.

Pour pallier ce manque de fonctionnalités, OCaml fournit néanmoins la possibilité d'utiliser des formats pour une impression de chaînes de caractères contenant des valeurs paramétrées.

Ainsi, le module `Printf`[26] 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 du langage OCaml, on obtient une fonction qui attend les valeurs manquantes en tant qu'arguments.

Exemples :

```
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;;
```

26. issu du langage C

```
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;;
```

L'essentiel

Malgré le fait que le nouveau langage à apprendre diffère bien des autres langages déjà rencontrés, il ne faut pas voir ceci comme un obstacle insurmontable.

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, typiquement lors de la manipulation d'entiers ;
- 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, ni oublier les points-virgules qui séparent plusieurs instructions (si on veut copier la syntaxe de Python, on peut faire une instruction par ligne, avec le réflexe de finir les lignes par des points-virgules quand elles s'imposent) ;
- 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 ;
— mélanger les langages, par exemple avec `for i in range n` et autres expressions qui sont à bannir.

Chapitre C)

Le langage C

Introduction

Le langage C est un langage dit *de bas niveau*, ce qui signifie qu'il est proche du langage machine, avec comme corollaire implicite qu'il impose de gérer des choses que les programmeurs dans des langages de haut niveau ont tendance à laisser de côté, notamment la gestion de la mémoire.

Par ailleurs, les programmes écrits en C sont réputés produire des exécutables fonctionnant bien plus rapidement que des programmes écrits dans un langage de plus haut niveau, ce qui se voit aussi sur les fonctions de Python qui ont été écrites directement en C par rapport à des surcouches ayant été mises dans des bibliothèques après avoir été écrites en Python.

Par rapport à l'assembleur (dont on pourrait dire qu'il est d'encore plus bas niveau), le langage C a l'avantage d'être utilisable de la même manière sur des architectures différentes, au prix d'adaptations minimes voire sans adaptation nécessaire.

Contrairement à Python (entre autres), le langage C est un langage compilé, dont les fichiers sources (d'extension .c) permettent de produire des exécutables, pouvant éventuellement être lancés à la volée suite à la compilation, notamment pour la phase de tests.

À la différence des programmes écrits en OCaml, langage bien plus strict, les programmes dont la compilation a réussi en C peuvent encore fourmiller de bugs et autres soucis, car il peut y avoir de nombreux comportements indéfinis (*UB* en anglais, pour *undefinied behaviour*). Un code C qui est considéré comme *UB* n'est pas spécifié dans la norme actuelle et peut déclencher une erreur, mais aussi (ce qui est en fait plus grave) fonctionner de manière inattendue et potentiellement indétectable.

Le programme définit un cadre strict et limité sur l'utilisation de C, et rester discipliné permettra d'éviter de s'aventurer dans l'inconnu.

Structure d'un programme en C

Un fichier .c contient nécessairement une fonction principale, justement appelée `main`, qui est lancée par l'exécutable produit par la compilation.

Comme on ne fera presque jamais de programme sans impression, on commencera toujours le fichier par le chargement de la bibliothèque contenant les fonctions d'entrée-sortie. Ce chargement (comme tous les chargements en adaptant le nom) se fait par l'instruction

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

La déclaration de la fonction `main` suit le modèle des déclarations de fonction, en annonçant le type de la valeur de retour avant le nom de la fonction, la parenthèse mentionnant les éventuels arguments dont le type est également précisé avant chaque nom puis le délimiteur de bloc associé à la fonction. La partie avant ce délimiteur est appelé le *prototype* de la fonction. En C, les blocs sont délimités par des accolades.

Ainsi, un programme minimaliste qui se contente d'imprimer un message sera :

```
#include <stdio.h>

int main()
{
  printf("Bonjour !\n");
  return 0;
}
```

On notera le mot-clé `return` dont on évitera de profiter de l'usage implicite en son absence.

Conventionnellement, on demande à la fonction `main` de retourner `0` pour signaler que tout s'est bien passé (et donc autre chose sinon), d'où le typage par `int`.

L'impression a été déclenchée par la fonction `printf`, qui a été récupérée par le langage OCaml. Cet appel à la fonction `printf` est une instruction, et chaque instruction se termine par un point-virgule en C.

On notera que le point-virgule n'a pas le même rôle qu'en OCaml, malgré la confusion possible, car dans ce dernier langage il sert de délimiteur entre une pseudo-instruction et une autre expression.

Comme en Python, le mot-clé `return` déclenche l'arrêt d'une fonction et le retour de la valeur associée.

Il existe aussi une fonction appelée `exit`, disponible dans la bibliothèque `stdlib.h`, pour interrompre le programme en entier. On s'en servira plus fréquemment en C qu'en Python, et son rôle peut être considéré comme équivalent à `return` quand elle est utilisée dans la fonction `main`.

La gestion de l'indentation et des retours à la ligne dépend de conventions personnelles, tant que la lisibilité est favorisée, car le compilateur ignore les espaces excédentaires et le découpage des lignes. En ceci, il n'y a pas de différence avec OCaml.

Compilation

Le compilateur classique pour le langage C, ainsi que quelques autres langages de bas niveau, est *gcc* sous Linux. Ce compilateur est disponible sous Windows au prix de l'installation de quelques programmes annexes, car Windows n'a pas son propre compilateur pour le langage C.

L'instruction la plus simple en ligne de commande pour demander la compilation d'un fichier, ici nommé `source.c`, est alors :

```
gcc source.c -o executable
```

En dehors d'un problème causé par des erreurs dans le code, un fichier nommé `executable` est alors produit, et peut être lancé par la console (ou éventuellement par une fenêtre graphique).

Afin de produire un code de meilleure qualité, il est bon d'utiliser l'option d'affichage de « tous » les avertissements à la compilation, qui est `-Wall`. On comprendra "Warnings : all", et non pas une quelconque histoire de mur...

Tous les avertissements sont-ils inclus ? En fait non, et une deuxième option pour ajouter des avertissements supplémentaires, dont la mention qu'on est en train d'effectuer une comparaison entre objets de type différent (par exemple un entier non signé et un entier signé), est `-Wextra`.

Pour les accès illicites à la mémoire, l'option est `-fsanitize=address`. Il est recommandé d'utiliser les trois, ce qui donne :

```
gcc -Wall -Wextra -fsanitize=address source.c -o executable
```

Types de base

Introduction

Comme pour OCaml, le typage de C est *statique*, ce qui fait que les variables ont un type connu à l'avance, et le non-respect de ce type sera une source de problèmes à la compilation avant même de pouvoir constater comment une exécution se déroulerait... ce qui est compliqué quand l'exécutable ne peut pas être construit.

Entiers

Le type déjà rencontré dans le premier exemple de ce chapitre est le type `int`, formé par les entiers sur 32 bits dans les versions actuelles.

La bibliothèque `stdint.h` introduit des types supplémentaires pour les entiers, de tailles différentes voire non signés (si le nom du type commence par la lettre 'u'), qu'on retrouve par ailleurs dans la bibliothèque `numpy` de Python : `int8_t`, `uint8_t`, `int32_t`, `uint32_t`, `int64_t` et `uint64_t`. Les types `int` et `int32_t` contiennent les mêmes valeurs, mais mieux vaut ne pas mélanger deux entiers de ces deux types, à plus forte raison pour deux autres types différents.

Sans utiliser `stdint.h`, on peut tout de même créer des entiers non signés sur 32 bits, en préfixant le mot `int` par le mot `unsigned`.

Les opérations sur les entiers reprennent le principe du langage OCaml : deux entiers donnent un entier. Ainsi, l'opérateur `/` fait une division euclidienne. Par ailleurs, la gestion des nombres négatifs est la même dans les deux langages et dévie donc de l'usage mathématique. L'opérateur pour le reste de la division euclidienne est %, comme en Python.

Contrairement à Python, le nombre de bits réservé pour un entier est prédéfini, et il n'y a pas fusion entre les entiers et les entiers longs. On prendra garde à éviter les dépassements arithmétiques, qui occasionnent dans certains cas un comportement indéfini, ce que le langage OCaml compense en calculant tout modulo le nombre d'entiers représentables.

Flottants

Les flottants existent sous deux types : `float` et `double`, respectivement sous 32 et 64 bits. On n'utilisera que ces derniers pour des raisons d'efficacité.

Les opérations utilisent les mêmes symboles sauf % qui est retiré.

Mélanger un entier et un flottant donne un flottant, mais il est recommandé de convertir au préalable.

À ce propos, le type d'une variable ne peut pas changer précisément en raison des principes du typage du langage, donc on peut utiliser une expression, stockée ou non dans une autre variable, obtenue par *cast* vers un autre type, en écrivant devant la valeur castée le nom du nouveau type entre parenthèses. **Ceci est valable pour tous les types, avec des comportements prévus ou non en fonction des types.**

```
float xf = 4.2;
int x = (int) xf;
```

Caster un flottant vers un entier tronque à la manière de la fonction `int` en Python et la fonction `int_of_float` en OCaml.

Booléens

Les booléens sont absents de la bibliothèque standard mais introduits par `stdbool.h` en tant que `true` et `false`, respectivement équivalents à un et zéro. En pratique, sans utiliser la bibliothèque en question, manipuler les entiers associés est imaginable (mais nous ne le ferons pas, pour la propreté du code et pour garder de bonnes habitudes par rapport à d'autres langages), car les opérations booléennes sont disponibles, quant à elles.

Ces opérations booléennes sont `&&`, `||` et `!`, reprenant les opérations vues en OCaml à la différence près que la négation utilise le symbole point d'exclamation devant la valeur à nier. L'évaluation paresseuse des opérations booléennes existe aussi en C, avec les mêmes conséquences.

Les priorités sont également les mêmes que classiquement : la négation d'abord, et les conjonctions priment sur les disjonctions.

Les comparaisons produisant les booléens sont celles de Python : `==`, `!=`, `<`, `>`, `<=` et `>=`, et le simple symbole égal est réservé à l'affectation.

Attention : les comparaisons ne s'enchaînent pas comme en Python : `0 <= x < 4` est toujours vrai en C, car `0 <= x` s'évalue prioritairement en un booléen, assimilé à `0` ou `1`, qui est comparé à `4` et effectivement strictement inférieur.

Caractères

Le langage C dispose d'un type à part pour les caractères, à savoir le type `char`. Le délimiteur des caractères est l'apostrophe simple, comme en OCaml, avec la possibilité de créer les mêmes séquences échappées. La taille en mémoire de chaque caractère est d'un octet.

Les caractères sont en pratique associés à leur code ASCII sur 8 bits et supportent l'addition entre eux et avec des entiers pour un résultat pris modulo 256, mais le programme de CPGE demande de s'en tenir à leur utilisation comme éléments constitutifs des chaînes de caractères.

Les chaînes de caractères seront abordées ultérieurement, en raison de leur structure particulière.

Le vide (attention, ce n'est pas un type)

Les fonctions n'ayant pas de `return` ou ayant un `return` vide en C ont un prototype mentionnant `void` à la place du type de retour. Il s'avère que `void` n'est pas un type, ce qui justifie en particulier qu'une fonction sans argument n'ait pas besoin de préciser de `void` entre les parenthèses.

Variables et constantes

Contrairement à Python et OCaml, le langage C permet de déclarer des variables sans les initialiser. Il s'agit en un sens de réserver la place dans la mémoire correspondant à la taille commune des éléments du type de la variable sans écrire sur l'ensemble de la plage réservée.

La déclaration d'une variable (pour nous, ce sera une déclaration par instruction) se fait par une instruction où le type est écrit, suivi du nom de la variable, puis du point-virgule qui doit terminer chaque instruction.

L'initialisation d'une variable avec la déclaration consiste à ajouter après le type et le nom de la variable le symbole égal, puis la valeur initiale de la variable.

Une affectation de variable déclarée, ou une réaffectation si la variable a déjà une valeur, se fait comme en Python par la syntaxe `nom = valeur`. **En particulier, on ne redéclare pas la variable et donc on ne réécrit pas le type.**

On retrouve les raccourcis `+=`, `-=`, etc. pour la mise à jour à partir de l'ancienne valeur, comme en Python. Les opérateurs `++` de pré-incrémentation (ou post-incrémentation, suivant la position) et `-` pour la pré-décrémentation ou la post-décrémentation ne sont cependant pas au programme et leur utilisation est découragée en CPGE, avec un seul cas de tolérance dans notre cours précisé plus loin.

Quant aux constantes, il s'agit de « variables » qu'on s'interdit de modifier par la suite, ce qui est garanti par le compilateur qui déclencherait une erreur pour toute tentative de réaffectation (même avec des pointeurs, notion abordée plus loin).

On déclare une variable comme constante en ajoutant le mot-clé `const` avant la déclaration de type au moment de l'initialisation de la constante.

L'utilité des constantes est de pouvoir s'en servir en tant que taille d'une structure, cette taille étant alors connue à la compilation (l'intérêt de ce fait est traité dans le chapitre sur la mémoire).

En raison du passage par valeur, quand on appelle une fonction sur une variable déclarée comme constante, l'argument lui-même de la fonction n'a pas de raison de garder le caractère constant, et le mot-clé `const` est à ajouter, si nécessaire, au prototype aussi.

Structures de contrôle

Puisque chaque instruction doit être terminée par un point-virgule, rien n'empêche d'enchaîner les instructions en les écrivant à la suite.

Une instruction composée, que nous appellerons aussi bloc d'instructions, est une partie du programme de taille arbitraire entourée d'accolades. Il est important de remarquer que **toute variable déclarée dans une instruction composée n'a pas d'existence en dehors de celle-ci, même lorsque l'instruction composée n'est pas introduite par une boucle ou un test (cette syntaxe est autorisée, même si son usage est a priori moins pertinent)**.

Ainsi, si on doit créer une variable ayant deux valeurs possibles suivant le résultat d'un test, il faut déclarer la variable avant le test et l'affecter ou la réaffecter dans les deux instructions (composées ou non) introduites par le test, ou utiliser une *expression conditionnelle* (hors programme, avec la syntaxe `b ? ey : en` s'évaluant à `ey` si `b` est vrai et à `en` sinon).

Test conditionnel (`if`)

Un test se fait suivant la syntaxe suivante :

```
if (condition)
  bloc1
else
  bloc2
```

Il est important d'insister que **les parenthèses délimitant la condition sont à écrire absolument**.

La nature de `bloc1` et `bloc2` est une instruction composée, même dans les cas où une seule instruction ne figure, et l'utilisation systématique d'accolades pour ce cas également est encouragée.

Quoi qu'il en soit, puisque même une seule instruction sera considérée comme un bloc, on ne pourra pas déclarer de variable en espérant que sa portée soit en dehors du bloc, même en l'absence d'accolades, pour faire écho à ce qui précédait cette sous-section.

Il n'est pas nécessaire d'écrire la partie introduite par `else`, et l'absence d'équivalent à `elif` nécessite d'enchaîner, si plusieurs conditions successives sont à tester, de la manière suivante :

```
if (condition)
  bloc1
else if (conditionbis)
  bloc2
else
  bloc3
```

Ici, le premier `else` n'aura pas d'accolades pour ne pas alourdir la syntaxe.

La sémantique est intuitive et reprend celle des tests vus dans d'autres langages.

Utiliser autre chose qu'un booléen pour la condition donne un comportement proche de celui de Python, et il s'agit d'une pratique à éviter en règle générale en C aussi. On retiendra pour la culture et pour lire d'obscurs codes qu'en gros seules les valeurs égales à `0` sont interprétées comme fausses en tant que booléens.

Boucle conditionnelle (`while`)

La syntaxe de la boucle conditionnelle est :

```
while (condition)
  bloc
```

Les mêmes remarques que pour le test conditionnel s'appliquent concernant les parenthèses et les accolades.

Si au premier test la condition n'est pas remplie, la boucle ne s'exécutera jamais. Il existe en pratique une syntaxe (hors programme et qu'on n'utilisera pas dans notre enseignement) permettant de forcer au moins un tour de la boucle :

```
do
  bloc
while (condition);
```

Boucle prétendument inconditionnelle (`for`)

La boucle inconditionnelle que l'on connaît en Caml n'existe pas en tant que telle en C, pas plus que la boucle faisant un parcours comme en Python et qui est introduite dans d'autres langages par un `foreach`.

En pratique, une boucle introduite par `for` est un sucre syntaxique pour une boucle conditionnelle, ce qui ne permet pas de déduire la terminaison de la boucle sans exhiber de variant.

La syntaxe est la suivante :

```
for (initialisation; condition; maj)
  bloc
```

Il est quasiment incontournable de créer une nouvelle variable dans la partie d'initialisation, la condition reprend l'idée du while et la partie finale est une instruction effectuée après chaque tour de boucle et avant la vérification de la condition pour lancer éventuellement le suivant. Ainsi, un exemple classique d'utilisation sera :

```
for (int i = 0 ; i < 10 ; i += 1)
  bloc
```

Avec un bloc bien écrit, on pourrait envisager une boucle dont l'introduction se ferait par `for (;;)` mais par souci d'esthétique on s'en abstiendra. L'intérêt d'utiliser `for` étant justement la lisibilité, ce serait contre-productif.

Si la variable est déjà déclarée par ailleurs, on peut simplement l'initialiser ou la réinitialiser, donc sans le rappel du type (voir plus haut). Cependant, simplement écrire son nom n'a aucun intérêt et consiste en une instruction ignorée.

La mise à jour, si elle consiste à ajouter (respectivement retirer) un à la variable numérique `i`, peut s'écrire `i++` (respectivement `i-`), voire `++i` (respectivement `-i`), et ce sera le seul cas d'utilisation autorisée de ces opérateurs, dont la différence entre la notation avant et après le nom de la variable ne sera d'ailleurs même pas expliquée.

Attention par ailleurs, il ne s'agit pas d'instructions mais d'expressions, et c'est bien là la difficulté que le programme cherche à contourner en laissant l'opérateur de côté.

Une boucle, conditionnelle ou non, peut être interrompue à tout moment par l'instruction `break;`, qui fait sortir du niveau le plus profond actuel.

L'instruction `continue`, qui n'est pas au programme de son côté, permet de passer directement à l'étape suivante de vérification (après la mise à jour dans le cas du `for`).

Compléments sur les fonctions

Le langage C gère naturellement la récursion, mais les différences entre Python et C sur lesquelles le doigt a déjà été mis font que la récursion mutuelle nécessite une subtilité. Il s'agit de ne pas utiliser une fonction avant qu'elle ne soit déclarée.

Ainsi, avant d'écrire le code d'une fonction qui en appellera une autre, il faut écrire le prototype de cette autre (éventuellement des deux), puis les deux fonctions peuvent être écrites normalement.

Une instruction introduite par `return` interrompt l'exécution d'une fonction et renvoie la valeur associée, dont le type doit être celui renseigné dans le prototype de la fonction.

Les fonctions en C, comme en OCaml, passent leurs arguments par valeur. C'est donc une copie des arguments qui est utilisée dans le code de la fonction, avec cependant la possibilité de modifier les variables associées par d'autres moyens que nous verrons dans la section suivante.

Le passage d'une fonction comme argument d'une autre nécessite la notion à suivre de pointeur, donc il sera présenté au moment opportun.

Pointeurs

Un *pointeur* en C est une variable pointant sur une adresse mémoire. Celle-ci est un entier dont le nombre de bits dépend du processeur (à l'heure actuelle, on peut s'attendre à ce que ce nombre soit 64).

L'idée derrière les pointeurs est à rapprocher de la notion de référence en OCaml, mais l'usage ne sera pas le même à plusieurs titres, pour commencer dans la mesure où on n'aura pas besoin de pointeurs en C à chaque occasion où un programme impératif similaire en OCaml utilisera une référence.

N'importe quel objet `x` de n'importe quel type `t` peut être stocké dans une adresse mémoire `a` qui sera la valeur d'un pointeur `p`. Le pointeur sera alors considéré en C comme de type `t*`, et on associera en pratique `p` à `&x` et `x` à `*p`, avec les opérateurs respectifs de *référencement* et de *déréférencement*.

Un pointeur peut être redirigé vers une autre adresse mémoire, mais ce genre d'instructions se rencontrera plutôt dans des programmes erronés qui ne font pas ce qu'on pensait, par exemple suite à une erreur classique d'oubli de l'étoile. En particulier, il n'y a pas d'intérêt a priori à affecter à un pointeur l'adresse mémoire déjà présente dans un autre.

Pour déclarer un pointeur sans l'initialiser, il faut lui allouer de la mémoire, ce qui nécessite la fonction `malloc`, présente dans `stdlib.h` qu'il faudra importer. L'argument est un nombre d'octets, qu'on n'a pas à connaître par cœur ni adapter en changeant d'architecture, si on utilise `sizeof(t)` où `t` est le type de la valeur à l'adresse pointée.

Utiliser `sizeof` sur une valeur et non un type fonctionne, mais ce n'est pas la méthode recommandée.

On rappelle que la mémoire allouée est à libérer, manuellement si nécessaire, une fois le travail terminé, et la fonction associée est `free` en C.

En pratique, l'arrêt du programme libère toute la mémoire allouée, donc on verra peu de `free` dans un premier temps, ce qui n'empêche pas de les écrire néanmoins afin d'acquérir de bons réflexes.

Attention, un pointeur créé dans une fonction sera considéré comme une variable locale, et la mémoire qui lui est allouée est libérée quand la fonction s'arrête, sauf si la mémoire est allouée explicitement avec `malloc`. En tout cas, dans le cas contraire, retourner le pointeur pourra donner un comportement indéterminé.

Dans un chapitre ultérieur, une réalisation en C de structures chaînées ou arborescentes sera l'occasion d'introduire le pointeur `NULL`, qui est prédéfini mais il est interdit de tenter d'accéder à une valeur référencée par ce pointeur. Les opérations permises avec `NULL` sont donc des comparaisons entre pointeurs. Ce pointeur est disponible dans l'en-tête `stddef.h`, qu'on pourra aussi supposer disponible dans les programmes qu'on écrit.

Un pointeur peut également pointer sur une fonction, car toute fonction a en pratique une adresse mémoire. La syntaxe est la suivante, en notant que **toutes les parenthèses utilisées sont nécessaires** :

```
#include <stdio.h>

int f(int x)
{
  return x * x;
}

int g(int x)
{
  return 2 * x + 1;
}

int compose(int (*f2)(int y), int (*f1)(int z), int x)
{
  return (*f2)((*f1)(x));
}

int main()
{
  printf("%d vs. %d\n", compose(&g, &f, 3), compose(&f, &g, 3));
  return 0;
}
```

Le pointeur vers une fonction s'obtient avec l'esperluette, et la priorité des opérateurs ou le risque d'ambigüité de la syntaxe (qu'on pourrait confondre avec un prototype) fait que si `*f` est un pointeur de fonction, on l'utilisera toujours entre parenthèses.

Quand ce pointeur est un argument, on le précède du type de retour et on précise entre parenthèses le type des arguments. Les noms de ces arguments n'ont pas de rôle, en témoigne l'utilisation de `y` et `z`, non mentionnés ailleurs (utiliser `x` dans les deux cas aurait fonctionné aussi).

Pour retourner une fonction, c'est un peu plus technique (et en dehors des compétences attendues), donc on évitera d'avoir besoin de recourir à ceci :

```
#include <stdio.h>

int f(int x)
{
  return x * x;
}

int g(int x)
{
  return 2 * x + 1;
}

int (*compose(int (*f2)(int y), int (*f1)(int z)))(int x)
{
  int f2of1(int x)
  {
    return (*f2)((*f1)(x));
  }
  return f2of1; // Avec ou sans étoile
}

int main()
{
  printf("%d vs. %d\n", compose(&g, &f)(3), compose(&f, &g)(3));
  return 0;
}
```

Expliquons `int (*compose(int (*f2)(int y), int (*f1)(int z)))(int x)` :

- Le premier `int` signifie que la fonction renvoyée par cette fonction (en l'occurrence le pointeur de fonction renvoyé...) renvoie un entier.
- Le dernier `(int x)` signifie que cette même fonction prend un entier en argument. Pour être complet, cela marche aussi sans même écrire `x`.
- La partie entre parenthèses est le prototype de la fonction permettant d'obtenir le pointeur de fonction, il couvre tout le reste : On commence par donner le nom de la fonction, précédé d'une étoile, puis on donne entre les parenthèses à suivre les arguments, ici, des fonctions avec la syntaxe de l'exemple précédent.

Attention, ce code ne fonctionne pas avec tous les compilateurs (`gcc` l'accepte). Une fonction locale est renvoyée, alors qu'elle n'est pas censée survivre à la fonction qui l'a créée.

La raison pour laquelle l'étoile est omise dans `return f2of1;` est qu'une fonction et le pointeur associé sont assimilés dans ce cas. Pour la même raison, les esperluettes n'étaient pas nécessaires ni les étoiles devant `f2` et `f1`. Cependant, l'absence de celle devant `compose` dans le prototype aurait déclenché une erreur.

Un autre exemple pour renvoyer une fonction avec des arguments plus simples :

```
#include <stdio.h>

int f_impair(int x) { return x * x * x; }
int f_pair(int x) { return x * x; }

int (*f(int num_fonction))(int x)
{
  if (num_fonction % 2 == 0) return &f_pair;
  else return &f_impair;
}

int main()
{
  printf("%d", (*f(42))(42));
  return 0;
}
```

Tableaux

Généralités

En C comme en OCaml, la structure de tableau existe et permet de collecter des valeurs de même type avec la possibilité d'accéder à n'importe laquelle de ces valeurs à l'aide d'un indice.

Comme l'intuition le suggère, le premier indice est zéro, et en l'occurrence un tableau se comporte comme un pointeur vers cet indice, mais nous ne tiendrons pas compte de ce fait et en particulier l'arithmétique de pointeur est hors programme.

La déclaration d'un tableau se fait en déclarant le type des éléments du tableau, son nom puis sa taille entre crochets (ce qui permet de déduire la taille mémoire nécessaire). On écrira par exemple `int t[10];` pour déclarer un tableau de dix entiers non initialisé.

Par la suite, l'initialisation se fera élément par élément, car **on ne peut pas mettre un tableau à gauche du signe égal d'affectation en C**. L'accès à un élément d'un tableau, que ce soit pour s'en servir ou pour une affectation, se fait par le nom du tableau suivi de l'indice entre crochets, donc par la même syntaxe qu'en Python.

Les tableaux multidimensionnels existent, il suffit de faire des tableaux de tableaux, et on aura par exemple `int M[3][2]` pour une matrice avec trois lignes de deux colonnes.

Les indices légitimes sont entre zéro et la taille du tableau moins un, sans pouvoir accéder aux derniers éléments à l'aide d'indices négatifs. Cependant, **aucune vérification n'est faite au niveau des indices fournis**, donc tenter d'accéder à un indice qui déborde, même un indice négatif, occasionnera un comportement imprévu sans erreur systématique, d'autant plus que **la taille d'un tableau n'est pas une donnée disponible ni récupérable**.

Pour cette raison, comme la taille est censément connue du programmeur, elle peut être nécessaire en tant qu'argument supplémentaire lorsqu'un tableau est passé en argument d'une fonction. Cette pratique n'est cependant pas à transposer à d'autres langages.

Un tableau peut en pratique aussi être initialisé en renseignant ses (premières) valeurs, séparées par des virgules et entourées d'accolades.

Attention : la partie entourée d'accolades est un initialisateur et non pas un tableau, et il n'existe pas de tableaux constants.

On ne peut alors pas manipuler de données brutes comme on pourrait s'amuser à faire `[|4; 3; 2|].(1)` en OCaml, par exemple.

Avec cette initialisation explicite, la précision de la taille dans les crochets n'est plus nécessaire lorsque l'on a renseigné tous les éléments.

Autrement, le reste du tableau s'il existe est initialisé à des valeurs nulles, et si la taille allouée n'est pas suffisante un avertissement est déclenché, et le reste de l'initialisateur n'est vraisemblablement pas utilisé, mais on n'est quoi qu'il en soit jamais à l'abri d'un *UB*.

Pour préciser sur l'usage de tableaux comme arguments de fonctions, il faut savoir que c'est l'adresse du tableau qui est à considérer comme le véritable argument, donc là aussi la notion de pointeur intervient.

Une conséquence est que toute modification d'un tableau dans une fonction a un effet global en dehors, comme on peut le voir en OCaml, où le passage par valeurs des arguments considère ces valeurs en tant que pointeurs.

Chaînes de caractères

En plus des tableaux d'entiers, nous verrons les chaînes de caractères, qui sont en pratique des tableaux de caractères. Une particularité des chaînes de caractères en C est qu'elles se terminent toujours par le caractère `'\0'` qui n'est pas comptabilisé dans la taille de la chaîne qui s'obtient par la fonction `strlen` disponible dans `string.h`.

Attention : la complexité de cette fonction est linéaire, car elle correspond à une recherche du caractère final.

La comparaison de deux chaînes de caractères doit se faire par la fonction `strcmp`, qui détermine si la première est inférieure à la deuxième (résultat entier `< 0`), égale (résultat `0`), ou supérieure (résultat entier `> 0`), dans l'ordre lexicographique.

Une comparaison avec les symboles de comparaison usuels serait sur les pointeurs associés aux chaînes, pour un résultat moins fiable.

L'entier correspondant à une chaîne pouvant être interprété en tant que tel s'obtient par la fonction `atoi`, sachant qu'il existe aussi une réciproque `itoa` d'utilisation moins simple, présentée dans le TP 5.

Les fonctions `strcpy` et `strcat`, mentionnées explicitement dans le programme, feront l'objet d'une démonstration dans ce même TP.

Types construits

À la manière des types enregistrement en OCaml, le langage C permet de créer des *structures* munies d'attributs, appelés *champs*. Ceci rend pour la suite du programme le type ainsi défini disponible.

On notera que la taille en mémoire dépend des types des champs, renseigné lors de la création de la structure. Tout est cohérent avec l'esprit du langage !

La syntaxe est la suivante :

```
struct nouvtype { type1 champ1; type2 champ2; ... };
```

Cela ne se voit pas en raison des points de suspension, mais le dernier champ doit être suivi d'un point-virgule (en OCaml ce n'est pas nécessaire).

Cette déclaration de structure ne peut pas être faite plusieurs fois dans un même fichier ni dans les en-têtes que l'on utilise (voir ultérieurement la section sur les modules).

Les champs d'une structure sont mutables sans avoir besoin de le spécifier comme en OCaml. On y accède de la même manière par ailleurs : le nom de la structure, un point, le nom du champ, ceci à gauche d'un signe égal pour la mutation.

Cependant, pour l'initialisation, la syntaxe est un peu délicate et nécessitera un exemple. On écrit `struct nomtype nomobjet = { .champ1 = valeur1, .champ2 = valeur2 };`, en notant bien la différence : une virgule au lieu d'un point-virgule, et le point présent devant le nom de chaque champ.

Par exemple, on créera et utilisera une structure de nombre complexe ainsi :

```
struct complexe { double re; double im; };

struct complexe z = { .re = 2. , .im = -2 };
double module_carre_z = z.re * z.re + z.im * z.im;
```

Attention, **une structure n'est pas un pointeur** dans le sens où, en tant qu'argument d'une fonction, c'est une copie qui est traitée. Ainsi, pour muter une structure dans une fonction, on utilisera un pointeur vers la structure (`struct nouvtype *p = &x;`). Une mutation sera par exemple `(*p).a = (*p).a * 2;`, avec des parenthèses nécessaires en raison de la priorité des opérations. *En pratique,* `*x.a` *aurait du sens si le champ* `a` *d'une structure* `x` *était un pointeur.*

Dans la mesure où cette syntaxe est lourde, une version raccourcie existe avec l'opérateur flèche : `p->a` est équivalent à `(*p).a`, mais `ceci est réservé aux pointeurs vers une structure`.

Pour éviter d'avoir à écrire le mot-clé `struct` devant le type, on peut aliaser ce type en écrivant `typedef struct nouvtype autrenom;`. Il est aussi possible de créer ici la structure afin de ne jamais avoir à lui donner de nom intermédiaire avec `struct`, mais ce n'est pas au programme.

Il est possible que `nouvtype` et `autrenom` soient identiques, mais on peut trouver plus propre. Une idée est de capitaliser la première lettre de `nouvtype` et de laisser `autrenom` en minuscule.

En tout cas la création du type doit précéder l'aliasage.

Pour compléter, au moment de l'initialisation, il n'est pas nécessaire de renseigner tous les champs, à la manière de ce qu'on peut voir avec l'initialisation des tableaux, mais il ne faudra pas accéder à un champ non initialisé, ce serait un *UB* comme dans le cas des tableaux.

Quelques remarques au sujet des champs d'une structure : tout d'abord le type peut être quelconque, y compris une autre structure, mais le besoin d'avoir une taille connue empêche d'avoir un champ dont le type est celui de la structure actuellement définie.

Pour compenser ce manque qui empêcherait de créer des structures récursives, un champ peut être un **pointeur vers une structure du même type**, car la taille est alors fixée : c'est la taille commune de tous les pointeurs.

Pour la même raison du besoin de connaître la taille, si un champ d'une structure est un tableau, sa taille doit être une constante du système, même une variable déclarée comme constante est en pratique rejetée.

Entrées et sorties

Toutes les fonctions mentionnées ici, y compris dans la partie sur les fichiers, sont dans l'en-tête `stdio.h`.

Les exemples précédents et les premiers TP ont déjà permis une familiarisation avec la fonction d'impression `printf`, permettant une impression formatée de variables.

La liste des commandes associées au symbole `%` n'est pas à connaître, mais il est bon de connaître les plus standards :

- `%d` pour un entier signé ;
- `%f` pour un flottant (sur 32 ou 64 bits) ;
- `%c` pour un caractère ;
- `%s` pour une chaîne de caractères ;
- `%p` pour un pointeur ;
- `%%` pour le symbole `%` lui-même ;
- ... mais rien n'est prévu pour les booléens !

Sans variables, la fonction `printf` est également utilisable, et on signalera l'existence de `puts` pour une chaîne de caractères quelconque (donnée explicitement ou dans une variable) et `putchar` pour un caractère quelconque (idem). Ainsi, par exemple, `puts(str)` est équivalent à `printf("%s", str)`.

Pour lire sur l'entrée standard, c'est-à-dire une saisie de l'utilisateur, la fonction `scanf` permet d'affecter à une ou plusieurs variable(s) dont un pointeur est passé en argument une valeur en cohérence avec une reconnaissance de motif effectuée en analysant la ligne qui a été entrée.

L'exécution est en particulier mise en pause le temps que l'utilisateur écrive dans la console et valide par la touche « entrée ».

Ainsi, après avoir déclaré deux variables entières `x` et `y`, initialisées ou non, écrire `printf("Entrer le score : "); scanf("%d à %d", &x, &y);` met à jour les valeurs de `x` et de `y` en fonction des entiers mis à gauche et à droite d'un `" à "`, sans présumer du comportement si la saisie de l'utilisateur est mal formatée (typiquement, négliger les espaces ne semble pas problématique, mais des changements plus importants mettront a priori n'importe quoi dans `x` et `y`).

Les esperluettes ne sont par exemple pas nécessaires avec des chaînes de caractères, qui sont déjà des pointeurs.

En plus de l'entrée et de la sortie standards, il est possible d'interagir avec des fichiers (on considère que `stdin`, `stdout` et `stderr` sont des fichiers).

La fonction d'ouverture d'un fichier, en précisant le mode, est `fopen`. Son premier argument est le chemin vers le fichier à ouvrir, son deuxième est le mode (`"r"`, `"w"` ou `"a"`, suivant qu'on ouvre en mode lecture, écriture ou ajout).

Comme pour les autres langages, le mode écriture crée le fichier s'il n'existe pas, et l'écrase sinon. De même pour le mode ajout.

Pour le mode lecture, la non-existence d'un fichier à lire, voire le manque de permissions dans les trois modes peut causer une erreur de segmentation.

Cette fonction retourne un pointeur de fichier, déclaré avec `FILE *nom`, où `nom` servira pour l'interaction.

Pour écrire dans un fichier ouvert en mode écriture ou ajout, on utilise la fonction `fprintf`, qui a la même syntaxe que `printf`, mais avec un argument supplémentaire, donné en premier, qui est précisément la variable retournée par `fopen`.

Pour lire dans un fichier, de manière analogue, on utilise `fscanf` avec le premier argument en plus indiquant le fichier.

Une fois l'écriture terminée, on ferme le fichier avec `fclose` dont l'argument est là encore la variable associée au fichier. Ceci peut être nécessaire pour que les changements soient pris en compte, et en règle générale, même quand on se contente de lire le fichier, c'est une bonne pratique.

Modules

Comme dans le cas des autres langages classiques, un projet conséquent écrit en C se présentera plutôt sous la forme d'un ensemble de fichiers, chacun rassemblant des fonctions et/ou variables ayant un rôle particulier, dans le but d'améliorer la lisibilité ou de pouvoir être utilisés dans le contexte d'autres projets, en tant que boîte à outils plus générale.

Tout ce qui est créé dans un des fichiers peut être utilisé dans un autre, à condition de le déclarer dans ce dernier, par exemple en donnant le prototype de la fonction, mais de manière bien plus pratique on utilisera des fichiers d'en-tête pour ne pas handicaper la lecture.

Il s'agit de créer, en parallèle à un fichier d'extension `.c`, un fichier de même nom (toujours pour le côté pratique) mais d'extension `.h`, à la manière des modules pré-existants.

Dans ce cas, la directive est similaire : on écrit `#include "fichier.h"`, avec les guillemets en guise de délimiteurs, car il s'agit ici d'un chemin depuis le répertoire où se situe le fichier `.c` où cette directive est écrite.

En particulier, dans cet exemple, c'est le répertoire courant qui est inspecté.

Les types construits ne peuvent cependant pas être redéfinis, donc ils ne devraient figurer que dans le fichier d'en-tête, et ce fichier serait alors déjà à inclure dans le fichier source associé lui-même.

Un risque existe également que le même fichier d'en-tête d'un fichier numéro un soit utilisé dans un fichier numéro deux, lui-même nécessaire en parallèle du fichier numéro un dans un fichier numéro trois.

Pour ne pas créer de conflit, nous allons utiliser une astuce qui fait intervenir les mots-clés `define` et `ifndef`.

Il s'agit ici du seul usage autorisé par le programme de `define` qui permet de créer des macros et qui se retrouve dans la pratique pour créer des constantes (mais par une association « syntaxique »).

Considérons un fichier d'en-tête dont on veut être certain de ne l'inclure qu'une fois (a priori, c'est le cas de tous les fichiers). L'esprit de la modularité est d'y mettre du code associé à une même idée, qu'on peut résumer à un mot-clé et qui est a priori aussi le nom du fichier, par exemple « moteur », « graphique », « data »....

Prendre un mot-clé similaire, voire identique, et l'associer à une macro permet de vérifier si la macro est déjà présente, ce qui donne ce code :

```
#ifndef GRAPH
#define GRAPH
#include "graphique.h"
#endif
```

La première ligne vérifie si la macro `GRAPH` n'existe pas encore (vérifier le contraire est également possible en utilisant `ifdef`). Si elle existe, on passe après le `endif` correspondant, sinon on exécute ce qui figure entre les deux lignes, c'est-à-dire commencer par définir `GRAPH` pour éviter de réexécuter le code de la ligne suivante, qui est ce qu'on cherchait à ne faire qu'une fois : inclure le fichier associé au mot-clé `GRAPH`.

Tout ceci est rendu nécessaire par l'impossibilité d'utiliser un même nom pour deux fonctions différentes, même avec un nombre d'arguments qui change. Contrairement à OCaml (et Python) où la fonction précédente est écrasée, C déclenche une erreur. Les types construits sont également sujets à cette impossibilité.

Compléments

`void*`

Il n'a pas échappé aux personnes les plus curieuses que le prototype de la fonction `malloc` comprend la mention mystérieuse de `void*`.

Alors même qu'il était signalé que `void` n'est pas un type, il est possible d'utiliser `void*` en tant que pointeur sans précision de type.

Ceci permet d'avoir la même notion de polymorphisme que ce que l'on rencontre en OCaml. Ainsi, un pointeur de type `void*` ne sera pas associé à un type en particulier. Dans ce cas, il est crucial de savoir la taille mémoire à traiter avec ce pointeur, quitte à ne pas dire ce que ce qui est écrit dans la zone en question est censé représenter.

Dans le cadre de la fonction `malloc`, on précise effectivement combien de mémoire est à allouer, sans que l'allocation n'ait à tenir compte du nombre de cases effectives de ce qui risque fort de s'apparenter à un tableau.

L'enseignement en CPGE ne donnera vraisemblablement pas l'occasion de manipuler des pointeurs de ce genre, quoi qu'il en soit.

Un bug classique

Considérons le code ci-après :

```
for (int i = 0 ; i < 10 ; i+= 1);
{
  // Ici un code quelconque
}
```

Il risque fort de ne pas se comporter comme on l'imaginait. La raison est que le contenu de la boucle est la première instruction rencontrée, en l'occurrence ici le point-virgule isolé, donnant lieu à une instruction vide comme `{}` le serait également.

Les accolades écrites par la suite constituent une instruction composée qui a le droit de figurer sans être introduite par une boucle ou un test.

Entre ces accolades, la variable `i` n'est pas définie, ce qui peut donner lieu à une erreur à la compilation si elle est mentionnée tout de même.

Retourner un couple

Il n'est pas prévu qu'une fonction retourne plus d'une chose en C, contrairement aux autres langages classiques.

Si la fonction doit créer et retourner deux valeurs, il y a plusieurs possibilités : soit elles sont du même type et on peut créer un pointeur représentant un tableau de taille deux (en leur réservant de la mémoire avec `malloc`) et le renvoyer, soit ce n'est pas le cas, ce qui rend la méthode précédente impossible, et il faut alors qu'au moins une des choses à retourner soit créée dans le contexte qui appelle la fonction, en passant un pointeur vers cette chose en argument.

La fonction mutera alors la valeur à partir du pointeur et pourra renvoyer l'autre valeur (ou alors muter les deux).

À ce stade, on peut se poser la question s'il ne serait pas plus simple de ne faire que des fonctions qui mutent des pointeurs et ne renvoient rien. Ce n'est pas aberrant en soi, mais c'est peu compatible avec l'esprit d'autres langages et tend à faire oublier que le passage reste par valeurs en C, il ne faudrait pas faire artificiellement du passage par référence uniquement.

Chapitre D)

Le langage SQL

Ici figure simplement une requête générique présentant l'essentiel de la syntaxe attendue au programme.

Lorsqu'il est écrit `<XXX>` ou similaire, il faut remplacer par le nom d'un objet de type `<XXX>`, que l'on peut entourer d'accents graves (recommandés en pratique, mais sur une copie on les laisse systématiquement de côté). Lorsqu'il est écrit `<nombre>`, il faut remplacer par une valeur numérique.

Dans tous les cas, les symboles $<$ et $>$ sont des délimiteurs pour l'explication ci-après et n'ont pas à figurer dans la vraie requête.

Des commentaires délimités par `/*` et `*/` apportent des précisions à certains endroits.

Les parties de la requête entourées de crochets sont optionnelles, avec l'imbrication de crochets signifiant que la partie ainsi délimitée nécessite que les niveaux d'imbrication plus superficiels soient présents mais n'a pas besoin de figurer quand ils sont présents.

L'agencement des sauts de ligne est personnel et ne fait pas l'objet d'une convention. Il est possible de choisir une autre disposition, et dans tous les cas mieux vaut ne pas écrire une requête sans agencement. La largeur de la page ne permet pas d'ajouter d'indentation ici, mais c'est tout à fait imaginable.

```
SELECT <attribut> [[AS ]<alias>] [, <attribut> [[AS ]<alias>]
[, etc.]]
  /* ou * pour tous les attributs */
  /* ou une fonction d'agrégation MAX(attribut), MIN(attribut),
  SUM(attribut numérique), AVG(attribut numérique), COUNT(*),
  COUNT(DISTINCT attribut) */
  /* Attention à ne pas prendre des attributs interdits, par exemple avec *,
  si on a utilisé GROUP BY */
FROM <table>
  /* considéré comme obligatoire, on ne fera pas SELECT 2 + 2; en prépa... */
[[LEFT ]JOIN <table2> ON [<table>.]<attribut> = [<table2>.]<attribut>]
  /* préfixage obligatoire pour tous les attributs ambigus,
  c'est-à-dire apparaissant au moins deux fois dans la table finale */
[[LEFT ]JOIN <table3> ON [<table[2]>.]<attribut> = [<table3>.]<attribut>]
WHERE <condition> [AND <condition> [etc.]] /* ou OR, avec éventuellement NOT */
[GROUP BY <attribut>[, <attribut> [, etc.]]
  HAVING <condition> [AND <condition> [etc.]] /* idem */]
[ORDER BY <attribut> [DESC] [, <attribut> [DESC] [, etc.]]]
[LIMIT <nombre> [OFFSET <nombre>]]
[UNION <toute une requête similaire> [UNION etc.]] /* ou INTERSECT ou EXCEPT */
```

On utilisera `<attribut> <comparaison> <valeur>` la plupart du temps pour les conditions, ou parfois `<attribut> <comparaison> <attribut>`, avec d'éventuelles opérations arithmétiques. Les comparaisons utilisent les symboles usuels des langages de programmation, avec = pour le test d'égalité. Si les valeurs impliquent des chaînes de caractères, il faut les entourer d'apostrophes ou de guillemets. Les valeurs numériques peuvent être entourées de ces délimiteurs, c'est à éviter par principe mais toléré.

Une possibilité avancée est de faire par exemple `<attribut> = <requête>`, si la sous-requête ne renvoie qu'un enregistrement avec un attribut, qui sera alors comparé.

Encore plus technique : `<attribut>[, <attribut>[, etc.]] IN <requête>`, si la sous-requête renvoie une liste d'enregistrements de schéma compatible avec ce qui est comparé. Cette liste d'enregistrements est alors considérée comme une table virtuelle, et MySQL, qui appelle cela *derived table* en anglais, oblige une telle table à avoir un alias (comme on le remarquera en TP). Sur une copie, l'oubli d'un tel alias ne serait pas pénalisable car on évalue le SQL standard.

Deuxième partie

Cours

Chapitre 1

Algorithmes et programmes

Partie du cours : Méthodes de programmation
Prérequis : Binaire et connaissances de base
Applications : Tout le reste de la vie !
Compétence visée principale : Justifier et critiquer une solution

---

1.1 Introduction

Commençons par un constat. Pour un public non averti, qui peut comprendre des professeurs de matières connexes à l'informatique, la distinction entre algorithmique et programmation n'est pas forcément connue de manière précise, de même qu'entre un algorithme et un programme.

Nous définirons ici un algorithme comme une suite de calculs organisée autour de structures de contrôle que l'on peut faire faire par un humain. Un programme est la réalisation d'un algorithme dans un langage informatique informatique particulier, avec une zone grise en ce qui concerne le pseudo-code.

Il est tout à fait possible que quelqu'un puisse maîtriser de nombreux algorithmes sans jamais écrire de programmes, bien que ce soit rare.

Il n'est pas exclu par ailleurs que quelqu'un ait une aisance particulière en programmation mais des lacunes en algorithmique, et ceci constitue un problème en ce qui concerne l'efficacité des programmes écrits.

En ce qui concerne la programmation, une fois écrit, un programme ne peut pas être exécuté directement sur l'ordinateur.

En effet, seul le code machine est utilisable.

Il s'agit alors de transformer le programme en code machine par une action qu'on appelle *compilation*.

Un *compilateur* est un programme, rédigé dans un certain langage L1, permettant d'analyser qu'un *code source* écrit dans un langage L2 en respecte bien la syntaxe (et les règles de typage pour OCaml, par ailleurs), et qui produit un programme dans un langage L3, a priori un exécutable en code machine mais potentiellement un fichier source d'un langage intermédiaire si possible plus proche du code machine que L2.

Un *interpréteur* exécute à la volée le code machine obtenu par la compilation sans que l'utilisateur n'ait à accéder à l'exécutable.

Python est plutôt un langage interprété, alors que C et OCaml sont plutôt des langages compilés. Les compilateurs classiques pour le langage OCaml sont `ocamlc` (produit du code binaire, nécessitant OCaml pour fonctionner) et `ocamlopt` (produit du code natif, ne nécessitant pas OCaml pour fonctionner mais pouvant dépendre du système d'exploitation) et celui pour le langage C est `gcc` sous Linux.

Un fichier source dans le langage OCaml est censé avoir l'extension `.ml`. On appelle ceci un *fichier d'implémentation*. Il est associé à un *fichier d'interface* d'extension `.mli` où sont reprises les signatures des fonctions, types des variables, etc. du fichier d'implémentation. Pour un module, il peut être pertinent de consulter simplement le fichier d'extension quand on cherche à se renseigner sur une fonction particulière et que le code n'est pas nécessaire (par exemple si on a oublié l'ordre des arguments de `List.mem`).

En programmation, différents styles (on dit aussi *paradigmes*) existent, dont certains sont intimement associés à l'esprit du langage. On a souligné à l'occasion de la présentation des langages au programme la notion de *programmation fonctionnelle*, caractéristique de langages comme OCaml et, dans une plus forte mesure, Haskell, associée à un style récursif, et celle de *programmation impérative*, où les instructions sont omniprésentes, comme dans la plupart des langages dont C et Python, associée à un style itératif.

La *programmation logique* est également un style associé à des langages tels que Prolog, et son utilité se retrouve dans la théorie cachée derrière les bases de données vues en fin d'année (sans que Prolog ne soit enseigné en CPGE, hélas).

Conjointement à l'algorithmique, d'autres paradigmes plus transversaux seront présentés en MP2I, dont notamment le *diviser pour régner* et la *programmation dynamique*, qui peuvent se retrouver dans des programmes itératifs comme récursifs.

1.2 Représentation des flottants

Remarque : Cette section présuppose une maîtrise du binaire. Un rappel peut être donné en classe mais ne figure pas dans ces notes de cours.

Le nom donné en informatique à la représentation des nombres non entiers est la *virgule flottante* (le type correspondant est "float", appelés « flottants » en français).

Rappelons qu'aucun nombre irrationnel ne peut être représenté de manière exacte avec des caractères de n'importe quelle base. D'ailleurs, tous les nombres représentables de manière exacte en base 2 sont en particulier décimaux, tandis que dans l'autre sens, le nombre $0,2$ par exemple, aussi innocent soit-il, a un développement infini en base 2, à savoir $\overline{0,001100110011\ldots}^2$.

Des conséquences de l'absence d'écriture exacte finie de certains nombres décimaux en base 2 sont présentées dans le TD 2.

Bien plus, même un nombre qui s'exprime de manière exacte en binaire n'est pas forcément représentable de manière informatique, puisque la mémoire est finie. Tout comme on peut représenter 2^{64} entiers sur 64 bits, cette restriction demeure évidemment pour les flottants.

Ainsi, on ne peut pas aller vers l'infiniment proche de zéro, et le plus grand nombre représentable sur 64 bits est limité à quelques centaines de chiffres, ce qui est néanmoins suffisant pour modéliser les grandeurs physiques.

La troisième limitation est la précision : entre deux nombres représentables, il y a un écart correspondant environ à un milliardième de milliardième de la valeur de ces nombres.

La représentation des nombres réels, qui est habituellement approximative, se fonde sur l'écriture dite scientifique en binaire, normalement connue pour les nombres décimaux Rappelons ici si besoin que l'écriture scientifique revient à écrire un nombre non nul, éventuellement en tant qu'arrondi, sous la forme $x \times 10^k$, où $k \in \mathbb{Z}$ et $1 \leq x < 10$, et l'écriture est unique.

Grâce à la représentation à virgule flottante, les nombres très grands ou très petits peuvent s'écrire sans une surcharge de caractères peu représentatifs au niveau de la virgule. De toute façon, comme annoncé ci-avant, les limites de la mémoire imposent de procéder rapidement à un arrondi.

Selon la norme IEEE754, avec 64 bits, un nombre en virgule flottante est alors un produit $(-1)^s \times m \times 2^n$, où s est bit de signe, m est la *mantisse* $\overline{1, b_1 b_2 \ldots b_{52}}^2$ (puisque le 1 est systématique, il n'entre pas dans la représentation qui est donc sur 52 bits) et n est un entier relatif entre -1022 et 1023 écrit sur 11 bits et représenté comme $n + 1023$. Avec 32 bits, la taille de la mantisse passe à 23 bits et l'exposant est sur 8 bits.

La disposition des bits est la suivante : d'abord s, puis n, puis m.

On notera que pour comparer deux nombres écrits en virgule flottante, on regarde d'abord le signe, puis à même signe on regarde l'exposant, puis à même exposant on regarde la mantisse.

Les valeurs non utilisées pour les exposants correspondent aux mots de 11 bits 00000000000 et 11111111111. On les réserve pour des valeurs exceptionnelles, qui ne sont pas à connaître. On ne citera pour la culture que le zéro, qui existe en version positive et en version négative (suivant le premier bit), dont les bits de l'exposant et de la mantisse sont tous nuls, et les infinis, qui s'obtiennent quand tous les bits de l'exposant sont à 1 et ceux de la mantisse à 0.

Conséquences

Il est évident que sur 64 bits, même en ne considérant aucune valeur exceptionnelle, on ne pourrait encoder que 2^{64} valeurs, et les nombres supérieurs à 2^{1024} en valeur absolue (une valeur qui reste certes déraisonnable en sciences) ainsi que ceux qui sont plus proches de zéro que 2^{-1022} n'auraient pas de représentation possibles.

En pratique, on traite les dépassements arithmétiques (et les « soupassements arithmétiques », le deuxième cas) de différentes façons, précisément à l'aide des valeurs exceptionnelles (d'où l'intérêt d'avoir un zéro positif et un zéro négatif, au passage). Le déclenchement d'erreurs à l'aide de valeurs exceptionnelles non évoquées est une façon de faire, peut-être plus efficace que l'utilisation systématique de zéros ou d'infinis seulement.

Dans l'intervalle où les réels sont représentables, la question de l'arrondi se traite de la même façon que pour les nombres usuels : si le premier bit dépassant la taille de la mantisse devrait être un 0, on arrondit le dernier bit par défaut, sinon par excès avec propagation éventuelle de retenue, à l'exception notable du cas où seul un bit à 1 dépasse la taille de la mantisse, ce qui est analogue à un arrondi d'un demi-entier à l'entier le plus proche.

Le fait d'arrondir des valeurs, quand bien même est-il nécessaire, favorise une perte d'informations lorsque des opérations sont effectuées entre des réels de valeurs très éloignées. Par conséquent, en tenant compte des priorités opératoires (et quand la priorité est la même, en effectuant les opérations de gauche à droite), deux expressions donnant dans la réalité le même résultat donneront parfois sur l'ordinateur des résultats différents.

Ce qu'il faut alors garder à l'esprit, c'est que les tests d'égalité entre réels sont très peu pertinents et qu'on leur préfèrera une comparaison de type « la différence est suffisamment petite en valeur absolue ».

1.3 Preuves de programmes

Écrire un programme, c'est bien. Écrire un programme correct, c'est mieux. Et comme on n'est jamais vraiment certain qu'un premier jet, voire une version retravaillée, ne comporte pas d'erreur, il faut se doter d'outils théoriques permettant de garantir que le programme respecte sa spécification, pour ne pas avoir à se fier aux tests, manquant parfois d'efficacité.

Pour se faire une idée, il n'est pas rare d'entendre que le budget alloué à la vérification et au débug couvre plus de la moitié du budget global pour la programmation, données pouvant être estimées à partir de la répartition du temps de travail effectif des personnes concernées.

Cette section présente les preuves de terminaison (le programme s'arrête sur toute entrée) et de correction (le programme calcule ce qu'il devait calculer) en tant qu'outils mathématiques, et le chapitre suivant fournira une méthode de présentation des programmes en parallèle de la rédaction des preuves, après quoi cet exercice, souvent redouté, devrait en être facilité, pour le plus grand bien des programmes ainsi validés.

Preuves de terminaison

Commençons par une bonne nouvelle : une boucle inconditionnelle (`for`) termine toujours en théorie. Certes, en Python, en cas de mutation de la séquence à parcourir ou si celle-ci est de taille infinie, des programmes dégénérés peuvent mettre à mal cette affirmation, et on verra également en C que la notion de boucle inconditionnelle est à prendre avec précaution, mais nous considérons ici des boucles qui comportent intrinsèquement un nombre de tour fixé à l'avance.

En pratique, nous allons nous concentrer sur les preuves de terminaison de programmes utilisant une boucle conditionnelle (`while`), et de programmes récursifs.

La théorie mathématique que l'on peut utiliser dans pour ainsi dire tous les cas est la suivante : l'ordre usuel sur $\mathbb{N}$ est bien fondé (voir chapitre 10), c'est-à-dire qu'il n'existe pas de suite infinie strictement décroissante d'entiers naturels.

Pour prouver qu'une boucle conditionnelle termine, il suffit de trouver une expression dépendant des variables du programme et qui s'évalue en un entier décroissant strictement à chaque tour dans la boucle.

Une telle expression, appelée *variant*, est souvent assez rapide à trouver.

Prenons pour exemple un programme qui parcourt une séquence par une boucle conditionnelle (par principe, mieux vaut faire une boucle inconditionnelle quand c'est possible) :

```
Fixer i à 0
Tant que i < taille(seq) faire
    Imprimer l'élément de seq à l'indice i
    Augmenter i d'un
Fin Tant que
```

On prend pour variant la taille de `seq` moins `i`. Puisque la boucle est quittée dès que `i` ne sera plus strictement inférieur à la taille de `seq`, on peut considérer que cette expression est toujours un entier naturel. Ensuite, puisque `seq` n'est jamais modifié, sa taille non plus à plus forte raison, alors que `i` est augmenté d'un à chaque tour dans la boucle. Par conséquent, le variant décroît strictement à chaque tour, ce qui garantit la terminaison du programme.

Un deuxième exemple est la fonction qui calcule la somme des chiffres d'un entier naturel représenté en base 10 :

```
Fonction somme_chf(n)
    Fixer s à 0
    Tant que n > 0 faire
        Augmenter s de n modulo 10
        Diviser n par 10 (division euclidienne)
    Fin Tant que
    Retourner s
Fin Fonction
```

Cette fois-ci, on prend simplement pour variant `n`, qui est positif d'après la première ligne et dont la décroissance est garantie puisqu'on procède à une division euclidienne par 10 d'un nombre strictement positif (par hypothèse) en fin de boucle.

Les variants peuvent dépendre de variables qui ne sont pas des entiers, comme des réels dans la preuve de terminaison de la recherche de zéro d'une fonction par dichotomie (auquel cas on passe par la partie entière d'un logarithme), ou des séquences, mais auquel cas le variant concerne souvent leur taille.

Le cas des fonctions récursives sera traité dans le chapitre idoine.

Preuves de correction

Une fois qu'on a prouvé que le programme terminait, et même dans les cas où il ne termine pas, en fait, il faut encore établir qu'il fait ce pour quoi on l'a écrit. La *preuve de correction partielle* revient à établir que le programme respecte sa spécification lorsqu'il termine, et la *preuve de correction totale* ajoute la terminaison dans tous les cas.

Tester sur plusieurs entrées, notamment des cas limites, est une solution assez efficace en pratique, mais très peu rigoureuse du point de vue scientifique.

On préfèrera écrire une preuve formelle du programme, ce qui sera relativement facile dans les cas étudiés dans ce cours, mais qui peut être très contraignant dans la vie réelle.

Pour aller plus loin, une troisième approche existe : la construction de modèles, donnant lieu à une branche de l'informatique appelée *model-checking*.

Comme pour la terminaison, un programme sans boucle est relativement aisé à prouver. Cette fois-ci, en revanche, les boucles inconditionnelles devront être étudiées, et on les traitera comme les boucles conditionnelles.

L'outil présenté ici est l'*invariant de boucle*. Il s'agit d'une propriété dont on veut prouver qu'elle est vraie en entrant dans une boucle et qui le demeure à chaque tour jusqu'à la sortie de la boucle. L'invariant devra être pertinent, de sorte qu'en le prouvant on aura prouvé que le programme est correct.

Dans le cas de boucles imbriquées, l'invariant de la boucle intérieure peut servir à prouver celui de la boucle extérieure.

Prenons encore l'exemple de la fonction `somme_chf`. L'invariant de boucle choisi est :

La variable `s` contient la somme des `i` derniers chiffres de l'argument de la fonction à la fin du `i`-ième tour.

Cet invariant sera difficile à prouver tel quel, donc nous allons l'étendre en précisant :

... et de plus la variable `n` contient $[\frac{|\mathtt{x}|}{10^{\mathtt{i}}}]$ (où `x` est l'argument de la fonction) à la fin du `i`-ième tour (... existant si `n` ne valait pas encore `0` au tour précédent, pour être totalement rigoureux).

Comme ce sera souvent le cas, il s'agit de faire une récurrence.

On suppose l'argument de la fonction non nul, pour que la boucle soit parcourue au moins une fois.

— À la fin du `0`^{e} tour, c'est-à-dire avant d'entrée dans la boucle, la variable `s` contient bien `0` et la variable `n` contient bien le nombre en entrée.
— Supposons l'invariant vrai à la fin du `k`-ième tour dans la boucle. Alors à la fin du tour suivant on divise `n` par `10`, ce qui fait que la deuxième partie de l'invariant est héréditaire, et de plus on a entre temps ajouté à `s` le chiffre des unités de `n`, qui est le `(k+1)`-ième chiffre de l'argument en partant de la fin, d'après la deuxième partie de l'invariant et la définition du `i`-ième chiffre d'un nombre écrit dans n'importe quelle base. Par conséquent, la variable `s` contient bien la somme des `k+1` derniers chiffres de l'argument de la fonction à la fin du `(k+1)`-ième tour.
— En sortie de la boucle, `n` vaut `0` et donc le nombre de tours est le nombre de chiffres de l'argument de la fonction, qui est un de plus que la partie entière du logarithme décimal de ce nombre. Ainsi, `s`, qui est aussi la valeur retournée, vaut bien la somme des chiffres de l'argument, une fois la boucle terminée, ce qui prouve la correction du programme.

Le cas des fonctions récursives sera là aussi traité dans le chapitre idoine.

1.4 Complexité

Introduction

Il est temps de discriminer les algorithmes en fonction de leur efficacité, c'est-à-dire leur coût en opérations élémentaires et l'espace mémoire qu'ils utilisent. Nous introduisons pour cela ici la notion de *complexité*, respectivement en temps et en espace.

En fait, dans la mesure où les ordinateurs ont une capacité de plus en plus grande, la complexité en espace est un critère moins prépondérant que la complexité en temps. Les problèmes viennent surtout du temps exponentiel, l'espace restant souvent raisonnable.

Prenons un exemple pratique avec la méthode de Horner pour déterminer l'image d'un réel par une fonction polynomiale.

En écrivant un programme qui calcule $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$, le nombre d'additions est de n et le nombre de multiplications est de $n + (n-1) + \cdots + 0$, soit $\frac{n(n+1)}{2}$.

La complexité en espace est la taille de la réponse finale, ou plus précisément la plus grande taille d'une valeur intermédiaire.

On peut améliorer un tel programme en se rendant compte que la valeur de x^i est recalculée pour tous les monômes de degré supérieur. En stockant les valeurs de tous les x^i, de sorte de ne les calculer qu'une fois par $n-1$ multiplications, on fait alors passer le nombre de multiplications à $2n-1$. Cette amélioration nette de la complexité en temps nécessite malheureusement un coût en espace supplémentaire pour le stockage des valeurs des x^i.

La méthode de Horner consiste à se rendre compte que

$$a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 = \underbrace{(\ldots((}_{n-1\ '('} a_n x + a_{n-1})x + a_{n-2})\ldots)x + a_1)x + a_0,$$

et donc à réaliser seulement n multiplications et n additions, sans utiliser d'espace mémoire supplémentaire.

L'exemple ci-dessus permet de voir que la complexité se fait en fonction des paramètres du problème (ici n, mais aussi les coefficients du polynôme qui détermineront la taille des nombres stockés, qui influe sur le temps demandé pour faire les opérations arithmétiques), et elle soulève la question de l'unité exacte employée : on ne peut pas parler de temps alors même que les performances de l'ordinateur peuvent causer des écarts de temps bien plus importants entre deux programmes.

La première définition mentionnait les opérations élémentaires, or une multiplication est toujours plus coûteuse qu'une addition pour un processeur. Ainsi, il s'agit de retenir un critère pertinent et adapté au problème.

Pour de nombreux algorithmes, notamment ceux qui sont vus en option et les tris, on distingue la complexité dans le pire des cas et la complexité dans le meilleur des cas. Nous parlerons ultérieurement de la complexité en moyenne et du coût amorti.

Cette nuance s'illustre par un programme naïf déterminant si un entier n est premier en testant la divisibilité par tous les entiers entre 2 et $\sqrt{|n|}$, avec une réponse immédiate pour des nombres proches de 0 (discutable).

Dans le meilleur des cas, notre nombre est pair (et $\neq \pm 2$). Alors la réponse est *faux* dès le premier test de divisibilité.

Dans le pire des cas, notre nombre est premier, ce qui nécessite $\sqrt{|n|} - 1$ tests de divisibilité (critère retenu).

On dira alors que la complexité du programme est dans le meilleur des cas un test de divisibilité et dans le pire des cas $\sqrt{|n|} - 1$ tests de divisibilité, ce qui correspond conventionnellement à un programme fonctionnant en temps exponentiel, car le calcul de complexité considère la taille de n, donc le nombre de bits utilisés dans sa représentation, qui est $\lceil \log_2(n+1) \rceil$.

Remarque : Certains informaticiens considèrent que les nombres sont représentés en « unaire », donc par autant de symboles `0` que leur valeur, ce qui donne dans le cas présent la complexité attendue en racine de n.

Calcul pratique

Le calcul de la complexité en temps d'un algorithme se fait par une analyse de sa structure, ainsi :

- Une instruction de base, par exemple un calcul arithmétique ou logique, une affectation, etc. aura un coût négligé ou retenu comme une unité (comme dit précédemment).
- Le coût de plusieurs instructions qui se suivent, notamment au sein d'un bloc, est la somme des coûts des instructions. Cela vaut bien entendu si les instructions qui se suivent sont eux-mêmes des blocs.
- Le coût d'un test conditionnel est au plus le maximum des coûts des blocs qui le composent, auquel on ajoute au plus le coût d'évaluation de tous les booléens dans la portée du `if` et des `elif` éventuels.
- Le coût d'une boucle inconditionnelle est la somme des coûts des instructions du bloc pour chaque tour dans la boucle. Dans les nombreux cas où les coûts sont tous identiques, on peut procéder à une multiplication du coût commun par le nombre de tours.
- Le coût d'une boucle conditionnelle est également la somme des coûts des instructions du bloc pour chaque tour dans la boucle, plus le coût de chaque vérification de la condition. Malheureusement, cette fois-ci, le nombre de tours dans la boucle n'est pas forcément connu. En règle générale, on cherche juste une bonne majoration, ceci étant.

Lorsque des boucles sont imbriquées, il s'agit de faire une somme double, qui se limite dans les meilleurs cas à une double multiplication.

Par exemple, le programme suivant a un coût de $mn+1$ affectations, mn impressions et $2mn$ multiplications :

```
Fixer s à 1
Pour i de 0 à n-1 faire
  Pour j de 0 à m-1 faire
    Imprimer 42 * s
    Multiplier s par -1
  Fin Pour
Fin Pour
```

Notations de Landau

Dans le programme précédent, la rigueur imposait d'ajouter 1 au coût en raison de la première affectation, mais il est évident que ceci est négligeable quand m et n sont assez grands.

En pratique, la complexité ne se donne presque jamais exactement mais toujours de manière asymptotique, c'est-à-dire en négligeant tout ce qui peut être négligé afin de donner un ordre de grandeur concis.

Pour ce faire, on utilise les *notations de Landau* (dont les plus utiles seront vues en mathématiques dans un cadre plus général au deuxième semestre) : soient f, g des fonctions, de $\mathbb{N}$ dans $\mathbb{N}$ pour nous. On dit que :

- $f = o_{+\infty}(g)$ si la limite en $+\infty$ de $\frac{f}{g}$ est 0.
- $f = \mathcal{O}_{+\infty}(g)$ si $\frac{f}{g}$ est bornée au voisinage de $+\infty$.
- $f = \Omega_{+\infty}(g)$ si $g = \mathcal{O}_{+\infty}(f)$ (cette notation est utilisée plus rarement).
- $f = \Theta_{+\infty}(g)$ si $f = \mathcal{O}_{+\infty}(g)$ et $g = \mathcal{O}_{+\infty}(f)$.
- $f \sim_{+\infty} g$ si la limite en $+\infty$ de $\frac{f}{g}$ est 1, donc si $g - f = o_{+\infty}(g)$.

On s'autorise à écrire $f = \mathcal{O}(g)$ pour aller plus vite : on ne signale pas comme en mathématiques que la comparaison se fait en l'infini.

Ainsi, on dira que la complexité en temps de notre programme est un / en $\mathcal{O}(mn)$. L'inconvénient de notre notation est que pour nous, $10^9 n = \mathcal{O}(n)$, ce qui n'est pas pertinent au niveau d'un ordinateur. Toutefois, de telles extrémités n'interviennent pas si fréquemment que cela.

Les algorithmes étudiés ici auront généralement une complexité en temps en $\Theta(1)$ (temps constant), en $\Theta(\log n)$ (temps logarithmique), en $\Theta(n)$ (temps linéaire), en $\Theta(n^2)$ (temps quadratique), en $\Theta(n^3)$ (temps cubique) ou en $\Theta(a^n)$ (temps exponentiel, pour un $a > 1$). Les algorithmes de tri efficaces ont une complexité en $\Theta(n \log n)$, que certains auteurs qualifient de « linéarithmique ».

Remarque : Dans le cas du temps logarithmique, la base du logarithme n'importe pas, vu que le rapport entre deux logarithmes se retrouvera dans la constante multiplicative.

Signalons pour finir qu'en théorie de la complexité, l'imprécision va encore plus loin : on peut se contenter de dire qu'un algorithme est en temps polynomial, sans déterminer a priori pour quel $k \in \mathbb{N} \setminus \{0\}$ la complexité en temps est en $\mathcal{O}(n^k)$ mais pas en $\mathcal{O}(n^{k-1})$. Dans d'autres cas, on peut être amené à chercher le meilleur k possible, sans qu'il ne soit entier. La notion de temps polynomial sera exposée en deuxième année.

Complexité en espace

Quelques rapides informations sur la *complexité en espace* : elle se calcule à peu près comme la complexité en temps, mais il faut tenir compte de la réutilisation ou non de la mémoire au fur et à mesure de l'exécution.

En quelque sorte, la complexité en espace est la taille maximale de la mémoire utilisée à tout moment de l'exécution.

Les deux programmes suivants, tous deux de complexité linéaire en temps en nombre d'instructions élémentaires, et retournant la même chose, auront ainsi des complexités en espace respectives de $\mathcal{O}(\log n)$ et $\mathcal{O}(n \log n)$ bits, en raison de la taille nécessaire pour écrire les entiers en binaire :

```
Fonction somme1(n)
    Fixer s à 0
    Pour i de 0 à n-1 faire
        Ajouter i à s
    Fin Pour
    Retourner s
Fin Fonction
```

```
Fonction somme2(n)
    Fixer s à 0
    Fixer l à la liste vide
    Pour i de 0 à n-1 faire
        Insérer i dans l
    Fin Pour
    Retourner la somme des éléments de l
    # Ne pas rire, j'en ai vu beaucoup depuis que j'enseigne
Fin Fonction
```

Attention : La complexité d'un programme n'est pas la complexité du problème auquel il a répondu, qui correspond à la complexité du meilleur programme imaginable, exprimée asymptotiquement en pratique.

Complexité en moyenne et coût amorti

Dans certains cas, la mesure de la complexité peut dépendre de plusieurs utilisations d'une fonction (par exemple), où on peut s'attendre à rencontrer très peu souvent les pires cas, qui sont alors noyés parmi d'autres cas au point de permettre d'observer une complexité moindre « au chronomètre » que ce que le calcul pessimiste aurait fait calculer.

Les deux notions introduites ici peuvent facilement être confondues, il faudra faire particulièrement attention à leur différence.

Complexité en moyenne

La *complexité en moyenne* suppose que le programme nécessitant le calcul est amené à servir pour plusieurs entrées différentes. Il faut alors une estimation précise de la complexité en fonction de chaque entrée possible. On définit alors la complexité moyenne comme la moyenne des complexités (éventuellement pondérées par la probabilité d'apparition de chaque entrée, sachant qu'on estimera par principe une équiprobabilité, et que la pondération servirait essentiellement à rassembler des entrées donnant une même complexité).

Le bagage mathématique qui intervient peut nécessiter de délayer le calcul d'une complexité moyenne à un moment ultérieur de l'année. Quoi qu'il en soit, le nombre d'occasions où un tel calcul est demandé sera limité en CPGE, y compris aux concours.

Un exemple simple peut être de déterminer la position du premier *vrai* dans une liste de booléens par un algorithme faisant un parcours intuitif.

L'ensemble des entrées possibles de taille fixée n est composé des 2^n listes de booléens possibles.

La complexité d'un appel à la fonction (calculée en nombre de comparaisons) sera considérée comme l'indice du premier *vrai*, c'est-à-dire la réponse elle-même.

Ainsi, la moitié des listes commence par un *vrai*, donnant une complexité d'une opération élémentaire.

Ensuite, la moitié des listes restantes a un *vrai* en deuxième position, donnant une complexité de deux opérations élémentaires.

On en déduit la formule de la complexité moyenne par une récurrence simple : $\text{cmoy}_n = \sum_{k=1}^{n} \frac{k}{2^k}$ (le tout plus $\frac{n}{2^n}$ pour le cas où la liste n'a pas de *vrai*), ce qui se résout mathématiquement en $\text{cmoy}_n = 2 - \frac{1}{2^{n-1}}$, et le calcul de limite permet de dire que la complexité moyenne est de 2 pour de grandes listes (sachant que de toute manière la soustraction est à ignorer en général).

Coût amorti

Concernant le *coût amorti*, il ne concerne a priori pas une fonction « principale » agissant par exemple sur l'entrée fournie par l'utilisateur, mais plutôt une fonction auxiliaire dont le calcul global nécessite plusieurs appels.

On calcule alors là aussi la moyenne des complexités des appels, en considérant que ces valeurs peuvent différer grandement dans certains cas.

Par exemple, si une fonction construit sa réponse par différentes opérations dont la complexité dépend de la taille de l'argument, le fait que la réponse puisse croître va jouer sur le coût successif des opérations, mais souvent de manière linéaire, de sorte qu'on observe en moyenne un temps de même tendance que le temps pour les derniers calculs.

Un cas bien plus édifiant est la construction d'une liste en Python par des appels successifs à la méthode `append`.

Il a sans doute été signalé que le coût de cette méthode était constant, en fait c'est vrai à condition de parler du coût amorti.

En l'occurrence, la création d'une liste nécessite l'allocation d'une zone mémoire suffisante et même plus, afin de laisser de la place aux éléments ajoutés par la suite.

Mais au bout d'un certain nombre d'ajouts, la zone mémoire sature et il faut réserver plus de place, malheureusement cette place en plus n'est pas forcément disponible juste à côté, et l'implémentation nécessitant d'avoir une zone en un bloc, il faut tout recopier dans une nouvelle zone plus grande (le facteur d'agrandissement est légèrement supérieur à un, nous allons faire le calcul comme si ce facteur était de deux pour faciliter les choses).

Ainsi, l'ajout d'un élément coute au choix une opération élémentaire quand il y a la place ou autant d'opérations élémentaires que la nouvelle taille (donc l'ancienne taille plus l'ajout naturel) quand il faut faire ce recopiage.

Pour 2^n ajouts depuis une liste vide, en admettant que le recopiage intervienne quand une puissance de deux est dépassée, cela donne 2^n ajouts élémentaires et le recopiage de toutes les puissances de deux jusqu'à 2^{n-1}.

Le total est de $2^n + (2^n - 1)$, pour une moyenne arrondie à 2 opérations élémentaires pour les ajouts. Les surcoûts ont donc bien été amortis.

Chapitre 2

Discipline de programmation

Partie du cours : Méthodes de programmation
Prérequis : Connaissances de base
Applications : Tout le reste de la vie !
Compétence visée principale : Justifier et critiquer une solution

---

2.1 Spécification

Lorsqu'on écrit un programme, pour peu qu'il ne provoque pas d'erreur de syntaxe, il n'est pas faux de dire qu'il donne un résultat, même si ce résultat est une erreur d'indexation ou n'importe quel calcul inutile.

Cependant, dire que le programme fait bien ce qu'on attend de lui implique évidemment d'avoir défini à l'avance ce qu'on attend de lui. Il s'agit de la notion de *spécification*.

À notre niveau, il s'agira simplement d'écrire pour chaque fonction ce qu'elle attend comme type d'argument, ce qu'elle renvoie, ses effets de bord et toute information utile d'ordre général, à compléter par des commentaires sur des endroits particuliers.

On l'aura compris, la spécification s'écrit dans la documentation de code, qu'il est recommandé de toujours rédiger en l'absence de contraintes de temps trop restrictives.

La version 3 de Python a progressivement introduit la syntaxe des annotations, dont un exemple figure ci-après :

```
def position_maximum(l : list) -> int:
```

Ici, la ligne traditionnelle `def position_maximum(l):` a été quelque peu étoffée. Il est indiqué que la variable `l` est de type `list`, et la flèche précise le type de la valeur de retour.

Ces informations se retrouveront, comme la documentation de code, dans le texte imprimé par l'appel de la fonction `help`. Cependant, **les types ainsi annoncés ne forment pas une contrainte au niveau de l'exécution**, au contraire de la syntaxe analogue utilisable en OCaml. On appelle ceci la *signature* de la fonction.

Le *prototype d'une fonction* en C reprend également le même principe, imposant également les types quitte à caster un résultat du mauvais type (mais on se forcera toujours à écrire les fonctions de façon à ce que cette opération n'ait pas lieu d'être).

Ensuite, la spécification de la fonction. On commencera par choisir un nom de fonction pertinent, pour qu'un coup d'œil rapide aux en-têtes permette de comprendre de quoi il retourne. Il faut encore apporter des clarifications, notamment sur le contexte. Une fonction équivalente en C aura par exemple des mentions supplémentaires juste avant ou à l'intérieur en commentaire.

Précisons que les commentaires sont délimités en C par /* et */, sans possibilité d'imbrication, et en OCaml par (* et *) avec possibilité d'imbrication. La syntaxe `//` a été ajoutée ultérieurement au langage C pour produire un commentaire valable jusqu'à la fin de la ligne.

```
int position_maximum(int *t, unsigned int n)
{
  /* Détermine un indice où se situe le maximum d'un tableau d'entiers
  dont la taille est également précisée.
  Si le maximum apparaît plusieurs fois,
  seule la première occurrence est retournée. */
  // code ici
}
```

Tout ce qu'on voulait savoir est désormais annoncé, notamment la gestion des égalités qui aurait pu diverger de la version intuitive.

Le programme peut alors se lire plus facilement.

Précisons pour autant que l'écriture de la documentation peut se faire une fois celle de la fonction terminée, voire s'ébaucher en premier puis s'étoffer avec d'éventuelles corrections à la fin.

```
int position_max(int *t, unsigned int n)
{
  /* Détermine un indice où se situe le maximum d'un tableau d'entiers
  dont la taille est également précisée.
  Si le maximum apparaît plusieurs fois,
  seule la première occurrence est retournée. */
  int rep = 0;
  for (int i = 1 ; i < n ; i += 1)
  {
    if (t[i] > t[rep]) rep = i;
  }
  return rep;
}
```

2.2 Préconditions, postconditions

Restons sur l'exemple précédent. Il se lit certes plus facilement, mais peut-être n'est-il pas immédiat de se convaincre qu'il fonctionne. Nous allons utiliser des *préconditions et postconditions*, qui sont à la base de la *logique de Hoare.*[1]

Pour ce faire, des passages techniques du code bien cernés vont s'accompagner de commentaires supplémentaires avant et après chaque passage, de sorte que le commentaire avant contienne une information supposée vraie à ce moment de l'exécution, la *précondition*, et que le commentaire après contienne une information qui se déduit de la précondition ainsi que de l'exécution du passage.

1. Il s'agit d'une méthode particulièrement efficace pour prouver des programmes. Elle n'est pas à maîtriser en détail mais n'en est pas moins abordable.

L'exemple le plus simple est le suivant :

```
// ici x vaut 3
y = 2*x; // ici y vaut 6... et x continue de valoir 3
```

Appliquons ceci à la fonction de recherche de position du maximum. Notons qu'une postcondition d'un passage peut devenir une précondition d'un autre passage, et un code entièrement balisé aurait les hypothèses sur les arguments en précondition et la spécification déductible de la postcondition.

Par choix personnel, qui n'est nullement une convention, les préconditions seront introduites par une flèche vers la gauche dans un programme et les postconditions d'une flèche vers la droite. Dans l'écriture sur papier d'un programme, on utilise usuellement des accolades pour les deux.

Ces préconditions et postconditions ont tout à fait leur place sur une copie quand une preuve est demandée ou quand le programme est si obscur que le correcteur pourrait se fâcher sans explications. L'autre cas d'utilisation est la recherche de bug, parfois même utile quand le bug est une bête coquille. On comprendra que la rédaction de preuves de correction en est facilitée !

```
int position_max(int *t, unsigned int n)
{
  // <- RAS
  int rep = 0;
  // -> rep est le premier indice du maximum jusque là,
  //       en considérant qu'on a traité l'indice 0 uniquement
  for (int i = 1 ; i < n ; i += 1)
  {
    // <- rep est le premier indice du maximum jusqu'à l'indice i exclu
    if (t[i] > t[rep]) rep = i;
    // -> rep est le premier indice du maximum jusqu'à l'indice i inclus
    // (ceci fait office de précondition pour le tour suivant)
    // (on en déduit un invariant de boucle)
  }
  // -> précondition du tour où i = n, n'ayant pas eu lieu
  // (donc rep est le premier indice du maximum sur l'ensemble du tableau)
  return rep;
}
```

Au niveau du test conditionnel dans la boucle, on réfléchit à ce qu'il se passe suivant que la condition soit vérifiée ou non, pour rédiger et confirmer la véracité de la précondition et de la postcondition. Il est important que la postcondition d'un tour de boucle soit une précondition pour le tour suivant. En logique de Hoare, on fait apparaître explicitement la condition de boucle autour de celle-ci.

Si la place le permet (ce qui n'était pas le cas ici), indenter les préconditions et postconditions est une bonne idée.

On remarquera que les boucles `for` du langage C présentent une difficulté car initialisation, condition de sortie et mise à jour sont écrites sur une même ligne alors même qu'elles s'exécutent à des moments différents. Il ne faudra pas hésiter à écrire un programme équivalent dans l'optique de la preuve.

```
int position_max (int *t, unsigned int n)
{
// [Spécification et restriction sur les arguments.]
// On suppose que n est strictement positif
// et que c'est la taille de t.
  int rep = 0;
  int i = 1;
  // <- rep est un indice du plus grand élément de t jusqu'à i exclu
  while (i < n)
  {
    // <- rep est un indice du plus grand élément de t jusqu'à i exclu
    if (t[i] > t[rep])
    {
      reponse = i;
    }
    // -> rep est un indice du plus grand élément de t jusqu'à i+1 exclu
    i += 1;
  }
  // -> i vaut n et rep est un indice du plus grand élément de t jusqu'à n exclu
  return rep;
  // donc la fonction retourne une position du maximum de t
  // (en rédigeant un peu plus, on peut préciser que c'est le premier
  // s'il y en a plusieurs)
}
```

Un exemple en OCaml, illustrant comme pour le cas précédent qu'on peut être amené à dilater le code (on avait utilisé des accolades non nécessaires pour aérer, et on pouvait éventuellement ajouter encore une précondition et une postcondition autour, voire en cas d'excès de zèle un else vide commenté) :

```
let sorteren tab =
(* t est un tableau dont tous les éléments sont 0, 1 et 2 *)
  let deb = ref 0 in
  let fin = ref (Array.length tab - 1) in
  let ind = ref 0 in
  while !ind < !fin do
    (* <- 0...0 (deb) 1...1 (ind) ?...? (fin) 2...2 *)
    if tab.(!ind) = 1 then
    (
      (* <- 0...0 (deb) 1...1 (ind) 1?...? (fin) 2...2 *)
      incr ind
      (* <- 0...0 (deb) 1...11 (ind) ?...? (fin) 2...2 *)
    )
    else if tab.(!ind) = 0 then
    (
    (* <- 0...0 (deb) 1...1 (ind) 0?...? (fin) 2...2 *)
      tab.(!ind) <- tab.(!deb);
      tab.(!deb) <- 0;
      incr deb;
      incr ind
    (* -> 0...00 (deb) 1...1 (ind) ?...? (fin) 2...2 *)
    )
    else
    (
      (* <- 0...0 (deb) 1...1 (ind) 2?...? (fin) 2...2 *)
      tab.(!ind) <- tab.(!fin);
      tab.(!fin) <- 2;
      decr fin
      (* -> 0...0 (deb) 1...1 (ind) ?...? (fin) 22...2 *)
    )
  done
 (* -> 0...0 (deb) 1...1 (ind = fin) 2...2 *)
;;
```

Pour la lisibilité, les préconditions et postconditions n'étaient pas des assertions mais leur signification peut être clarifiée par ailleurs (et en pratique il ne faut pas hésiter à écrire préconditions et postconditions dans une zone bien délimitée du code sans se limiter à ce qu'on peut écrire en tant que commentaire, quand on écrit un programme sur une copie) : `0...0 (deb) 1...1 (ind) ?...? (fin) 2...2` signifie qu'il y a un certain nombre de zéros au début du tableau (potentiellement aucun), jusqu'à l'indice `deb` exclu (ce n'était pas clair), puis idem pour un et `ind`, puis des valeurs indéterminées jusqu'à l'indice `fin` **inclus**, puis idem pour deux.

2.3 Programmation défensive, assertions

Nous verrons dans le chapitre suivant le besoin de tests réguliers au cours de l'écriture d'un programme, notamment s'il comporte de nombreuses fonctions.

Une source possible d'erreurs peut être l'utilisation de valeurs imprévues pour une certaine fonction `f`, faisant que celle-ci déclenche une erreur alors même qu'elle a bien été écrite.

Cela signifierait que sa spécification n'a pas été respectée du point de vue des contraintes sur les arguments.

Pour traiter ce genre de problèmes, le début du code de la fonction peut être assorti d'assertions, qui existent en OCaml et en C (ainsi qu'en Python, où elles ont fait leur entrée au programme officiel).

La syntaxe est la suivante : `assert condition` (OCaml) / `assert (condition)` (C).

Il faut noter qu'il ne s'agit pas d'une fonction en OCaml mais d'un mot-clé, et que `assert condition` est une expression de type `unit` nécessitant que `condition` soit de type booléen. La sémantique est la suivante : `assert condition` ne fait rien si `condition` s'évalue à `true` et déclenche une erreur localisant l'assertion ayant échoué sinon.

En C, le principe est le même, et la fonction `assert` est définie dans `assert.h`, qu'on peut également supposer être disponible dans les programmes qu'on écrit. L'avantage dans ce langage est que l'expression qui s'est évaluée à `false` (en pratique `0`) est reportée dans le message d'erreur.

Attention, la vérification des assertions compte dans la complexité tout comme n'importe quel autre morceau de code. Pour cette raison, il peut être recommandé de retirer les assertions une fois la phase de tests passée et qu'on s'est assuré que les assertions réussissent toujours dans le programme définitif.

Pour éviter de repasser sur l'ensemble du code, on peut demander au compilateur d'ignorer les assertions, par une option nommée `-DNDEBUG` en C et `-noassert` en OCaml. Pour autant, dans ce dernier cas, une assertion dont l'énoncé se limite à `assert false` n'est jamais ignorée et peut servir de déclenchement d'erreur à la place de `failwith`.

Chapitre 3

Validation, test

Partie du cours : Méthodes de programmation
Prérequis : Connaissances de base
Applications : Tout le reste de la vie !
Compétence visée principale : Imaginer et concevoir une solution

---

3.1 Introduction

Les preuves vues précédemment permettent de garantir qu'un programme, en admettant qu'il suive à la lettre l'algorithme qui a de manière sous-jacente fait l'objet de la preuve, respecte bien sa spécification. Cependant, elles ne permettent pas toujours de détecter des bugs plutôt dus à l'inattention qu'à une erreur de conception, et il n'est pas exclu qu'un programme n'ait pas le fonctionnement espéré malgré une preuve en bonne et due forme.

"Beware of bugs in the above code ; I have only proved it correct, not tried it."
(Donald Knuth)

La phase de tests est une étape incontournable du développement d'un programme. Il ne s'agit pas de remplacer la preuve mais de la compléter.

"Program testing can be used to show the presence of bugs, but never to show their absence !" (Edsger Wybe Dijkstra)

En pratique, suivant la spécification de la fonction, il est parfois suffisant pour se rassurer de cibler des instances qui donneraient une idée générale du comportement de la fonction.

Par exemple, une recherche dichotomique a de grandes chances de fonctionner systématiquement quand elle fonctionne pour localiser un élément qui est le premier d'une structure, le dernier de la même structure et à quelques endroits arbitraires, en prenant des tailles de structure qui sont des puissances de deux, des puissances de deux moins un et aussi des autres valeurs.

C'est l'expérience et la connaissance du problème à traiter qui permettront de trouver des valeurs pathologiques à cibler en priorité.

Il est intéressant de noter que la validation automatique d'un programme, ne pouvant a priori pas faire de tests exhaustifs sur toutes les entrées, ni accepter un unique programme déclaré comme solution pour des raisons évidentes, peut se baser sur des tests aléatoires et faire considérer comme correct un programme qui a peut-être simplement passé tous les tests par chance. Ainsi, même une fonction répondant au hasard pourrait être, avec une probabilité non nulle, considérée comme correcte (on est ici dans un cas extrême des algorithmes probabilistes qui sont au programme en deuxième année).

Être amené à écrire son propre jeu de tests est une compétence attendue au programme, elle est amenée à être évaluée dans les rendus à l'avenir.

3.2 Graphe de flot de contrôle

La structure de *graphe de flot de contrôle* permet d'abstraire un programme (plutôt écrit en style impératif) et de représenter l'exécution de ce programme en suivant un chemin dicté par les valeurs des variables associées, en particulier ce chemin dépend de l'entrée du programme.

La représentation usuelle est d'encadrer dans des rectangles tous les blocs d'instructions consécutifs, et d'entourer par des cercles toutes les conditions, qu'elles proviennent de tests ou de boucles. En pratique, les boucles inconditionnelles seront traitées comme des boucles conditionnelles, et remplacées implicitement par un code équivalent, même quand le nombre de tours sera fixé et connu.

On obtient alors un graphe en ajoutant des flèches entre les objets ainsi formés, sachant que de tout bloc part une flèche et de toute condition en partent deux, étiquetées par vrai et faux.

L'entrée et les sorties du programme sont mis en évidence, et l'étude des chemins entre ces deux points permet de vérifier que par exemple :

— Aucun chemin ne termine dans une sortie associée à un message d'erreur quand l'entrée respecte certaines conditions.
— Il existe une valeur de l'entrée qui permet à un chemin d'être exploré (on parle de *chemin faisable*), ou non (on parle alors de *chemin infaisable*).

Des instances de test intéressantes correspondront à différents chemins qu'on souhaiterait parcourir dans le graphe de flot de contrôle.

Attention : Déterminer si un chemin est faisable est un problème indécidable. Ceci ne peut être vérifié par l'ordinateur. On peut éventuellement le vérifier à la main (si c'est oui), le prouver à la main (si c'est non), ou déterminer les conditions pour que le chemin puisse être emprunté, conditions qui peuvent ne jamais être remplissables car elles portent une contradiction en elles.

Une conséquence intéressante est que le programme en C ci-après déclenche un avertissement

```
  warning: control reaches end of non-void function [-Wreturn-type]
```

```
int abso(int x)
{
  if (x > 0) return x;
  else if (x <= 0) return -x;
}
```

De même, en OCaml, on a un avertissement `Warning 8 [partial-match]: this pattern-matching is not exhaustive. All clauses in this pattern-matching are guarded.` sur le programme équivalent :

```
let abso x = match x with
  | x when x > 0 -> x
  | x when x <= 0 -> -x;;
```

... et ceci n'empêche en particulier pas les erreurs de typage si on enchaîne `if condition` et `else if not condition` avec un type différent de `unit`.

3.3 Test exhaustif de la condition d'une boucle ou d'une conditionnelle

À la lumière de ce qui précède, lorsqu'une condition dans le programme est composée de plusieurs booléens reliés par des opérateurs, il sera pertinent de produire des instances de tests qui donnent toutes les possibilités non seulement de résultat du test, mais aussi de booléens intermédiaires, par exemple pour qu'une conjonction puisse être rendue fausse de deux façons différentes.

Le cas d'application le plus simple qu'on puisse imaginer est une vérification globale sur un tableau, avec un indice qui augmente et un test effectué pour chaque indice. La condition de la boucle sera a priori de la forme « tant que l'indice est inférieur à la taille du tableau et que le test est vérifié ». Suivant la raison pour laquelle cette condition finit par être fausse, la réponse de la fonction écrite sera un booléen différent. Il s'agit de confirmer par les tests effectués que la pratique confirme la théorie... et la spécification.

De manière plus générale, dans le graphe de flot de contrôle, il peut être intéressant de voir comment faire pour ne pas entrer dans une boucle conditionnelle (peut-être est-ce impossible), comment y faire un seul tour, ou deux tours exactement, éventuellement que la boucle soit infinie (on espère que c'est impossible !)... Cette dernière vérification constitue également un problème indécidable, puisqu'on y retrouve le problème de l'arrêt, également au programme de deuxième année.

Chapitre 4

Récursivité

Partie du cours : Récursivité et induction
Prérequis : Programmation en OCaml
Applications : Tout le reste de la vie !
Compétence visée principale : Imaginer et concevoir une solution

---

L'idée même de la *récursivité* a déjà été traitée en début d'année en s'appuyant sur l'intuition et sur les mathématiques enseignées sur les premiers jours. Tout se passait un peu comme si c'était un prérequis finalement. Ainsi, ce chapitre va surtout en présenter les détails techniques du point de vue de l'implémentation d'une récursion.

4.1 Introduction

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

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 OCaml 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 :

```
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 OCaml 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, OCaml 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 simultanée des fonctions qui s'appellent mutuellement. Ainsi, on écrira :

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

4.2 Pile d'appels

Lorsqu'on appelle une fonction récursive, les instructions sont comme mises sur plusieurs niveaux, 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, alors même que la complexité en temps n'est parfois pas affectée.

La notion hors programme de récursivité terminale indique dans quels cas on peut échapper à ce souci, sachant que les programmes dits récursifs terminaux sont optimisés en OCaml (par ailleurs, Python ne le fait pas).

Il est possible qu'un appel d'une fonction récursive procède à plusieurs appels récursifs de la fonction.

Cela engendre usuellement une explosion de la complexité, et la suite du chapitre permettra de confirmer théoriquement la complexité attendue par le calcul.

Formellement, la notion de pile d'appels reste en vigueur, mais il est plus pratique de considérer que les appels s'organisent selon une structure arborescente, et cette structure d'arbre est fondamentale en informatique ; elle sera vue en détail plus tard dans l'année.

Par exemple, un programme naïf ci-après pour calculer un coefficient binomial $\binom{n}{k}$ utilise la formule additive de Pascal, et les appels récursifs imbriqués explorent alors (plusieurs fois, et c'est bien le drame) tous les points d'une zone rectangulaire du triangle de Pascal de coins opposés `(0, 0)` et `(n, k)`.

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

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.

Pour autant, il n'est pas obligatoire que les programmes récursifs soient aussi naïfs que l'exemple précédent, et nous verrons plus tard une méthode pour éviter de recalculer des valeurs utilisées plusieurs fois dans l'arbre d'exécution suggéré par l'intuition.

Toujours est-il que quand on écrit un programme récursif, on traite souvent les cas de base en premier et on cherche comment organiser les appels ensuite.

4.3 Preuve d'un programme récursif

Prouver la terminaison d'un programme récursif nécessite une récurrence permettant de garantir qu'un cas de base, donc sans appel récursif, est forcément atteint.

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

La raison est que l'ensemble $\mathbb{N}$, muni de l'ordre usuel, est bien fondé (il n'admet pas de suite infinie strictement décroissante). Ainsi, si tous les appels imbriqués occasionnent pour le paramètre choisi une suite strictement décroissante, ces appels finissent par atteindre un cas de base (éventuellement une erreur, mais cela fait aussi terminer le programme), car l'arborescence des appels est de profondeur finie.

On comprendra alors que, plutôt que d'utiliser un entier positif, une valeur prise dans un ensemble bien fondé suffit. Attention à toujours préciser l'ordre, car c'est grâce à lui qu'on dit que l'ensemble est bien fondé. En fait, la notion à l'origine est celle d'ordre bien fondé.

Pour la culture, un bon ordre est une relation d'ordre telle que toute partie non vide ait un plus petit élément, et un ensemble muni d'un tel ordre est dit bien ordonné. [1]

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.

Souvent, le plus simple est cependant de se fonder sur le variant établissant la terminaison du programme [2] et de faire une récurrence forte sur ce variant.

En outre, la mention que les programmes récursifs s'écrivent souvent plus facilement s'applique aussi aux preuves de correction : si les cas de base et les appels récursifs sont des transcriptions de formules mathématiques, la preuve peut être immédiate.

Au deuxième semestre, une technique qui généralise la récurrence sera enseignée, et elle s'appuiera sur la structure sur laquelle la récurrence est faite.

1. Exercice de mathématiques : prouver que bien ordonné implique bien fondé, avec une équivalence si l'ordre est supposé total.

2. adapter ceci dans le cas exceptionnel où il faudrait prouver une correction partielle, c'est-à-dire la correction du programme dans les cas où il termine...

4.4 Complexité

Nous nous limitons ici à la complexité dans le pire des cas. Il faut savoir que le meilleur des cas s'obtient souvent en ayant l'intuition de valeurs particulières du paramètre permettant de rejoindre plus rapidement les cas de base.

En outre, chercher la complexité moyenne et le coût amorti reste possible dans le cas de récursions, quand l'algorithme étudié s'y prêtera le mieux.

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 des 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 éviter de se surcharger, on peut être amené à rassembler deux valeurs proches dans une formule, comme par exemple en découpant une structure en deux parts équilibrées, considérer $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$ séparément n'a pas vraiment d'intérêt.

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).

La plupart du temps, on a une suite récurrente linéaire, mais la dichotomie est un contre-exemple majeur, et plus généralement les algorithmes de type « diviser pour régner », vus ultérieurement, auront des formules décrivant la complexité qu'un théorème hors programme permet de résoudre simplement.

Il reste néanmoins possible de faire une étude au cas par cas, souvent en considérant c_{b^k} après avoir posé $n = b^k$ la taille de l'entrée supposée une puissance d'un certain b, correspondant au nombre de parts quand un algorithme fractionne l'entrée (donc deux pour la dichotomie), et en introduisant la suite $T_k = c_{b^k}$ qui se trouve souvent être récurrente linéaire.

Donnons dès à présent les complexités les plus classiques dans le cas de fonctions récursives vues en classes préparatoires, à partir de la relation de récurrence déterminée en analysant le programme, en notant c_n le coût pour une valeur n d'une expression dépendant des arguments, en pratique souvent l'un d'entre eux.

— $c_n = c_{n-1} + \mathcal{O}(1) : c_n = \mathcal{O}(n)$.
— $c_n = ac_{n-1} + \mathcal{O}(1)$, $a > 1 : c_n = \mathcal{O}(a^n)$.
— $c_n = c_{n-1} + \mathcal{O}(n) : c_n = \mathcal{O}(n^2)$.
— $c_n = nc_{n-1} + \mathcal{O}(1) : c_n = \mathcal{O}(n!)$.

Comme annoncé précédemment, les formules qui suivent seront prouvées dans un chapitre ultérieur.

— $c_n = c_{\frac{n}{2}} + \mathcal{O}(1) : c_n = \mathcal{O}(\log n)$.
— $c_n = c_{\frac{n}{2}} + \mathcal{O}(n) : c_n = \mathcal{O}(n)$.
— $c_n = 2c_{\frac{n}{2}} + \mathcal{O}(1) : c_n = \mathcal{O}(n)$.
— $c_n = 2c_{\frac{n}{2}} + \mathcal{O}(n) : c_n = \mathcal{O}(n \log n)$.
— $c_n = ac_{\frac{n}{b}}$, a et b étant des entiers strictement positifs : $c_n = \Theta(n^{\log_b a})$.

Dans toutes ces formules, apporter une précision en remplaçant $\mathcal{O}$ par Θ dans la relation de récurrence fait que le Θ se maintient dans la complexité qui en résulte.

Chapitre 5

Gestion de la mémoire d'un programme

Partie du cours : Gestion des ressources de la machine
Prérequis : Connaissances de base
Applications : Programmation en C, compréhension des écueils de base en programmant
Compétence visée principale : Mettre en œuvre une solution

Le programme officiel exige de découvrir les bases de la gestion de la mémoire d'un ordinateur lors de l'exécution d'un programme.

Ceci justifie le choix d'un langage de bas niveau comme le C parmi les langages d'enseignement de la filière, où l'on ne bénéficie pas de la gestion automatique de la mémoire comme en OCaml.

L'architecture des ordinateurs étant cependant hors programme, les lecteurs curieux pourront se renseigner d'eux-mêmes sur le sujet pour leur culture.

La partie sur la gestion des ressources de la machine, et tout particulièrement le chapitre suivant, est l'occasion de s'intéresser aux systèmes d'exploitations libres.

En particulier, en cas de divergences, c'est Linux qui servira d'exemple.

Ce chapitre présuppose une prise en main préalable du langage C, permettant de s'appuyer sur un « comment » déjà su, et on s'efforcera ici d'expliquer « pourquoi ».

5.1 Pile et tas

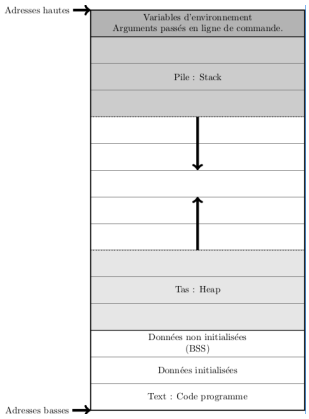
Un programme compilé travaille sur de la mémoire *volatile*, définie comme de la mémoire qui ne demeure pas après extinction de la machine, et effectivement cette mémoire se perd après la fin de l'exécution du programme. Il s'agit plus précisément de la *mémoire vive*.

En pratique, il est également possible pour un programme de travailler sur de la mémoire *de masse* (donc non volatile) en tant que fichiers annexes ouverts en mode lecture ou écriture. Ceci sera traité au chapitre suivant car c'est hors sujet par rapport au propos actuel.

Les adresses manipulées sont en pratique virtuelles, et tiennent actuellement sur huit octets (en admettant un processeur 64 bits), quelle que soit la place effective disponible dans la machine. Ceci permet en cas de besoin d'aller chercher ailleurs que dans la mémoire vive, si celle-ci fait défaut.

Un autre avantage pratique est que des processus différents ne partageront normalement pas la même mémoire, le contraire serait source de risques comme évoqué précédemment lors de la présentation sommaire de `malloc` et `free` dans le chapitre sur le langage C.

L'organisation de la mémoire virtuelle est la suivante :



— Les plus petites adresses contiennent le code source du programme (car il prend effectivement de la place lui-même), qui ne sera pas modifié. La taille est constante et connue à l'avance.
— Les adresses suivantes contiennent les données statiques initialisées (à la compilation, typiquement les chaînes de caractères écrites telles quelles) puis les données statiques non initialisées (il s'agit par exemple de variables globales simplement déclarées en C, dont la manipulation ne peut pas se faire en dehors d'une fonction, comme on a pu le constater en observant des erreurs sibyllines si on l'essayait). La taille est effectivement constante et connue à l'avance. Ces données seront amenées à être modifiées (sauf les `const`).
— Les adresses suivantes constituent *le tas*. Il s'agit de mémoire libre qui sera réservée dynamiquement au fur et à mesure des besoins, en laissant potentiellement des trous se créer en fonction de l'évolution des allocations et libérations.
— Il y a ensuite une zone libre car par la suite les allocations se font en descendant depuis les plus grandes adresses.
— Les adresses les plus hautes constituent *la pile*. Il s'agit de mémoire libre qui sera utilisée au fur et à mesure de l'exécution, avec une allocation fixe au moment où une fonction se déclenche et une libération dès que la fonction se termine, en respectant les principes d'une pile.
— Les toutes dernières adresses contiennent les arguments et variables d'environnement. C'est par exemple là qu'on trouvera `argv`. On ne peut pas modifier cette zone non plus.

Un *bloc d'activation* d'une fonction est un bloc de mémoire stocké dans la pile et contenant les informations utiles pour exécuter la fonction : quelles sont ses variables globales, où on en est de l'exécution de la fonction (à quelle ligne, pour comprendre l'idée), et où la valeur de retour sera transmise.

L'illustration de la pile d'appels pour une fonction récursive a pu montrer que les valeurs renvoyées, qui étaient utilisées « magiquement » au bon endroit, étaient à gérer d'une manière ou d'une autre concrètement, tout comme le fait qu'une fonction mise en attente par l'appel prioritaire d'autres fonctions redémarrait tout aussi magiquement au bon endroit.

La pile d'appels a une taille limitée, de l'ordre de quelques mégaoctets. Au-delà (par exemple en raison d'une récursion infinie), le programme déclenchera un *dépassement de pile* (le fameux *stack overflow*). Concrètement, il tentera d'accéder à une zone mémoire interdite, ce qui déclenche une *erreur de segmentation* (*segmentation fault*).

5.2 Portée, durée de vie

Une variable a une *portée*, qui correspond aux parties du programme où cette variable est utilisable. Dans la plupart des cas, c'est la même chose que la *durée de vie* de la variable, qui correspond au moment entre sa création et sa disparition. Les variables statiques en C donnent un cas de divergence, mais elles ne sont pas au programme. *Stricto sensu*, la durée de vie d'une variable globale est l'ensemble de l'exécution du programme, vu que la mémoire associée est réservée dans la zone des données, initialisées ou non, du début à la fin. Cependant, sa portée est la portion du programme qui commence à l'initialisation.

Suivant le langage, la portée des variables manipulées par une structure de contrôle peut varier. Par exemple, la variable créée par une boucle `for` en Python ont une durée de vie dépassant la boucle elle-même, mais ce n'est pas le cas en C ni en OCaml. La portée d'une variable en C est l'ensemble du contexte où elle est déclarée : une fonction, voire un bloc délimité par une accolade, ou d'un autre côté l'ensemble du programme. En OCaml, on retrouve aussi des variables globales et des variables locales, ces dernières voyant leur déclaration assortie d'un `in` (ou alors il s'agit de la variable d'une boucle, comme dit juste avant). Les arguments d'une fonction (notamment dans le cas de C et OCaml où le passage des arguments se fait par valeur) agissent comme des variables locales, mais n'en sont pas si l'on s'en réfère à la définition usuelle.

Si une variable a une portée plus étendue qu'une autre variable de même nom, la dernière à avoir été créée masque sur l'ensemble de sa durée de vie la précédente. Il est possible que ce faisant la première à avoir été déclarée n'ait plus jamais l'occasion de réapparaître. Masquer une variable globale par une variable locale ou une variable locale par une variable de boucle est souvent une mauvaise pratique du point de vue de l'esthétique, mais on trouvera parfois des cas où c'est fait exprès. Une utilisation potentiellement sujette à débat mais qui a de la pertinence est de masquer une variable locale qui est un argument de fonction par une copie de cette variable afin de préserver l'objet mis en argument des mutations qui auront lieu.

En C et en OCaml, la portée d'une variable est connue à la compilation, ce qui est très pratique. Ce n'est pas le cas de tous les langages. L'avantage est qu'on peut détecter les erreurs consistant à utiliser une variable non définie au moment de la compilation et pas lors de l'exécution (en Python, qui n'est certes pas un langage compilé, l'erreur peut arriver à n'importe quel moment).

5.3 Allocation dynamique

Cette section servira de rappels et de compléments par rapport à l'allocation dynamique de mémoire en C.

La fonction `malloc`, nécessitant d'inclure `stdlib.h`, permet de chercher un bloc de mémoire disponible **sur le tas**, dont la taille en octets est mis en argument. **La mémoire ainsi réservée n'est pas initialisée et ne doit pas être lue avant qu'on écrive dessus**. En cas de réussite, la fonction `malloc` retourne un pointeur vers le début de la zone mémoire, en cas d'échec elle retourne le pointeur `NULL` (on peut supposer que ceci ne se produira pas, et se permettre de ne pas vérifier après chaque appel que le pointeur retourné n'est pas `NULL`, sachant que tenter de déréférencer le pointeur `NULL` cause une erreur de segmentation).

Dans la mesure où le type des éléments gérés par le pointeur va être fixé au moment de la déclaration du pointeur, cette zone mémoire peut contenir un nombre arbitraire d'éléments du type en question (un nombre qui est connu au moment où la fonction est appelée, mais qui n'est pas récupérable automatiquement sinon), et le pointeur sera géré comme un tableau avec la même syntaxe. C'est par ailleurs ainsi qu'on pourra s'en servir comme argument ou valeur de retour d'une fonction.

En tant qu'exemple rare de polymorphisme en C au programme de MP2I, le prototype mentionne une valeur de retour de type `void*`. Caster le résultat de cette fonction,par exemple `int* p = (int*) malloc(4 * sizeof(int));`, est nécessaire en C++, mais pas en C. Ceci étant, il faut mentionner cette possibilité pour aborder le *transtypage* mentionné dans le programme officiel.

Quand l'utilisation d'une zone mémoire allouée par `malloc` est terminé, et dans tous les cas **avant de perdre l'information de l'adresse**, il faut libérer cette zone avec la fonction `free`, également dans `stdlib.h`. On retiendra que tout ce qui est alloué avec `malloc` doit être libéré avec `free` **et uniquement ceci**.

On appelle *fuite de mémoire* le fait qu'un bloc alloué dynamiquement ne soit pas libéré, et l'option de compilation `-fsanitize=address` permet de détecter toutes les fuites de mémoire, même celles qui sont laissées par une allocation dans la fonction `main` en voulant tenir compte du fait que la fin de l'exécution du programme libère tout.

Il est bon de prendre l'habitude de libérer correctement la mémoire, par exemple pour des structures séquentielles (pointeur de pointeurs ou liste chaînée, suivant les implémentations), l'ordre est important, car on n'est plus censé accéder à de la mémoire après sa libération. On observe par ailleurs une erreur quand on tente de libérer une deuxième fois de la mémoire déjà libérée par `free`.

Les fonctions `calloc` et `realloc` ne sont pas au programme, on évitera de les utiliser dans une production évaluée.

En OCaml, où la gestion de la mémoire est automatique, le besoin de créer des variables donne lieu à une allocation dynamique sans intervention de l'utilisateur, et la libération est également faite dès que c'est nécessaire par l'intervention d'un *ramasse-miettes* (*garbage collector* en anglais), qui se base sur une information associée à chaque bloc de mémoire concernant le nombre de références vers ce bloc.

Chapitre 6

Gestion des fichiers et entrées-sorties

Partie du cours : Gestion des ressources de la machine
Prérequis : Connaissances de base
Applications : Manipulations dans l'utilisation quotidienne d'exécutables qu'on a créés soi-même
Compétence visée principale : Mettre en œuvre une solution

Ce deuxième chapitre sur les ressources de la machine présente aussi des connaissances utiles pour l'utilisation de l'ordinateur au quotidien.

Certaines notions auront sans doute été découvertes, du moins leur intuition, par des manipulations antérieures.

Il s'agit également de préparer le chapitre sur la concurrence et la synchronisation, au programme de seconde année, et d'appuyer les chapitres techniques sur les langages de programmation en comprenant mieux les actions effectuées, et tout d'abord permises, sur les fichiers.

6.1 Interface de fichiers

En manipulant un fichier, que ce soit en lecture ou en écriture, à partir d'un programme que l'on écrit, on peut se rendre compte que les fonctions associées utilisent un curseur qui se place à l'endroit où les données seront lues ou écrites.

En pratique, le contenu des fichiers s'apparente à une structure de liste chaînée vue en fin de premier semestre, et demander le n-ième octet d'un fichier nécessite de parcourir les précédents en décalant d'autant le curseur de lecture, placé au début du fichier lors de l'ouverture.

Lorsque les données lues ont subi un certain encodage, cette lecture séquentielle permet de reconstituer les informations sans risquer de démarrer en plein milieu d'un objet, en particulier. On pourra se référer aux exemples de sérialisation vues cette année en cours, TD ou TP.

Dans l'ordinateur, les fichiers sont stockés selon une certaine logique, qui dépend du système de fichiers. Pour l'utilisateur cependant, une vue hiérarchique sera présentée, sous forme d'une arborescence.

La racine de l'arborescence sera un dossier[1] appelé `/` sous Linux, ou `C:\` (éventuellement une autre lettre) sous Windows. C'est d'elle que partent tous les chemins absolus. Au contraire, les chemins relatifs partent du dossier dit courant, où les programmes considèrent être à leur lancement, mais peuvent aussi avoir été déplacés par des commandes de navigation. Ces commandes ainsi que d'autres sont supposées connues à ce stade de l'année (relire le TP 0 à ce sujet).

6.2 Blocs et nœuds d'index

Pour la rapidité d'accès aux données, il est idéal que les fichiers d'un même dossier soient dans des blocs consécutifs de la mémoire, et bien entendu que chacun tienne en un morceau. Cependant, la taille d'un fichier peut varier au fur et à mesure de ses manipulations, et des trous risquent de se former, voire pire, des déplacements bien plus loin dans la mémoire faute d'espace suffisant.

Les systèmes de fichiers Unix utilisent ce qu'on appelle un *nœud d'index* (en anglais *inode*) pour caractériser un fichier.

Un dossier est alors une liste de couples formés par le nom et le nœud d'index de chaque fichier qui s'y situe du point de vue de l'utilisateur.

1. On dit « répertoire » en informatique, mais Windows a tellement popularisé le terme de « dossier »...

Les informations dans le nœud d'index dépendent de la norme (on citera POSIX comme norme principale, encore mal respectée par Windows), mais on retiendra surtout la taille en octets, l'emplacement (dans quel périphérique et à quelle adresse), le type, le propriétaire et les droits d'accès, la date de dernière modification, et on notera que le nom n'en fait pas partie, ce qui est cohérent avec ce qui précède.

Les métadonnées en question sont essentiellement accessibles en consultant les propriétés d'un fichier.

Chaque dossier commence par deux pseudo-fichiers associés à un nœud d'index chacun, qui sont `"."` (dossier courant) et `".."` (dossier parent). Ils apparaissent en premier lorsqu'on affiche le contenu du dossier avec `ls`.

On pourra faire le parallèle avec les tableaux de pointeurs en C, qui contiennent des adresses placées à la suite vers des endroits potentiellement éloignés (mais là aussi, plus ils sont proches et mieux c'est, d'où le fait que les tableaux de tableaux soient en fait nécessairement à la suite et doivent être de taille fixe et connue selon toutes les dimensions au-delà de la première).

De la même façon qu'on peut avoir deux pointeurs égaux, donc pointant sur la même adresse mémoire, deux « fichiers » peuvent correspondre à la même adresse mémoire.

En pratique, il s'agira de créer un nouveau nœud d'index correspondant au même emplacement, en tant que *lien physique* (forcément vers un fichier et non un dossier, et dans le même périphérique, pour information). Ce nœud d'index sera associé à un nom identique ou non (mais forcément différent si le dossier est le même).

Il est aussi possible de créer un *lien symbolique* (les fameux « raccourcis » de Windows), avec un nœud d'index contenant comme seule information l'emplacement.

En pratique, le nombre de liens est également une information d'un nœud d'index, afin que lors de la suppression d'un fichier (qui n'est pas un recyclage de la mémoire, comme on peut le voir à d'autres occasions cette année) le système puisse déterminer si la mémoire doit rester réservée ou si elle est de nouveau disponible.

Ceci fait écho à l'explication sommaire de la gestion de la mémoire en OCaml avec le *"refcount"*.

Enfin, on notera que les nœuds d'index peuvent se trouver sur la mémoire de masse ou sur la mémoire vive, auquel cas ils sont effacés lors de la fermeture de la session (arrêt du programme / de la machine virtuelle / de l'ordinateur).

6.3 Accès, droits et attributs

Parmi les informations des nœuds d'index, les propriétaires et permissions ont été mentionnés. Ce point mérite un peu de détail.

Pour de nombreuses personnes, l'ordinateur est commun à une partie de la famille, et chacun dispose d'un compte.

Pour autant, il n'est pas nécessaire de fractionner le disque dur en autant de parties que d'utilisateurs, en utilisant plusieurs fois de la mémoire pour des données et fichiers système à faire figurer partout.

Ainsi, dossiers et fichiers sont associés à un compte particulier, qu'on appelle leur propriétaire, et qui en gère les permissions.

Cependant, comme il peut être utile d'associer un fichier à plusieurs comptes, en plus de l'utilisateur propriétaire, il y a aussi le groupe propriétaire (qui ne peut cependant pas changer les permissions), sachant que les groupes d'utilisateurs peuvent être constitués de n'importe quel sous-ensemble de l'ensemble des utilisateurs (parfois des utilisateurs virtuels associés à un logiciel).

Les droits de manipulation (lecture, écriture, exécution) d'un fichier sont alors définis séparément pour le propriétaire (code `'u'` pour *"user"*), le groupe propriétaire (code `'g'` pour *"group"*) et les autres (code `'o'` pour *"others"*), sachant que les droits s'appliquent de manière prioritaire dans cet ordre (si par hasard le propriétaire n'a pas les droits d'écriture mais les autres utilisateurs si, il ne pourra pas écrire dedans, aussi incongrue que soit cette situation).

La syntaxe sous Linux est la suivante : `chmod q*t nom`, où `q` est remplacé par le code en question, avec la possibilité d'écrire `'a'` pour `"all"`, modifiant les permissions pour tout le monde à la fois, `*` est remplacé par le symbole `+` ou le symbole `-` suivant qu'on veuille ajouter ou retirer des droits, et `t` est remplacé par une ou plusieurs lettres parmi `r`, `w` et `x`, respectivement pour la lecture, l'écriture et l'exécution.

Il existe aussi une syntaxe utilisant un nombre de trois chiffres entre 0 et 7, calculés en binaire par l'addition de 4 si les droits en lecture sont alloués, 2 pour les droits en écriture et 1 pour les droits en exécution.

Les chiffres concernent dans l'ordre l'utilisateur, le groupe propriétaire et les autres.

Ainsi, une valeur de 640 permet à l'utilisateur de lire et d'écrire, au groupe propriétaire de lire uniquement et rien aux autres. Pour autant, l'existence du fichier elle-même ne leur est pas cachée, sauf si le dossier lui-même leur est interdit d'accès en lecture.

Attention : Pour pouvoir explorer les sous-dossiers d'un dossier, et même s'y déplacer, ce dossier doit être exécutable ! Les droits en lecture permettent alors de consulter la liste des fichiers, et les droits en écriture d'y ajouter ou supprimer des fichiers (les droits en écriture sur le fichier ne suffisent pas).

On notera que dans les chapitres sur C et OCaml, les modes d'ouverture d'un fichier étaient lecture, écriture et ajout, mais l'ajout lui-même ne fait pas l'objet d'une permission particulière et se confond avec l'écriture.

6.4 Fichiers spéciaux et redirections

Comme déjà signalé par ailleurs, trois fichiers sont ouverts de base lors de l'exécution d'un programme : l'*entrée standard* (*stdin*) qui correspond généralement au clavier, la *sortie standard* (*stdout*) et la *sortie d'erreur* (*stderr*), qui est a priori confondue avec la sortie standard et correspond à la console d'un interpréteur, par exemple.

Le mode d'ouverture de ces trois fichiers est évident, par ailleurs.

Pour qu'un programme imprime son compte-rendu ou son résultat sur un autre fichier, on peut simplement l'ouvrir en mode écriture et adapter les fonctions d'impression, mais il y a plus simple : lancer son exécution en *redirigeant* la sortie standard sur un autre fichier.

Sous Linux, la syntaxe est de faire suivre la commande d'exécution du programme par le symbole > puis du nom du fichier (créé pour l'occasion s'il n'existe pas, effacé s'il existe car il est ouvert en mode écriture), sachant que des droits d'écriture sont nécessaires.

Pour ouvrir cette sortie redirigée en mode ajout, le symbole est à écrire deux fois.

En pratique, la redirection de la sortie d'erreur est également possible, avec la syntaxe `2>` ou `2»` suivant le mode là aussi.

Une redirection plus variée, faisant subir une nouvelle commande au résultat de la précédente (les commandes pouvant être en particulier l'exécution d'un programme), utilise le symbole de barre verticale, aussi appelé « *tube* » (*"pipe"* en anglais).

Un exemple issu de nombreuses années de galère avec un certain navigateur :

```
ps -A | grep firefox
```

La commande `ps` correspond au gestionnaire de tâches, et affiche toutes les tâches avec l'option `'A'`.

La commande `grep` est une recherche de motifs, et fait apparaître toutes les lignes du « fichier » en dernier argument (ici la sortie redirigée) contenant la chaîne en premier argument (les guillemets ne sont pas toujours nécessaires).

Ainsi, on affiche ici tous les processus dont le nom contient `"firefox"`. Pour quoi faire ? Hé bien, si ça plante, on retrouve les identifiants des processus à tuer par la commande `kill`. Naturellement, on peut aussi le faire en une fois avec une commande plus élaborée...

Chapitre 7

Types et abstraction

Partie du cours : Structures de données
Prérequis : Connaissances de base
Applications : Toutes les structures de données
Compétence visée principale : Décrire et spécifier

Pour faire un bon programme, il faut généralement organiser les valeurs récupérées, créées et retournées dans une structure permettant leur traitement de manière optimisée.

Le choix pourra être guidé par la structure fournie en argument, mais même celle-ci ne sera pas toujours fournie, et c'est d'autant plus vrai lorsque le programme à écrire est en fait un projet d'ampleur dont l'ensemble de la conception depuis zéro est à la charge de la personne qui va programmer.

Ainsi, en guise de préparation aux chapitres d'algorithmique du second semestre, l'ensemble de cette partie présentera les structures de données les plus classiques, ainsi que la théorie autour de la notion même de structure de données.

Certaines des structures au programme sont exclusivement vues en deuxième année, par ailleurs.

7.1 Types usuels

Les types prédéfinis ne sont certes pas entièrement communs à tous les langages, mais on peut globalement considérer les entiers, flottants et booléens comme des briques de base incontournables. Mis à part pour compenser une place limitée par le nombre de bits alloué quand il est insuffisant, on ne définira pas d'autre type numérique.

Les caractères forment dans l'esprit (et en C) un type numérique, qu'on traitera cependant comme à part, en raison de la distinction entre ce qui est visible et l'encodage par la table ASCII ou une table plus étendue, cette distinction étant plus prononcée que celle qui existe entre la base 10 pour l'affichage des entiers et le binaire pour leur stockage.

Les pointeurs en C et références en OCaml forment un type particulier. On peut les voir comme une simple boîte où se situe de l'information, qui a elle-même un certain type, et cette information peut être amenée à être mutée.

Les raisons d'utiliser des pointeurs en C et des références en OCaml ne sont pas forcément identiques, mais on gardera en tête le principe d'avoir une valeur consistant en un entier, qui est l'adresse mémoire où l'information est effectivement, et le traitement des données à l'adresse en question, qui se présentent de toute manière sous la forme d'un mot en binaire d'une taille stockée en parallèle du pointeur / de la référence, est adapté au type, lui aussi connu.

C'est pour cela qu'on a des `'a ref` pour un certain type `'a` et que les pointeurs ont également une information de type bien visible.

Au sujet des pointeurs, le langage OCaml en dispose aussi de manière interne et non manipulable directement, en ce sens que tout ce qui n'est pas une constante est un pointeur.

Le choix qui a été fait à la conception du langage est que toutes les données utilisent le bit de poids faible, appelé *tag bit*, pour signaler s'il s'agit ou non d'un pointeur (non : valeur 1, oui : valeur 0).

Dans ce cas, les nombres, booléens et caractères n'en sont pas, les constructeurs constants non plus (y compris une liste vide), mais tout le reste pour ainsi dire en est un.

Quand bien même l'arithmétique en est rendue plus compliquée (même l'addition demande une correction), les avantages pratiques font que l'utilisation de ce bit reste répandue.

Les types composés permettent d'associer des valeurs de types différents en une même structure. Cela peut être un simple n-uplet comme en OCaml, où l'on observe un type obtenu par une sorte de produit cartésien, ou une structure / un type enregistrement où les champs / attributs sont nommés et aident à la compréhension des programmes utilisant ces structures lorsqu'on les lit. On peut considérer qu'une fonction a également un type composé, avec un typage qui mentionne le type des arguments et celui de la valeur de retour de manière bien distinctive.

Le tableau forme l'exemple le plus universel de type paramétré, de par la nécessité de pouvoir organiser plusieurs données de même type de manière séquentielle, avec la possibilité d'accéder directement à n'importe laquelle de ces données.

On notera que « directement » ne veut pas dire « immédiatement », et que l'accès à une position veut dire au niveau de la mémoire le calcul de la position en question et le déplacement de la tête de lecture à cette position. C'est en particulier pour cette raison qu'on privilégiera les parcours des tableaux dans l'ordre croissant ou décroissant et non pas une autre permutation, quand bien même elle serait facile à exprimer en termes algorithmiques, à moins bien entendu que l'algorithme qu'on écrit nécessite un ordre particulier. Pour aller plus loin, cela implique également qu'un parcours d'une structure de dimension deux est plus rapide ligne par ligne que colonne par colonne, dans la mesure où cela respecte l'organisation la plus classique dans la mémoire.

On distingue en C les tableaux statiques des tableaux dynamiques en fonction de la zone mémoire où ils sont stockés (voir le chapitre associé). Pour les tableaux dynamiques, l'allocation (dynamique) se fait avec la fonction `malloc` (de `stdlib.h`) et la désallocation (dynamique) avec la fonction `free`. Sans libération explicite, la mémoire reste réservée, quitte à ce qu'elle manque pour la suite de l'exécution du programme, donc il faut prendre garde à libérer la mémoire une fois que l'on ne s'en sert plus et avant que l'endroit où elle est située ne soit oublié, en s'autorisant, à la rigueur, à ne pas le faire uniquement si le programme est censé s'arrêter tout de suite après.

7.2 Structures de données abstraites

7.2.1 Introduction

Une *structure de données abstraite*, en informatique, se résume à un type muni d'opérations théoriques associées. 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.

Par « abstraite », il est impliqué 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.

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 dans le cas précédent).

Bien entendu, une structure de données abstraite peut être implémentée en diverses structures de données concrètes, en utilisant éventuellement des structures de données déjà existantes avec quelques aménagements.

En particulier, on retiendra qu'il y a une différence entre la structure de données abstraite et une implémentation de celle-ci. L'implémentation consiste en un programme effectivement écrit, qui reprend la création du type et l'écriture des fonctions imaginées dans la conception de la structure de données abstraite.

7.2.2 Notion d'interface

La notion de fichier d'en-tête / fichier d'interface a déjà été présentée dans les chapitres traitant des langages de programmation.

Il s'avère qu'une utilité essentielle de ces fichiers, dont on rappelle que l'extension est `.h` en C et `.mli` en OCaml, est de présenter des structures de données abstraites en indiquant le nom de ces structures et les fonctions associées avec leur prototype / signature.

En pratique, ces informations sur le nom et les fonctions forment précisément ce qu'on appelle l'*interface* de la structure de données.

On notera que là aussi le type lui-même n'est pas présenté, donc l'interface ne détaille pas l'implémentation.

7.2.3 « Barrière d'abstraction »

Faisant suite au propos précédent, l'idée de masquer les détails de l'implémentation à l'utilisateur présente de nombreux avantages. Notons que cet utilisateur peut aussi être un programmeur qui doit mettre en œuvre des algorithmes utilisant une structure de données en n'en voyant que l'interface.

On appelle l'ensemble des fonctions dont le prototype / la signature est révélé(e) à l'utilisateur une *barrière d'abstraction*.

Il s'agit de ne permettre l'utilisation d'une structure de données que par le biais de ces fonctions, et d'éventuelles fonctions construites à partir de celles-ci.

Ainsi, par exemple, la structure de pile peut être associée à une barrière d'abstraction limitée à la création, l'empilement, le dépilement et le test de vacuité, mais un utilisateur peut s'en servir pour écrire la fonction calculant la taille d'une pile sans connaître l'implémentation de la pile, et ajouter cette fonction à la liste des fonctions disponibles.

Un premier avantage est donc la dissimulation et la protection du code ainsi masqué : d'éventuelles propriétés dont l'implémentation dépendent risquent d'être perturbées par une intervention directe du programmeur sur les données en connaissant des détails censés rester secrets.

Un deuxième avantage est de pouvoir séparer le travail entre le concepteur de l'implémentation et son utilisateur, après s'être simplement mis d'accord sur l'interface.

Un troisième avantage est de permettre de changer d'avis et de modifier intégralement l'implémentation, tout en préservant l'interface, sans avoir besoin de revenir sur le code où que ce soit.

7.2.4 Invariant de structure

Pour permettre une bonne utilisation d'une structure de données, il est bon de pouvoir garantir qu'elle vérifie constamment certaines propriétés.

De telles propriétés sont appelées *invariants de structure*. Il suffit qu'elles soient vérifiées à la création d'une structure (a priori par une des fonctions de la barrière d'abstraction) et maintenues vraies par les autres fonctions fournies.

L'utilisateur pourra alors profiter de la véracité de ces invariants pour prouver les programmes manipulant ces structures, en s'efforçant de préserver les invariants, soit parce qu'il se limite aux fonctions de la barrière d'abstraction, soit parce que les programmes écrits n'ont pas d'effet sur les invariants (sauf éventuellement pendant la durée de l'exécution d'une fonction, mais dans tous les cas pas à la fin de cette exécution), si possible preuve à l'appui.

Nous reviendrons sur cette notion au deuxième semestre et, pour les structures présentant une récursion, cela nous permettra d'introduire les preuves par induction structurelle.

7.3 Mutabilité

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 OCaml**.

De même, la notion de variable étant abusive dans ce langage, 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 OCaml, car si on recrée la variable `l` pour que cette liste devienne elle-même sans son élément de tête ou avec la même queue mais un nouvel élément de tête, l'ancien élément de tête peut encore être vu comme un objet qui s'enchaîne avec le reste de la liste, mais plus rien ne pointe sur lui-même.

Pour aller plus loin, c'est pour cela qu'en OCaml, un élément d'une liste est stocké en mémoire avec trois informations : la valeur en elle-même, la position de l'élément suivant et une information permettant de savoir si quelque chose pointe dessus. Si ce n'est plus le cas, lors de l'inspection par le ramasse-miettes, la mémoire pourra être libérée, et ceci pourra continuer récursivement pour les éléments au-delà.

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 pointeurs / références.

Chapitre 8

Structures de données séquentielles

Partie du cours : Structures de données
Prérequis : Notions en structures de données
Applications : Écriture de l'essentiel des programmes simples
Compétence visée principale : Mettre en œuvre une solution

Ce chapitre, découpé en deux parties dont seule la première est au programme du premier semestre, présente une vaste collection de structures, ayant toutes en commun de représenter des collections de valeurs organisées dans un ordre particulier. Ces valeurs auront par exemple une position bien identifiée, dans le cas des tableaux ou des tables de hachage, ou des voisins directement accessibles, dans le cas des piles et files entre autres.

Toutes les implémentations en C et en OCaml seront brièvement expliquées en cours et détaillées dans les TP associés.

Les dessins faits au tableau lors des séances seront reproduits dans les notes de cours à un moment ultérieur...

8.1 Liste

La structure de liste traitée ici correspond aux listes en OCaml, c'est-à-dire des *listes chaînées*, à ceci près que le chaînage en OCaml est limité par le fait que la structure y soit persistante.

Dans le cas général, rien ne nous empêche de modifier une liste chaînée, et l'implémentation en tiendra compte.

Une liste chaînée est constituée de ce qu'on appellera des *maillons*. Un maillon est simplement la donnée de deux informations : une valeur (ce qu'on veut effectivement stocker dans la liste chaînée) et un pointeur vers le maillon suivant, ce pointeur pouvant être nul pour le cas du dernier maillon de la liste chaînée.

La liste elle-même se résume à un pointeur vers le premier maillon.

Une opération qui est alors permise dans une liste chaînée est l'insertion d'un nouveau maillon (voire en répétant cette opération d'autant de maillons qu'on veut) à n'importe quel endroit : il suffit de rediriger un pointeur vers le nouveau maillon, dont le pointeur donnera l'ancienne destination du pointeur ainsi redirigé.

De manière analogue, un maillon peut être retiré en le court-circuitant au niveau de son prédécesseur, sur lequel le travail est fait. Il est intéressant de noter que le maillon retiré de la liste chaînée n'est pour autant pas effacé de la mémoire.

De la façon dont on a présenté les listes chaînées, le calcul de la taille se fait en suivant les maillons, donc en temps linéaire. La recherche d'une valeur particulière également, tout comme l'accès au `n`-ième maillon (sous réserve d'existence) dont la complexité est de l'ordre de `n`.

Une autre implémentation possible des listes chaînées se base sur les tableaux, redimensionnables ou non (mais le programme impose une taille maximale donc on partira sur le principe que non). Il s'agit de la même implémentation que celle qu'on verra sur les piles, à la possibilité près de faire une insertion ou une suppression n'importe où, opération causant des décalages et devenant alors de complexité linéaire. L'ensemble des calculs de complexité sera fait dans le TP associé.

Les fonctions récursives (ou faisant appel à des sous-fonctions récursives, ce qui est souvent plus agréable) sont très adaptées à l'exploration des listes chaînées même immutables, ce qui se voyait déjà dans le traitement de la récursion en OCaml.

De plus, certains algorithmes peuvent profiter de l'utilisation d'une structure de taille variable en passant par la liste chaînée, qui sera ici nécessairement modifiable.

Finalement, mentionnons l'existence des listes doublement chaînées, pour lesquelles chaque maillon a également un pointeur vers son prédécesseur. La liste elle-même peut alors contenir un pointeur vers le dernier maillon.

8.2 Tableau redimensionnable

Pour faire écho à la section précédente qui mentionnait la possibilité d'utiliser des tableaux redimensionnables, présentons cette structure en détails.

La structure de *tableau redimensionnable*, qui est implémentée naturellement dans les listes de Python, permet de compenser le manque de dynamisme des tableaux dont le contenu ne pouvait pas être prévu à la création, en particulier la taille n'était pas connue, et aucune borne supérieure n'a été donnée, soit parce qu'elle était également indisponible, soit parce que sa valeur était rédhibitoire.

L'idée est que le tableau redimensionnable soit la donnée d'un support pour les données, vers lequel on pointe de sorte qu'en cas de besoin le pointeur soit redirigé vers un support plus grand.

Ce support aura une certaine capacité, dont tout ne sera pas forcément utilisé, ce qui fait qu'une donnée supplémentaire du tableau redimensionnable est la taille effective du tableau. En ceci on rejoint certaines implémentations des sections suivantes.

De la même façon que pour la fonction `append` de Python, l'augmentation de la capacité lors de l'opération élémentaire de redimensionnement sera exponentielle, cela permettra de reparler de coût amorti lors du calcul de complexité fait en TP.

Le tableau redimensionnable illustre également les propriétés de la barrière d'abstraction : le tableau de support ne sera jamais visible pour l'utilisateur, et les opérations de redimensionnement se feront silencieusement, tout au plus observera-t-on occasionnellement un ralentissement qui le trahit, quand la taille sera assez conséquente.

Il a déjà été observé dans des copies d'étudiants que les tableaux redimensionnables étaient utilisés en OCaml en tant que références de tableaux. En termes d'esthétique, c'est un peu douteux.

Dans tous les cas, sans redimensionnement, on s'interdira bien entendu de faire des références de tableaux !

Les opérations élémentaires sont la création, l'accès en lecture ou en écriture à un indice autorisé, l'ajout d'un élément à la fin ou le retrait du dernier élément (qui est renvoyé par principe).

8.3 Pile

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).

Chaque élément libère donc, une fois retiré, celui qui est juste plus « ancien » que lui dans la pile, de sorte qu'on retrouve l'idée des listes chaînées sans possibilité d'intervention ailleurs qu'au sommet. C'est donc sans surprise que les implémentations seront similaires.

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) et le test de vacuité. On y ajoute parfois l'accès au sommet sans dépilement et le calcul de la taille de la pile, bien que ces dernières opérations puissent déjà s'écrire à partir des précédentes.

La structure de pile peut être persistante ou modifiable selon l'implémentation. Le choix le plus classique sera en pratique surtout fait en fonction du langage.

En OCaml, le type `list` peut déjà être considéré comme une implémentation d'une pile. Il en existe une autre, modifiable quant à elle, dans le module `Stack`, avec le type associé qui est `Stack.t`.

En C (ou en OCaml en fait), mis à part l'implémentation avec une liste chaînée dont on bride l'utilisation, on peut réaliser une structure de pile de capacité limitée à l'aide d'un tablau dont le premier élément indique le nombre d'éléments à considérer comme étant dans la pile, le reste faisant office de mémoire disponible.

Ici encore, le programme suggère de ne pas utiliser de tableaux redimensionnables.

Dans ce cas précis, la pile ne peut contenir que des entiers, alors qu'autrement le type aurait été fixé au moment de définir la structure, mais avec un choix libre.

8.4 File

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.

Les implémentations possibles d'une structure modifiable de file sont une fois de plus par une liste chaînée en limitant la manipulation, ou par un tableau associé cette fois-ci à deux informations (position de la tête et position de la queue, ce qui a l'avantage de limiter la complexité des opérations en ne forçant pas à faire des décalages systématiquement). Il existe en OCaml un type associé, qui est `Queue.t` avec là aussi l'utilisation d'un module éponyme.

Une structure persistante de file peut être simulée par deux piles : l'une pour l'entrée et l'une pour la sortie. En pratique, si les piles utilisées sont modifiables, cela fait une structure modifiable de file. Concrètement, l'enfilement se fait en empilant dans la pile d'entrée, et le défilement en dépilant la pile de sortie si elle n'est pas vide, ou si elle l'est en transférant toute la pile d'entrée dans la pile de sortie. Le calcul de la complexité associée à cette opération est un nouveau cas de présentation du coût amorti.

8.5 Tableau associatif

La structure de *tableau associatif* (le terme équivalent de *dictionnaire* est éventuellement mieux connu) est une structure qui étend la notion de tableau dans la mesure où l'indexation ne se fait plus forcément par des entiers entre zéro et le nombre d'éléments moins un, mais l'ensemble des indices, que l'on appellera plutôt *clés*, peut désormais presque être n'importe quoi et contiendra usuellement des chaînes de caractères.

En pratique, les limitations des langages imposeront que les clés soient tous du même type (en Python ce n'est pas imposé, mais en C et en OCaml oui).

L'implémentation usuelle des tableaux associatifs se fait en réalisant une *table de hachage*, qui utilise un tableau (éventuellement redimensionnable) comme support et qui stocke les couples (clé, valeur) à un indice déterminé par une fonction associée à la table de hachage, appelée *fonction de hachage*, elle aussi a priori flexible en termes de type pour l'argument mais retournant forcément un entier, avec en particulier une zone restreinte par les capacités du processeur (donc un entier sur 64 bits selon toute attente).

Dans l'idée, si on veut stocker par exemple la chaîne de caractères `"Quarante-deux"` dans une table de hachage munie de la fonction `h`, on calcule `h("Quarante-deux")` et le reste de cet entier dans la division euclidienne par la taille du tableau support sera la position où cette clé figurera.

Bien entendu, on a peu d'espoir d'injectivité, et même une garantie de non-injectivité si le nombre de clés dépasse la taille du tableau support, donc il faut trouver une méthode pour gérer la présence de deux clés ayant la même image par la fonction de hachage dans une table de hachage (phénomène que l'on appelle une *collision*).

La méthode que nous utiliserons dans nos implémentations est le *chaînage*, elle consiste à considérer que tous les éléments du tableau support seront des listes chaînées, où se stockeront les couples (clé, valeur) l'une après l'autre quand elles auront effectivement la même image par la fonction de hachage.

La structure de table de hachage est, de par son esprit même, mutable et même redimensionnable.

Les opérations élémentaires sont la création, l'ajout d'une clé en garantissant qu'elle n'y est pas déjà (autrement, soit on remplace la valeur existante, soit on déclenche une erreur), le test de présence d'une clé, la mise à jour de la valeur associée à une clé (si elle n'y est pas, on peut déclencher une erreur ou procéder à l'insertion, créant une redondance entre les opérations élémentaires dont l'une peut finalement être absente suivant l'implémentation), la consultation de la valeur associée à une clé (là aussi, si elle n'y est pas on peut déclencher une erreur) et le retrait d'une clé (idem).

Il peut être problématique qu'une clé dans une table de hachage soit mutable, car la fonction de hachage elle-même tient compte de la valeur d'un objet et non pas de son adresse (ce serait une très mauvaise idée !).

Il y a ici un risque de ne plus retrouver des données stockées à l'indice où on est censé les avoir mises, puisque cet indice a changé. C'est une raison pour laquelle il est très satisfaisant de voir que les chaînes de caractères ne sont plus mutables en OCaml.

De manière plus ou moins réciproque, il est nécessaire que deux valeurs considérées comme égales aient la même image par toute fonction de hachage. C'est en fait surtout important en Python, où l'on peut imaginer des égalités arithmétiques entre des objets de type différent.

Une fonction de hachage appelée sur un entier renverra normalement l'entier lui-même, et pour cette raison des valeurs sensiblement proches de zéro justifieront de faire un simple tableau, quitte à laisser des trous, plutôt qu'une table de hachage. Surtout s'il n'y a vraiment pas de trou !

En TP, une implémentation des tables de hachage à l'aide de tableaux redimensionnables contenant des listes chaînées sera réalisée en C et en OCaml.

En OCaml, il est également possible d'utiliser un type (appelé `Hashtbl.t` et paramétré par `('a, 'b)` représentant le type des clés et celui des valeurs) réalisant une table de hachage, à l'aide du module dont on aura compris le nom. Les opérations élémentaires sont réalisées par les fonctions suivantes :

— `Hashtbl.create n` crée une table de hachage avec `n` places pour commencer, mais en adaptant si besoin (donc on devine `n` sans qu'il n'y ait de risque si l'estimation est mauvaise) ;
— `Hashtbl.add th cle valeur` ajoute une association à la table de hachage, en masquant une éventuelle clé déjà existante (l'autre valeur sera de nouveau accessible en cas de retrait de ce qui l'a masqué) ;
— `Hashtbl.find th cle` détermine la valeur associée à la clé dans la table de hachage, en déclenchant l'erreur `Not_found` si la clé est absente ;
— `Hashtbl.mem th cle` détermine si la clé est présente dans la table de hachage ;
— `Hashtbl.remove th cle` retire une occurrence de la clé dans la table de hachage s'il y en a une (sinon la fonction n'a pas d'effet) ;
— `Hashtbl.replace th cle valeur` remplace la valeur associée à la clé dans la table de hachage par une nouvelle valeur (une éventuelle valeur masquée n'est pas impactée) en ajoutant la clé si elle n'y était pas encore.

À celles-ci s'ajoutent deux fonctions utiles (on ne mentionne pas ici la fonction `fold`, qui peut être plus difficile à maîtriser, à l'image des deux fonctions analogues classiquement utilisées dans le module `List`) :

— `Hashtbl.find_opt th cle` agit comme la fonction `find`, mais retourne une option, donc si la clé est absente aucune exception n'est levée, c'est simplement le cas où `None` est renvoyé ;

— `Hashtbl.iter f th` appelle la fonction fournie, prenant des clés et des valeurs (dans cet ordre) en argument, à tous les éléments de la table de hachage, sans contrôle sur l'ordre, sachant que si des clés sont masquées par d'autres clés identiques, elles subiront aussi la fonction (et on sait que ce sera dans l'ordre inverse de leur apparition dans la table de hachage). [1]

Toutes ces fonctions ne sont pas à connaître mais à comprendre si elles sont rappelées dans un sujet, ce qui (on l'espère) ne manquera pas d'être le cas quand la structure de tableau associatif sera pertinente. **... c'est à dire a priori systématiquement !**

8.6 Sérialisation

Commençons cette section par l'ajout d'une information concernant les paramètres à passer à un programme compilé. On rappelle que lorsqu'on compile un programme, on obtient un exécutable qu'on peut lancer en ligne de commande (de préférence à un clic sur un raccourci).

Ceci étant, plutôt que de faire saisir d'éventuels paramètres rendant le programme interactif, ne serait-ce que pour fournir des valeurs initiales pour un seul calcul potentiellement long, on peut saisir ses paramètres lors de l'exécution en ligne de commande.

Cela donne par exemple avec la syntaxe Unix :

```
./programme_qui_multiplie 6 7
```

Et la console s'empressera d'exécuter ce programme en lui fournissant ces deux arguments, avec pour résultat attendu l'affichage de `42`.

1. Ne pas bricoler la fonction de hachage pendant l'itération !

Comment utiliser ces arguments ? Avec le tableau `argv`, qui contient l'ensemble des éléments de la ligne de commande exécutée, soit `"programme_qui_multiplie"`, `"6"` et `"7"`. **Tous les trois sont des chaînes de caractères**, nous reviendrons sur ce fait.

Ce tableau est disponible en OCaml en écrivant `Sys.argv`, dont la taille permet de déterminer le nombre d'arguments effectivement fournis, dont il faudra éventuellement vérifier qu'il est cohérent à ce qu'il fallait.

En C, ce tableau peut être passé en argument (en tant que `char**`) de la fonction `main`, mais puisqu'on ne peut pas obtenir la taille d'un tableau dans ce langage, un autre argument le précède, nommé `argc` (de type `int`), c'est le nombre total d'éléments dans `argv` en incluant comme annoncé le nom de l'exécutable.

Il est possible de renseigner le contenu de `argv` dans OnlineGDB dans le prompt suivant `Command line arguments:` sans se soucier de la valeur de `argc` en C, qui s'en déduit.

L'utilisation de `argv` est à maîtriser dans les deux langages après rappel, d'après le programme officiel.

Il est cependant recommandé de ne pas avoir besoin de rappels et donc de prendre l'habitude de s'en servir.

Quoi qu'il en soit, revenons sur le fait que les arguments sont forcément récupérés en tant que chaînes de caractères. En soi, rien ne force la machine à devoir considérer que ce qu'elle reçoit a un type plutôt qu'un autre. Qui plus est, de toute manière, en rédigeant le programme, on peut décider soi-même comment la chaîne de caractère en entrée sera utilisée, en particulier convertie s'il le faut.

Le principe de *sérialisation* reprend exactement cette idée : il s'agit de créer une chaîne de caractères représentant un objet, quel qu'il soit, en vue de le stocker dans un fichier de manière efficace, et pourquoi pas réutiliser le fichier en question ultérieurement, par exemple sur une autre machine, ou par un autre programme...

L'opération inverse est appelée *désérialisation*. Elle ne peut se faire indépendamment de la sérialisation, comme un décryptage utilise la connaissance de la clé de cryptage pour appliquer la réciproque.

Une première intuition de la sérialisation est donnée par la fonction `print` de Python, qui fait ce qu'on appelle du *pretty-print*. Elle produit (uniquement pour l'afficher) une représentation « esthétique » de l'argument, mais cette représentation, vue comme une chaîne de caractères, peut se faire évaluer pour récupérer la valeur de ce qui a été imprimé.

Deux objectifs difficilement conciliables peuvent être poursuivis par la sérialisation : la lisibilité de la chaîne produite par un humain ou la concision de l'encodage. C'est en principe ce deuxième objectif qui prime.

En OCaml, la sérialisation (resp. désérialisation) est réalisée par la fonction `to_string` (resp. `from_string`) du module `Marshal`. Au regard de ce qui précède, la chaîne « marshalisée » n'est pas prévue pour une utilisation en tant que telle mais seulement pour la restitution de la valeur qui avait été transformée.

Le module `Marshal` n'est pas à connaître ni maîtriser, et on fera une sérialisation à la main dans le TP 11. Pour les curieux, la fonction `Marshal.to_string` prend un argument supplémentaire en plus de ce qui est à sérialiser, une liste d'options supplémentaires qu'on pourra laisser vide, et la fonction `Marshal.from_string` prend un argument supplémentaire en plus de la chaîne à désérialiser, la position de départ de la lecture de la chaîne, qui est normalement `0`.

Comme le programme le suggère, nous ferons des exemples de sérialisation sur des structures de données présentées dans les chapitres suivants.

Chapitre 9

Décomposition d'un problème en sous-problèmes

Partie du cours : Algorithmique
Prérequis : Récursivité
Applications : Trouver des idées d'algorithmes dans des cas d'application déjà connus ou adapter l'expérience acquise pour trouver des algorithmes originaux
Compétence visée principale : Imaginer et concevoir une solution

---

Il arrive en informatique qu'un algorithme soit amené à traiter une entrée, typiquement une structure de données, en pouvant s'appuyer sur des portions de l'entrée, pour lesquelles le travail est plus facile.

L'intérêt est bien entendu de continuer cette scission des portions en autres portions de plus en plus petites, et ainsi de suite. La gestion de ces portions, avec ou sans chevauchement, en combinant les solutions issues de la considération des portions d'une manière ou d'une autre, donne lieu à des méthodes de programmation classiques que nous détaillerons dans ce chapitre.

Décomposer un problème en sous-problèmes signifie ici a priori que le sous-problème soit le même, mais sur une sous-structure. On pourra cependant songer à la notion d'*analyse descendante*, dont le principe est de séparer un problème en des sous-problèmes tous différents, qui pourraient par exemple être résolus par des fonctions annexes appelées par une fonction principale.

Pour commencer, présentons un exemple de méthode où, malgré la possibilité d'une décomposition, un seul sous-problème est considéré.

9.1 Algorithmes gloutons

Pour certains problèmes en informatique, l'intuition d'une solution grossière, qui présente l'avantage de ne pas avoir besoin d'explorer toutes les possibilités, fournit la solution optimale.

De tels algorithmes qui, dans l'arborescence des calculs pouvant être faits, qu'une récursion soit en jeu ou non, ne considèrent qu'une seule branche, sont appelés des *algorithmes gloutons*.

Un algorithme glouton dispose d'une relation de comparaison sur les choix, de sorte qu'il ne retienne que l'option considérée comme optimale, sans jamais revenir en arrière, à tort ou à raison.

Il s'agit ici de ne pas procéder par exploration exhaustive, méthode traitée dans un autre chapitre.

En plus d'une complexité bien moindre, la facilité d'écriture relative est un intérêt de ces algorithmes.

Un exemple de la vie réelle pour fixer les idées : un déménageur qui doit remplir un camion avec des cartons peut tenter un nombre incommensurable de dispositions, mais sera a priori enclin à commencer par mettre les cartons les plus gros en premier, puis de boucher les trous avec les petits cartons.

Cela ne marchera pas forcément, mais s'il y a assez de marge totale dans le camion on a toutes les raisons d'y croire.

Il est parfois possible de prouver que la réponse obtenue par un algorithme glouton est suffisamment proche de la solution optimale pour qu'on s'en contente, notamment si le calcul de la solution optimale n'est pas possible dans un délai raisonnable. Ce genre de contextes est au programme de la deuxième année.

Au cas par cas, on pourra aussi établir que l'algorithme glouton qu'on a écrit est optimal.

En pratique, la notion hors programme de *matroïde* permet de donner une condition suffisante sur un problème pour admettre un algorithme glouton donnant la solution exacte.

Pour épargner aux curieux la recherche sur un terme, on signale qu'un matroïde est 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 $y \in Y \setminus X$.[1]

Une séance de TP sera consacrée aux algorithmes gloutons optimaux.

9.2 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 **de même nature**, qu'on résout encore une fois en les divisant jusqu'à arriver à des problèmes élémentaires (important !), 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 scinder le travail (avec récursion ou boucles) sur des entrées de taille divisée au lieu de diminuée d'une constante, ce qui limite évidemment de manière drastique le nombre d'appels.

Dans le TD associé à cette section, quelques exemples de problèmes où un algorithme DPR peut se substituer à l'algorithme intuitif, qui est alors moins efficace en termes de complexité, seront abordés.

1. En mettant de côté le caractère fini de S, on peut faire un parallèle avec l'algèbre linéaire, où les éléments de I peuvent être assimilées à des familles libres dans un espace vectoriel.

On rappelle les formules de complexité données dans le chapitre sur la récursivité :

- $c_n = c_{\frac{n}{2}} + \mathcal{O}(1)$: $c_n = \mathcal{O}(\log n)$.
- $c_n = c_{\frac{n}{2}} + \mathcal{O}(n)$: $c_n = \mathcal{O}(n)$.
- $c_n = 2c_{\frac{n}{2}} + \mathcal{O}(1)$: $c_n = \mathcal{O}(n)$.
- $c_n = 2c_{\frac{n}{2}} + \mathcal{O}(n)$: $c_n = \mathcal{O}(n \log n)$.
- $c_n = ac_{\frac{n}{b}}$, a et b étant des entiers strictement positifs : $c_n = \Theta(n^{\log_b a})$.

On aura reconnu la recherche dichotomique dans la première formule et le tri fusion (ou un tri rapide optimisé) dans la quatrième.

En pratique, un tri peut être la partie la plus coûteuse d'un algorithme, permettant de résoudre par la suite un problème en temps linéaire alors qu'une méthode naïve aurait par exemple été quadratique. Un tel algorithme sera qualifié de DPR à condition que le tri permette tout de même une séparation intelligente des données.

Les formules ci-avant découlent en fait toutes d'un théorème hors programme qu'on abordera dans le même TD, reporté ici.

Théorème

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

Ce théorème 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. En outre, remplacer Θ par $\mathcal{O}$ dans la relation et le résultat est possible.

9.3 Programmation dynamique

Remarque : Les programmes données en exemple dans cette section sont en OCaml vu le parallèle avec la récursivité, mais rien n'empêche de faire de la programmation dynamique en C.

Nous avons déjà signalé que la pile d'appels d'un programme récursif peut engendrer un coût en espace que l'utilisation de boucles ne connaît pas forcément (on peut certes 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).

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 !";
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 :

```
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;;
```

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 $0 \leq k \leq n$, 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 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 tab.(i)
      else (let v = aux(i-2) + aux(i-1) in tab.(i) <- v; v)
    in aux n; tab.(n);;
```

Chapitre 10

Induction

Partie du cours : Récursivité et induction
Prérequis : Récursivité
Applications : Structures de données, logique, langages formels (SPE)
Compétence visée principale : Justifier et critiquer une solution

---

Dans certains cas, les preuves de programmes agissant sur des structures, voire de propriétés qu'ont les structures en question, permettent péniblement d'écrire une récurrence. Il s'agira généralement de récurrences fortes pour lesquelles il faut tenir compte de toutes les possibilités de tailles de sous-structures, et d'autres difficultés peuvent encore survenir.

Pour échapper à ces problèmes, ce chapitre propose un principe de preuve qui généralise toutes les récurrences et qui pourra s'utiliser à leur place autant qu'on le souhaite : l'induction structurelle.

Afin de justifier son fonctionnement, un petit détour par la théorie des ensembles, dont certains éléments sont malheureusement absents du programme de mathématiques, s'impose.

10.1 Prérequis mathématiques

Commençons par des rappels de notions déjà vues en mathématiques.

Définition

Une relation binaire entre deux ensembles E_1 et E_2 (possiblement égaux) est un sous-ensemble, noté R, du produit cartésien $E_1 \times E_2$. Un couple $(x_1, x_2) \in E_1 \times E_2$ est dit en relation par R, ce qu'on note $x_1 R x_2$ pour simplifier, s'il appartient à R.

Cette idée de relations sera encore étendue dans le chapitre sur les bases de données... relationnelles, justement.

Définition

Une relation binaire R entre un ensemble E et lui-même est un ordre (on dit aussi une relation d'ordre) sur E si R est réflexive, antisymétrique et transitive. On dit alors que (E, R) est un ensemble ordonné.

La réflexivité signifie que pour tout élément x de E, on a xRx, l'antisymétrie signifie que pour tous éléments x et y de E, la conjonction xRy ET yRx implique que x et y soient un seul et même élément, et la transitivité signifie que pour tous éléments x, y et z de E tels que xRy ET yRz, on ait xRz.

Retirer tous les couples (x, x) pour $x \in E$ à une relation d'ordre donne une relation d'ordre stricte.

On notera que, tout comme un groupe est la donnée d'un ensemble avec sa loi de composition interne, il ne peut pas manquer à un ensemble ordonné la mention de son ordre.

Définition

Soient (E_1, R_1) et (E_2, R_2) deux ensembles ordonnés. La relation R entre $E_1 \times E_2$ et lui-même telle que $(x_1, x_2) R (y_1, y_2)$ si, et seulement si, $x_1 R_1 y_1$ ET $x_2 R_2 y_2$ est appelée ordre produit sur $E_1 \times E_2$.

Définition

Soient (E_1, R_1) et (E_2, R_2) deux ensembles ordonnés. On note $\overline{R_1}$ l'ordre strict correspondant à R_1. La relation R entre $E_1 \times E_2$ et lui-même telle que $(x_1, x_2)R(y_1, y_2)$ si, et seulement si, $x_1\overline{R_1}y_1$ OU ($x_1 = y_1$ ET $x_2R_2y_2$) est appelée ordre lexicographique sur $E_1 \times E_2$.

Les ordres produits et lexicographiques s'étendent à un produit cartésien de plus de deux ensembles. On peut montrer aisément qu'il s'agit de relations d'ordre.

Dans les ordres produits et lexicographiques, on ne rappelle pas l'ordre sur les ensembles de base. On les sous-entend même s'il s'agit des ordres usuels.

Définition

Une relation d'ordre R sur un ensemble E est totale si pour tous x et y dans E on a xRy OU yRx. Sinon, on dit que la relation d'ordre est partielle. L'ensemble lui-même est alors qualifié d'ensemble totalement ordonné ou ensemble partiellement ordonné suivant le cas.

La relation $\leq$ sur les ensembles de nombres usuels est totale, ainsi que l'ordre lexicographique sur des n-uplets d'éléments issus d'ensembles munis d'un ordre total. La relation d'inclusion entre des parties d'un ensemble ne l'est pas, et la relation de divisibilité sur $\mathbb{N}$ non plus (en signalant que ce n'est même pas une relation d'ordre sur $\mathbb{Z}$ faute d'antisymétrie).

Définition

Soit (E, R) un ensemble ordonné. Soient $x, y \in E$. Alors x est un prédécesseur de y si xRy et x est différent de y, et dans ce cas y est un successeur de x. De plus, x est un prédécesseur immédiat de y s'il n'existe pas de successeur de x qui soit un prédécesseur de y, et dans ce cas y est un successeur immédiat de x.

Définition

Soit (E, R) un ensemble ordonné. Un élément minimal de E est un élément de E qui n'a pas de prédécesseur, et un plus petit élément de E est un élément dont tous les autres éléments sont des successeurs. De manière analogue, un élément maximal de E est un élément de E qui n'a pas de successeur, et un plus grand élément de E est un élément dont tous les autres éléments sont des prédécesseurs.

Dans $\mathbb{N}$ muni de la relation de divisibilité (ordre partiel), 1 est le plus petit élément et 0 est le plus grand élément. Si on exclut 1 de cet ensemble, tous les nombres premiers deviennent des éléments minimaux mais il n'y a en particulier plus de plus petit élément, si l'on exclut 0 de cet ensemble il n'existe même plus d'élément maximal.

On remarquera que le plus petit élément, s'il existe, est un élément minimal, et il est alors unique.

Il peut arriver qu'un ensemble ordonné possède un élément minimal unique et que celui-ci ne soit pas le plus petit élément, mais ceci n'est pas possible quand l'ordre est total.

Un exemple artificiel est de considérer $\mathbb{Z} \cup \{\perp\}$ muni de l'ordre usuel sur $\mathbb{Z}$ et de considérer que $\perp$ n'est en relation avec aucun élément. Alors $\perp$ est le seul élément minimal (et maximal), mais ce n'est pas le plus petit élément ni le plus grand élément.

Il existe par ailleurs aussi des ensembles totalement ordonnés n'admettant pas de plus petit élément ni de plus grand élément, $\mathbb{Z}$ muni de l'ordre usuel en est un exemple.

Définition

Un ensemble bien ordonné est un ensemble muni d'un ordre bien fondé (aussi appelé bon ordre), c'est-à-dire qui n'admet pas de suite infinie strictement décroissante pour cet ordre.

Une expression s'évaluant en un élément d'un ensemble bien ordonné (typiquement $\mathbb{N}$ muni de l'ordre usuel) peut servir de variant pour une boucle conditionnelle. Ainsi, il n'est pas nécessaire de transformer une telle expression en un entier naturel par des calculs compliqués. En effet, l'ordre lexicographique sur des n-uplets est bien fondé, quel que soit le nombre fixé d'éléments.

Proposition

Un ensemble est bien ordonné si, et seulement si, toute partie non vide de cet ensemble admet un élément minimal.

La preuve de cette proposition, dans les deux sens de l'équivalence, est un exercice de mathématiques laissé aux lecteurs.

Proposition

L'ordre produit et l'ordre lexicographique sur un produit cartésien d'ensembles bien ordonnés est lui-même bien fondé.

La preuve se fait par récurrence sur le nombre d'ensembles concernés, l'hérédité dépendant de l'une ou l'autre des propriétés d'un ensemble bien ordonné suivant le cas.

10.2 Ensemble inductif

Les définitions dans ce chapitre rappellent de nombreuses structures de données déjà vues à ce stade de l'année, parfois de manière non intuitive, et les notions en question se retrouveront encore dans des exemples ultérieurs (par exemple dans la partie sur la logique).

Définition

Un ensemble inductif est le plus petit ensemble engendré à partir de valeurs de base grâce à des régles appelées règles d'inférence.

Concrètement, il s'agit de construire E à partir d'un ensemble E_0, fini ou infini, et de constructeurs réalisant les règles, ces constructeurs étant notés $C_1, \ldots, C_n$, chaque C_i étant paramétré par un nombre d'arguments entier supérieur ou égal à un (dont au moins un, disons d_i, qui soit dans E, et un certain nombre potentiellement nul, disons a_i, qui sont dans d'autres ensembles) nommé l'*arité de la règle* (comme pour les arbres), de sorte que si $x_1, \ldots, x_{d_i} \in E$ sont des éléments déjà construits et $y_1, \ldots, y_{a_i}$ sont des valeurs issues d'ensembles adéquats, alors $C_i(y_1, \ldots, y_{a_i}, x_1, \ldots, x_{d_i})$ s'obtient comme nouvel élément pour E.

On peut alors considérer que pour tout $k \in \mathbb{N}$, E_{k+1} est la réunion de E_k et de tous les nouveaux éléments obtenus en utilisant un C_i sur le nombre correspondant d'éléments de E_k en plus des arguments extérieurs.

La réunion sur $\mathbb{N}$ de ces E_k forme alors l'ensemble E. Il s'agit de ce qu'on appelle « définition par le bas ».

L'ensemble E a également la propriété d'être le plus petit ensemble contenant E_0 et stable par l'application des règles C_i, qui est également l'intersection de tous les ensembles contenant E_0 et stables par l'application des règles C_i. On appelle ceci « définition par le haut ».

Nous pouvons alors donner ici des exemples d'ensembles inductifs :

— L'ensemble des entiers naturels, avec $E_0 = \{0\}$ et l'unique règle est « successeur », représentant $x + 1$ pour tout entier x, notée S dans l'axiomatique de Peano.

— L'ensemble des listes chaînées, avec E_0 réduit à la liste vide et la règle « conse », d'arité deux avec un argument extérieur du type des données à stocker.

— L'ensemble des expressions arithmétiques, avec E_0, infini, contenant tous les nombres qu'on souhaite utiliser, et une règle par opérateur arithmétique disponible.

— L'ensemble des arbres binaires, celui des arbres... plus généralement, tout ce qui peut correspondre à un type somme en OCaml, avec E_0 représentant tout ce qui peut s'obtenir par les constructeurs non récursifs et les règles simulées par les constructeurs récursifs.

On remarquera que dès que E_0 est non vide et qu'il y a au moins une règle, l'ensemble inductif obtenu est infini.

Définition

Soit E un ensemble inductif. On pose R la relation entre E et lui-même de sorte que xRy si, et seulement si, y s'obtient à partir d'un constructeur dont x est un des arguments. L'ordre structurel sur E est la relation $\tilde{R}$ telle que $x\tilde{R}y$ si, et seulement si, il existe une séquence finie $x_0, \ldots, x_n$ telle que $x_0 = x$, $x_n = y$ et pour tout $0 \leq i < n$, on ait $x_i R x_{i+1}$.

Le fait que n puisse valoir 0 garantit la réflexivité. La transitivité vient en fait que $\tilde{R}$ est déjà ce qu'on appelle *clôture transitive* de R, et on peut se convaincre de l'antisymétrie en observant que dans la définition par le bas de E, quand xRy (où on utilise bien la relation de base ici), c'est que x est dans un ensemble E_k et y dans un ensemble E_l tel que $l \geq k + 1$ (potentiellement plus car il peut y avoir d'autres arguments). Il est alors impossible que $x\tilde{R}y$ en même temps que $y\tilde{R}x$ sans avoir $x = y$,

Proposition

L'ordre structurel est bien fondé.

C'est encore la définition par le bas d'un ensemble inductif qui permet de prouver cette propriété.

Corollaire

Toute fonction récursive sur une structure pour laquelle les appels récursifs ont uniquement pour arguments des éléments strictement inférieurs (qu'il y en ait un ou plusieurs, les mêmes ou non) à l'argument de l'appel en cours, ceci selon l'ordre structurel, termine.

Voilà pour le fondement mathématique des preuves de terminaison des fonctions récursives. La section suivante traite des preuves de correction. Quant au calcul de complexité, on continuera de le faire en établissant une formule de récurrence à partir des tailles des arguments.

10.3 Induction structurelle

Cette section est très courte, mais lourde de conséquence.

Proposition

Soit E un ensemble inductif défini par E_0 et un ensemble de règles $C_1, \ldots, C_n$. Soit une propriété vraie sur E_0 et demeurant vraie sur les éléments produits par une règle C_i à partir d'arguments dans E pour lesquels la propriété est supposée vraie. Alors la propriété en question est vraie pour tout élément de E. On appelle ce principe l'induction structurelle.

La preuve repose sur la minimalité de l'ensemble inductif parmi les ensembles contenant E_0 et stables par les règles. L'ensemble formé de tout ce qui rend la propriété vraie en faisant partie, on peut conclure.

La propriété mentionnée peut s'énoncer comme un théorème parlant d'une structure ou comme le fait qu'une fonction récursive prenant en argument une instance d'une structure respecte sa spécification.

Par exemple, on montrera au chapitre 11 par induction structurelle que tout arbre binaire strict a une feuille de plus que son nombre de nœuds internes.

En termes de fonctions, on peut aussi montrer que la fonction récursive qu'on écrira aisément pour déterminer la taille d'un arbre binaire est correcte.

L'induction structurelle remplacera désormais de manière agréable les récurrences fortes sur la taille d'une structure.

Chapitre 11

Structures de données hiérarchiques

Partie du cours : Structures de données
Prérequis : Les deux chapitres précédents de la partie ainsi que l'induction
Applications : Programmes avancés, et tout ce qui tourne autour des arborescences
Compétence visée principale : Analyser et modéliser

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 distinctes ? Ce branchement donne, comme le vocabulaire le laisse deviner... un *arbre*.

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

Le titre de ce chapitre fait référence au fait que les données sont organisées de sorte que chacune soit associée à une autre (sauf la racine, déjà évoquée) plus haut qu'elle dans l'arborescence dessinée, en tant que son supérieur direct.

En cela, on peut se rapprocher de beaucoup de situations de la vie courante.

De nombreux types d'arbres, avec des propriétés particulières, ont été inventés pour répondre à des besoins de complexité ou d'expressivité.

Seul un petit nombre d'entre eux sont au programme, mais beaucoup sont en fait relativement faciles à comprendre, pouvant faire l'objet d'un sujet de concours ultérieur ou être découverts en parcourant la littérature pendant des recherches pour le TIPE...

11.1 Arbres

Introduction et vocabulaire

Commençons par du vocabulaire : 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 (comme nous le verrons dans les deux définitions des types dans la section suivante), 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 qui n'est pas une feuille et qu'on appelle *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, 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.

On retrouvera la notion de connexité plus en détail dans le chapitre sur les graphes.

Une *branche* est une suite de nœuds telle que chaque nœud soit un fils du précédent, certaines 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, sachant que la dernière est la plus répandue) 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.

Afin d'uniformiser les définitions, le programme suggère que la hauteur d'un arbre vide soit -1. Il faut donc en déduire que la profondeur de la racine est 0, valant donc la hauteur d'un arbre de taille 1.

Pour information, il n'est pas exclu de tomber à l'occasion sur une notion de profondeur d'arbre ou de hauteur d'un nœud, dans ce cas tout sera expliqué pour fixer une convention et éviter les confusions faute de précisions suffisantes.

Les nœuds des arbres véhiculent la plupart du temps une information, afin que la structure ait un intérêt. On appelle *étiquette* l'information associée à un nœud.

En C et en OCaml, les étiquettes auront un certain type. Par ailleurs, dans certains cas [1], il est utile que les nœuds internes et les feuilles véhiculent une information différente, ce qui peut même amener à ce que le type des étiquettes soit différent entre ces deux types de nœuds.

On peut alors définir les arbres en OCaml de la façon suivante (où il est possible mais pas forcément recommandé d'ajouter le constructeur `Vide`, en notant que son absence ci-après fait que l'arbre vide n'est pas représentable) :

```
type ('a,'b) arbre =
Feuille of 'a | Noeud_interne of 'b * ('a,'b) arbre list;;
```

Pour éviter d'avoir une redondance entre les feuilles et les nœuds internes sans fils, on pourra imposer au niveau des programmes sur ce type que les listes soient non vides, mais mieux vaut éviter de l'imposer structurellement (par exemple en isolant le premier fils puis en mettant les autres dans une liste). De même, l'absence d'un constructeur `Feuille` serait malvenue, quand bien même les types `'a` et `'b` seraient les mêmes. On veillera néanmoins à rester flexible si un énoncé propose un type différent.

En C, une définition possible revient à donner tous les enfants dans un tableau, dont le nombre est précisé comme d'habitude, en fournissant si deux types différents sont nécessaires deux informations dont une sera à ignorer. Pour ne pas avoir à inventer une valeur à ignorer, on peut remplacer les informations par des pointeurs vers de la mémoire non initialisée.

1. voir par exemple le codage de Huffman, étudié plus tard en TP

Un exemple (le polymorphisme n'étant pas prévu par le programme, on donnera un cas particulier avec des booléens aux feuilles et des chaînes aux nœuds internes) :

```
struct s_b_a { char etiq_n[64]; bool etiq_f;
int nb_fils; struct s_b_a* fils; };

typedef struct s_b_a asb;
```

Applications

Cette section présente quelques-unes des applications classiques de la structure d'arbre.

Expressions arithmétiques

Lorsqu'on exécute un programme, la première étape est ce qu'on appelle l'analyse lexicale, suivie de l'analyse syntaxique.

Ces notions seront éventuellement développées en guise d'ouverture en deuxième année, et il suffit pour le moment de comprendre que le résultat de ces analyses est la détection d'une erreur de syntaxe, s'il y en a une, ou la construction de ce que l'on appelle l'arbre de syntaxe abstraite, qui va être évalué par l'intepréteur ou le compilateur.

Ceci s'applique aussi de manière restreinte aux calculs dans le cadre des expressions arithmétiques.

La construction de l'arbre tient compte des priorités opératoires, et nous observons ici un cas particulier de distinction au niveau des étiquettes, puisque les nœuds internes, ayant au moins un fils, correspondront aux opérateurs et leurs fils aux opérandes, les feuilles étant alors les constantes.

L'expression $2 + 3 * 5$ sera alors un arbre dont la racine est étiquetée par le symbole d'addition et a pour fils une feuille d'étiquette 2 et un nœud interne d'étiquette le symbole de multiplication, ce nœud interne ayant pour fils deux feuilles d'étiquettes respectives 3 et 5.

Le résultat du calcul s'obtient alors récursivement, en évaluant chaque nœud interne à partir de la valeur de ses fils et de l'opérateur dans l'étiquette.

Trie

Un trie, aussi appelée arbre préfixe, est un arbre dont les nœuds sont étiquetés par des chaînes de caractères, avec la contrainte que l'étiquette de la racine soit la chaîne vide et que les étiquettes de tous les fils d'un même nœud soient différentes deux à deux et correspondent à l'étiquette du nœud en question à laquelle on a ajouté un caractère à la fin.

Cette structure s'utilise notamment pour encoder un ensemble de mots, puis par extension un tableau associatif indexé par les chaînes de caractères, sachant que certains nœuds ont potentiellement besoin d'être invalidés dans ce cas, ne servant qu'à progresser vers une entrée effectivement dans le dictionnaire.

Une extension optimisée du trie est l'arbre dit "PATRICIA" (acronyme anglais).

Arbre de décision

Le modèle des arbres de décision est régulièrement rencontré dans la vie courante, en tant que restriction des organigrammes ("flowcharts") où la réponse à diverses questions successives mène à un résultat. Ici, contrairement aux organigrammes usuels, il n'est pas question de revenir à un endroit déjà visité, et les questions seront toujours fermées.

On observe que dans ce cas les liaisons entre chaque nœud et ses enfants sont étiquetées par un booléen.

Pour un sujet de concours portant sur les arbres de décision, on pourra consulter Centrale 2013 (à cette époque, la seule épreuve d'informatique était celle d'option en MP).

Dans une version ultérieure de ces notes, une figure donnant un exemple sera ajoutée...

11.2 Arbres binaires

Introduction

Les arbres les plus classiques sont les *arbres binaires*, c'est-à-dire dont chaque nœud a au plus deux fils.

Ici, le type retenu va ignorer la notion de feuille et reformuler la définition en « chaque nœud a exactement deux fils, pouvant être des arbres vides ». On en déduit le type suivant en OCaml :

```
type 'a arbre_bin = V | N of 'a arbre_bin * 'a * 'a arbre_bin;;
(* Rappel : un constructeur ne peut être utilisé qu'une fois,
et on compte utiliser ce type avec celui de la section précédente. *)
```

En C, les étiquettes sont d'un type à préciser, disons par exemple qu'on prend des entiers ici :

```
struct i_a_b { int etiquette;
struct i_a_b* fils_gauche; struct i_a_b* fils_droit; };

typedef struct i_a_b abi;
```

Mentionnons ici quelques arbres binaires particuliers.

Un *peigne droit* (gauche analogue) est un arbre binaire dont le fils gauche de tous les nœuds est l'arbre vide. [2]

Un peigne gauche ou droit est donc un cas particulier d'arbre réduit à une seule branche maximale.

Un *arbre binaire complet* est un arbre binaire de hauteur notée n, avec $n \in \mathbb{N}$, dont tous les 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.

Un *arbre binaire presque complet* est un arbre binaire complet dont il manque des nœuds au niveau de profondeur maximal, la plupart des conventions imposant que ces nœuds manquant soient « le plus à droite possible ».

2. 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. Un sujet de concours avait par exemple une définition différente de celle donnée ici.

Propriétés de base

Proposition

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

La preuve se fait par une récurrence simple : le nombre de nœuds de profondeur k pour $0 \leq k \leq n$ est 2^k car il y a deux fils par nœud à la profondeur $k-1$ (si $k > 0$) et un seul nœud à la profondeur 0 : 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 (voir chapitre 10 pour la notion d'induction) : 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 $x_g + 1 + x_d + 1$.

Représentation d'un arbre binaire complet dans un tableau

À la suite de la première proposition de la section précédente, il s'avère qu'un arbre binaire complet (mais en pratique aussi un arbre binaire presque complet) a une propriété permettant de le simuler dans un tableau : en commençant par placer la racine à l'indice zéro, il est possible de placer par la suite les enfants de tout nœud déjà place à un indice noté `i`, en l'occurrence le fils gauche à l'indice `2*i+1` et le fils droit à l'indice `2*i+2`.

Ceci garantit d'une part qu'il n'y a jamais deux nœuds mis au même indice en suivant cette formule (la propriété resterait vraie pour tout arbre binaire), et d'autre part qu'aucun trou n'est laissé (dans ce cas précis, on a besoin que l'arbre soit presque complet). Le principe se retrouve dans la numérotation Sosa en généalogie.

Bien entendu, la structure de tableau est alors bien plus maniable, et l'on peut même avoir un accès direct à tout nœud en faisant intervenir un calcul en binaire.

11.3 Arbres vs. arbres binaires

En pratique, le programme suggère la présentation des arbres binaires avant les arbres. **En fait, la structure d'arbre binaire n'est pas un cas particulier de la structure d'arbre, au vu de la façon dont on utilise ces deux structures.**

Dans la section abordant ces derniers, la conversion des arbres d'arité quelconque vers les arbres binaires est à présenter, ce qui va être fait ci-après. Par ailleurs, les applications ayant déjà été mentionnées, ce sera tout pour ce point.

Le principe de cette conversion est le suivant : puisque les enfants d'un nœud sont stockés en séquence, on peut imaginer que cette séquence est en fait une liste chaînée. Alors dans ce cas, chacun de ses éléments sauf le dernier a un pointeur vers le suivant, mais aussi un pointeur vers la liste chaînée de ses enfants. Cela fait donc deux pointeurs, comme un arbre binaire !

Ainsi, l'arbre binaire correspondant à un arbre d'arité quelconque verra tout nœud N de l'arbre de départ remplacé par un nœud dont le fils gauche est le premier fils de N s'il en avait et le fils droit est le « frère » suivant de N s'il en avait. En particulier, la racine de cet arbre ne peut pas avoir de fils droit (ce qui laisse la possibilité d'étendre la conversion à une liste d'arbres, en fait !).

Cette transformation peut très facilement s'inverser pour passer d'un arbre binaire dont la racine n'a pas de fils droit à un arbre d'arité quelconque.

On comprendra aisément l'intuition que la hauteur de l'arbre ainsi obtenu a tendance à être supérieure à celle de l'arbre de départ.

De manière amusante, un arbre d'arité 2 subissant la transformation donnera généralement un arbre binaire de forme différente.

11.4 Parcours d'arbres

La notion de parcours sera abordée dans le chapitre sur les graphes, il suffit de comprendre la base dans le cas des arbres :

— Parcours en largeur : les frères sont visités avant les fils. En quelque sorte, on pourrait considérer qu'on parcourt la descendance de quelqu'un par âge décroissant en imaginant une certaine régularité dans les naissances.

— Parcours en profondeur : les fils sont visités avant les frères. Un exemple classique est l'ordre de succession au trône dans la plupart des monarchies.

Il y a en pratique deux variantes du parcours en profondeur, plus une troisième dans le cas spécifique des arbres binaires :

— parcours préfixe : la visite d'un nœud précède celle de ses descendants (véritable ordre de succession) ;

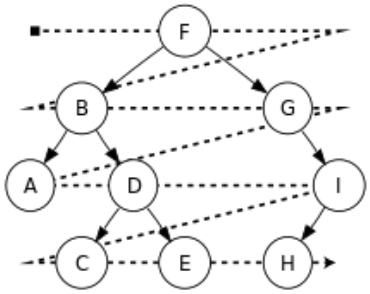
— parcours postfixe (parfois nommé suffixe) : la visite d'un nœud se fait uniquement après celle de tous ses descendants ;

— parcours infixe : la visite d'un nœud se fait après celle de son fils gauche et de ses descendants mais avant celle de son fils droit et de ses descendants.

Pour clarifier ces notions, rien ne vaut un dessin[3], en l'occurrence deux dessins.

Dans la figure ci-après, le parcours en largeur est présenté. Signalons à la lumière de cet exemple que l'organisation par niveaux de la représentation d'un arbre simplifie grandement les choses.

Le parcours en largeur de l'arbre en question traitera donc dans l'ordre `F`, `B`, `G`, `A`, `D`, `I`, `C`, `E` et `H`.



Dans la figure ci-après, les trois versions du parcours en profondeur sont matérialisées

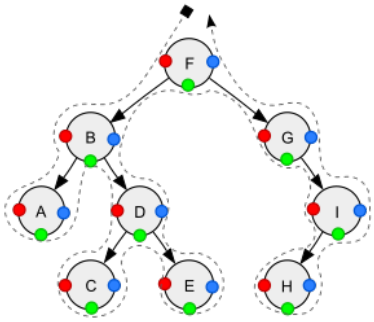
3. ... surtout quand on peut le récupérer depuis Wikipédia sans avoir à le recréer !

par l'utilisation de cercles rouges (à gauche), verts (en bas) et bleus (à droite).

Comme on peut le constater, le parcours en profondeur est unique : il s'agit de l'itinéraire fléché.

Ce qui change est le moment où un nœud est relevé : soit au passage du cercle rouge qui lui est associé, dans le cadre du parcours préfixe, soit au passage du cercle vert, dans le cadre du parcours infixe, soit au passage du cercle bleu, dans le cadre du parcours postfixe.

On peut aussi voir cela comme la première, la deuxième et la troisième visite, en notant que les trois coïncident pour une feuille, et deux coïncident pour un nœud interne ayant exactement un fils vide.



Ainsi, le parcours en profondeur préfixe traitera dans l'ordre `F`, `B`, `A`, `D`, `C`, `E`, `G`, `I` et `H`, le parcours en profondeur infixe traitera dans l'ordre les nœuds dans l'ordre alphabétique (nous reviendrons sur ceci à la section suivante...), et le parcours en profondeur postfixe traitera dans l'ordre `A`, `C`, `E`, `D`, `B`, `H`, `I`, `G` et `F`.

Il est intéressant de noter que l'écriture de fonctions récursives effectuant les parcours des arbres permet, de manière duale, de comprendre comment l'arbre d'exécution associé à une fonction récursive empile les appels récursifs et les blocs d'activation (voir chapitre 5 pour la notion).

11.5 Arbres binaires de recherche

Définition et propriétés

Définition

Un arbre binaire de recherche (noté désormais ABR) est un arbre binaire dont les nœuds sont étiquetés de sorte que toutes les étiquettes (ou « clés ») du sous-arbre gauche d'un nœud `x` *soient inférieures (ou égales) à l'étiquette de* `x` *et toutes les étiquettes du sous-arbre droit d'un nœud* `x` *soient supérieures (ou égales) à l'étiquette de* `x`*.*

Proposition

Un parcours en profondeur infixe d'un ABR donne la séquence triée dans l'ordre croissant des étiquettes des nœuds de l'arbre.

La preuve se fait par induction.

Cas de base :
C'est vrai pour un ABR vide (ou réduit à sa racine, suivant les constructeurs constants utilisés).

Induction :
Si c'est vrai pour le sous-arbre gauche et le sous-arbre droit de la racine, alors le parcours infixe traite d'abord tout le sous-arbre gauche (donnant la séquence triée dans l'ordre croissant de ses étiquettes), puis poursuit la séquence par la racine, qui est par définition supérieure à toutes les étiquettes du sous-arbre gauche mais inférieure à toutes les étiquettes du sous-arbre droit, puis traite celui-ci (terminant la séquence par les étiquettes du sous-arbre droit triée dans l'ordre croissant).

Le fait que l'ensemble des étiquettes de l'arbre soit ainsi couvert (car celles-ci sont soit la racine soit dans un sous-arbre), certes si intuitif qu'on se dispense généralement de le souligner, permet de conclure.

La propriété caractérisant un ABR permet de faire des recherches à bas coût [4].

4. des recherches de pétrole, donc !

Les trois opérations élémentaires (en plus de la création) sur les ABR sont :

— La recherche parmi les clés : l'élément recherché est-il la clé d'un nœud, à chercher dans son sous-arbre gauche ou dans son sous-arbre droit ?

— L'insertion d'une clé : il y a plusieurs méthodes possibles, la plus simple revient à ajouter le nœud au fond de l'arbre en suivant un chemin respectant la structure d'ABR ; si la clé existe déjà on décide d'une manière ou d'une autre.

— La suppression d'une clé : si elle se fait à un nœud dont aucun sous-arbre n'est vide, il faut chercher dans un sous-arbre quel nœud remonter, une solution simple étant l'élément maximal du sous-arbre gauche, en prenant garde à ne pas rendre les opérations trop compliquées.

On notera que la complexité de ces trois opérations est au pire des cas la profondeur de l'arbre, car on explore toujours une seule branche.

Théorème

[Admis] La profondeur d'un ABR de taille n est au pire des cas n (dans le cas d'un peigne, par exemple) et en moyenne un $\Theta(\log n)$.

En ce qui concerne l'implémentation, la propriété sur les éléments n'est pas liée à la syntaxe. Il s'agit d'un invariant de structure que les opérations élémentaires doivent préserver. On utilisera donc un type d'arbre binaire tel quel dans les deux langages mais avec de nouvelles opérations élémentaires.

Implémentation d'un tableau associatif

Il s'avère que, pour implémenter un dictionnaire, un ABR est un meilleur choix que les versions naïves reposant sur des tableaux, qu'ils soient triés ou non. Il résout aussi les difficultés que posent les tables de hachage.

Les clés du dictionnaire, a priori des chaînes de caractère, sont les étiquettes des nœuds, et comme on l'a vu, la recherche d'une clé se fait en temps logarithmique en moyenne, de même que l'insertion et la suppression.

Une contrepartie à l'utilisation d'un ABR est que la localisation d'une clé se fait par des comparaisons à chaque nœud, nécessitant que la comparaison en question puisse être faite et exploitée, donc que les clés appartiennent toutes au même ensemble muni d'un ordre total.

Le fait d'avoir un temps logarithmique en moyenne n'est pas assez puissant par rapport au constant en moyenne d'une table de hachage avec une capacité suffisante, nous allons donc améliorer ceci en renforçant les ABR par une propriété supplémentaire, qui permettra d'avoir un temps logarithmique dans le pire des cas. C'est l'objet de la sous-section suivante.

Arbres rouge-noir

Le but de cette section est de présenter une structure particulière d'ABR dont la hauteur est de l'ordre du logarithme de la taille. Certains auteurs considèrent que cette propriété est la définition même d'un ensemble d'*arbres binaires équilibrés*.

Pour d'autres, la propriété d'équilibre est plus contraignante et chaque nœud doit avoir un fils gauche et un fils droit de profondeur égale ou différente d'au plus un. Les arbres AVL, qui ne sont pas au programme, ont par exemple cette propriété, qui implique à plus forte raison la propriété liant hauteur et taille.

Dans l'idée, plus on se rapproche d'un arbre (presque) complet, mieux c'est !

Définition

Un arbre bicolore (ou arbre rouge-noir) est un ABR dont les nœuds sont associés à une information binaire (pour l'aspect graphique, il a été décidé qu'on les appellerait des nœuds rouges ou des nœuds noirs) ayant deux propriétés : d'une part le père d'un nœud rouge doit être noir, et d'autre part le nombre de nœuds noirs de toute branche maximale (c'est-à-dire partant de la racine et finissant sur un arbre vide) est une constante sur l'ensemble de l'arbre (appelée hauteur noire).

Proposition

Considérons un arbre bicolore dont la taille est notée n, la hauteur est notée h et la hauteur noire est notée k. Alors $h \leq 2k$ et $n \geq 2^k - 1$.

Corollaire

La hauteur d'un arbre bicolore est dominée par un logarithme de sa taille en utilisant une constante valable pour tous les arbres bicolores.

La preuve de la proposition repose sur une induction structurelle. Elle sera faite en TD, de même que la preuve qu'une différence d'un entre les hauteurs de deux fils implique une hauteur logarithmique en la taille.

Le type associé pour un arbre bicolore est le même que pour les arbres binaires en OCaml, mais les opérations élémentaires feront comprendre au moment du typage qu'on n'aura pas simplement un `'a arbre_bin` mais un `('a * bool) arbre_bin`. En C, cela revient à ajouter un champ booléen à chaque nœud.

Les opérations élémentaires sur les arbres binaires sont à adapter, en raison de l'ajout d'un invariant de structure difficile à maintenir. En particulier, la suppression devient une opération très difficile à mettre en œuvre, et l'implémentation de la structure occupera une grande partie du TP associé, où les explications figureront.

Une étape cruciale dans la réalisation des arbres bicolores est de procéder à des rotations, opérations préservant l'invariant de structure des ABR.

Tri ABR

À la lumière des propriétés des ABR, un algorithme de tri possible revient à construire un ABR par insertions successives des éléments de la séquence à trier, puis à construire la version triée par un parcours en profondeur infixe.

La complexité du parcours est linéaire en la taille de l'arbre donc de la séquence (taille notée n), et le coût de construction est n fois le coût de l'insertion, celui-ci étant la hauteur de l'arbre au moment de l'insertion.

Le total est alors un $\Theta(n \log n)$ si notre ABR est équilibré, suggérant d'optimiser la structure ou de faire confiance à la chance pour éviter les peignes.

L'astuce classique de mélanger avant de commencer, évoquée pour le tri rapide, peut être mise en œuvre ici aussi.

11.6 Tas

Définition et propriétés

Définition

Un tas est un arbre dont les nœuds sont étiquetés par des clés, avec la propriété que la clé d'un nœud est supérieure à la clé de chacun de ses fils.

Cette définition a plusieurs conséquences.

La principale est que la clé maximale d'un tas est à la racine.

Par ailleurs, la deuxième plus grande clé d'un tas est au niveau d'un fils de la racine et au moins un exemplaire de la clé minimale est sur une feuille.

Il faut aussi noter qu'une recherche ne peut pas exclure a priori de branche tant que la valeur recherchée est inférieure à l'étiquette de la racine de la branche en question.

En particulier, on ne peut pas localiser les autres valeurs que le maximum, et la recherche d'une clé ne figurera pas parmi les opérations élémentaires pour cette raison.

Une variante de la structure consiste à remplacer dans la définition le mot « supérieure » par « inférieure ».

On a alors la notion de « tas-min », par opposition aux « tas-max ».

Pour terminer, la plupart du temps, on ajoute comme contraintes que l'arbre soit binaire (on parle alors de tas binaire) et presque complet.

Dans ce cas, **on peut représenter un tas par un simple tableau** comme on l'a vu pour les arbres binaires.

En pratique, ne pas imposer d'arité maximale à un tas ruinerait la structure, car on pourrait simplement maintenir la propriété de tas en ayant toutes les étiquettes sauf le maximum directement sous la racine.

Les opérations élémentaires usuelles sur les tas sont :

— Le tamisage, consistant à faire descendre une racine n'étant pas le maximum d'un tas jusqu'à la place qui lui revient. Il s'agit de faire des échanges (en maintenant donc l'éventuel caractère presque complet de départ) de manière récursive : tant qu'on n'a pas une structure de tas, on échange la racine avec son fils de clé maximale et on tamise le sous-arbre qu'on vient de modifier.

— L'extraction, consistant à supprimer la racine, à récupérer la clé en dernière position dans le parcours en largeur et à tamiser.[5]

— La suppression d'une clé quelconque, demandant à opérer un rééquilibrage. On peut là aussi imaginer que la clé se situant à la dernière position dans le parcours en largeur remplace la clé supprimée, puis on la fait remonter ou redescendre jusqu'à la place qui lui convient.[6]

— L'insertion en faisant remonter une nouvelle clé, insérée à la première position libre, jusqu'à la position qu'elle devrait occuper. Cette méthode marche bien pour les tableaux, mais son implémentation à l'aide d'un type spécifique demande à savoir où l'insertion doit être faite. En fait, comme pour d'autres opérations, la difficulté réside dans la recherche du dernier élément ou du premier trou dans le parcours en largeur. Une solution peut être de mémoriser la taille de l'arbre quelque part, mais en prenant garde à respecter la structure définie, ainsi qu'à ne pas sacrifier la complexité en espace[7]. Une meilleure solution est d'utiliser une implémentation à l'aide d'un tableau et d'utiliser un tableau annexe indexé par les clés du tas[8] et maintenant leur position.

— L'augmentation d'une clé, suivi d'un replacement de ladite clé. Il est incontournable de disposer d'une structure permettant d'associer une clé et sa position, afin de pouvoir procéder à des échanges sans surcoût.

En plus des tas binaires, on trouve dans la littérature des tas binomiaux, ainsi que des tas de Fibonacci, permettant d'abaisser la complexité de l'algorithme de Dijkstra au prix d'une difficulté d'implémentation drastiquement accrue.

---

5. L'extraction peut se faire plus facilement en allégrant les contraintes sur le tas. S'il ne doit pas être presque complet, on peut faire une extraction récursive en faisant remonter le fils de clé maximale à chaque étape.

6. Le fait de devoir remonter **ou** redescendre une clé soulève déjà une difficulté possible si l'implémentation est sous la forme d'arbres où chaque nœud pointe vers ses fils, sans que l'on ne puisse accéder en temps constant à une clé quelconque ni identifier sa position.

7. Ceci étant, il est alors également possible d'insérer par la racine et de faire descendre la nouvelle clé ou une ancienne clé qu'elle remplace jusqu'à la position libre en question, à condition de connaître la direction.

8. Les fonctions de hachage, c'est magique !

Structure de file de priorité

Une *file de priorité* est une file dans laquelle les éléments sont munis d'une clé entière appelée la *priorité*. Ce n'est alors pas l'élément ajouté en premier qui est retiré en premier, mais celui qui a la plus grande priorité, en gérant les égalités de manière arbitraire ou selon une spécification particulière.

On considère qu'enfiler un élément peut revenir à lui donner une priorité moindre que les éléments de la file, mais on peut aussi définir une priorité à la main au moment de l'ajout de l'élément.

Pour rendre la structure pertinente, on a donc besoin d'une opération de plus des opérations classiques sur les files : 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).

Implémentation d'une file de priorité par un tas

Nous allons réaliser une structure de file de priorité à l'aide de tas-min.

De même que pour un dictionnaire, on va distinguer la clé et l'information, de sorte que les éléments de la file seront représentés par des couples (position de l'élément , élément).

Enfiler un élément revient à insérer dans le tas un couple (clé , élément), de sorte que la clé soit maximale parmi les clés existantes.

La façon la plus naïve de faire ceci est de mémoriser la dernière clé ajoutée et de l'incrémenter.

L'utilisation d'un tableau maintenant la position de chaque élément rend cette opération automatique, en fait, puisque la nouvelle clé sera alors la taille du tableau, dans la mesure où on ne libère pas la mémoire du tableau quand un élément sort de la file.

Une insertion intelligente dans la file prendra alors un temps logarithmique en sa taille, et l'extraction de l'élément de tête aussi, en notant que l'accès à cet élément est immédiat.

La diminution d'une clé, augmentant alors sa priorité, consiste à faire remonter la clé jusqu'à la position qu'elle doit occuper dans le tas.

La question de la présence de deux clés identiques se pose ; on peut choisir de laisser la priorité à l'élément dont la clé a été modifiée le plus tôt.

Tri par tas

Parmi les tris par comparaison asymptotiquement optimaux, signalons l'existence d'un tri appelé tri par tas (*heapsort*), dont le principe est de construire un tas puis d'en extraire les racines successivement.

La complexité en temps du tri par tas est un $\mathcal{O}(n \log n)$, en raison de la répétition d'un nombre linéaire d'opérations de complexité logarithmique.

En pratique, il s'avère que le tri rapide est meilleur que le tri par tas d'un facteur 2.

Le tri par tas sera implémenté en TP.

Chapitre 12

Structures de données relationnelles

Partie du cours : Structures de données
Prérequis : Les deux premiers chapitres de la partie
Applications : Modélisations, surtout avec des données conséquentes
Compétence visée principale : Analyser et modéliser

Ce chapitre présente la structure de graphe, en introduisant le vocabulaire associé.

Les applications des graphes sont innombrables.

Cette structure sert de base à des structures fondamentales au programme de la deuxième année, notamment les arènes de jeux et les automates.

À ceci s'ajoutent les machines de Turing qui sont trop utiles et proches du programme pour ne pas être au moins présentées.

L'idée derrière l'utilisation du terme de structure relationnelle est qu'un graphe peut représenter une relation de nature variée entre différentes données.

Ceci est en accord avec la diversité de ses applications, et de manière toute simple une relation sur un ensemble peut être décrite géométriquement par le graphe orienté correspondant.

12.1 Définitions de base

Définition

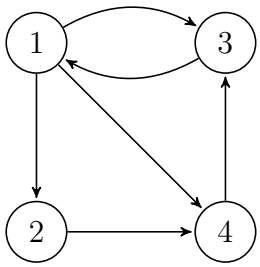
Un graphe orienté est la donnée d'un ensemble fini S de sommets (ou nœuds, comme pour les arbres) et d'un ensemble A d'arcs, qui est un sous-ensemble du produit cartésien $S \times S$. Pour tout couple $(s, t) \in A$ on appelle source ou origine de l'arc le sommet s et destination de l'arc le sommet t. Les arcs sont donc en pratique des flèches (d'où l'adjectif « orienté ») reliant deux sommets, et le fait que A soit un ensemble garantit qu'on n'ait jamais deux arcs qui ont la même source et la même destination. Le graphe est alors décrit comme le couple (S, A).

Remarque : Dans un *graphe non orienté*, il y a des *arêtes* qui n'ont pas de sens précis. Un graphe non orienté pouvant aisément être simulé par un graphe orienté, nous ne considérons que ces derniers par défaut et nous dirons simplement « graphe ». Les graphes non orientés n'auront a priori pas de type dédié dans les programmes que nous écrirons, mais éventuellement une utilisation qui tient compte que les arêtes n'ont plus d'orientation. Utiliser des ensembles de taille exactement deux pour les arêtes ne semble pas vraiment intéressant, pour résumer.

Définition

Le degré d'un sommet dans un graphe non orienté est le nombre d'arêtes touchant ce sommet. Le degré entrant (resp. sortant) d'un sommet dans un graphe orienté est le nombre d'arcs de destination (resp. source) ce sommet. On peut en déduire le degré maximal et le degré minimal d'un graphe, avec la notion de graphe régulier (non orienté, quand les degrés maximal et minimal coïncident).

On représente habituellement un graphe ainsi :



Remarque : Rien n'interdit à un arc d'arriver à son sommet de départ. Il s'agit alors d'une *boucle*. **Les boucles seront utilisées pour les graphes orientés uniquement.**

Le graphe ci-dessus est $(\{1;2;3;4\},\{(1;2);(1;3);(1;4);(2;4);(3;1);(4;3)\})$.

Un graphe peut aussi être représenté par une liste d'adjacence, en donnant S (si ce n'est pas implicite) et pour chaque élément s de S la liste des sommets tels qu'il existe un arc de s à ces sommets.

Exemple : Toujours avec l'exemple ci-dessus, la représentation par liste d'adjacence est $(\{1;2;3;4\},\{(1,\{2;3;4\});(2,\{4\});(3,\{1\});(4,\{3\})\})$.

Enfin, un graphe peut être représenté par une matrice d'adjacence, qui est une matrice carrée de taille le nombre de sommets, avec un 1 dans la cellule à la ligne i et la colonne j s'il existe un arc du i-ième sommet au j-ième sommet (en ordonnant les sommets au préalable), et un 0 sinon. Ainsi, le nombre de 1 est le nombre d'arcs.

Exemple : La matrice d'adjacence du graphe précédent est

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Pour chacune de ces représentations, on peut créer un type en OCaml et une structure en C, parfois par simple renommage.

Pour la description explicite, on propose[1] :

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

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

1. Il est possible de remplacer `string` par un autre type pour les sommets, et donc les arcs, en OCaml, mais en se souvenant de la notion de sérialisation on sait qu'avec des chaînes de caractères aucune porte n'est fermée.

Les sommets sont listés dans un tableau de chaînes de caractères, chaque élément pointant sur le début de chaque chaîne, et les arcs dans un autre tableau de chaînes de caractères, par exemple en considérant les origines aux indices pairs et les destinations aux indices impairs directement supérieurs à l'indice de l'origine correspondante.

Pour les listes d'adjacence, suivant que les sommets soient les premiers entiers naturels ou non, on pourra utiliser un tableau de listes (chaînées en C, natives en OCaml) ou un dictionnaire de telles listes indexé par les sommets.

En pratique, le programme suggère de se contenter de tableaux statiques en C, et pour les listes d'adjacence d'utiliser des « sentinelles » signalant qu'il n'y a plus de voisin pour gérer l'utilisation de tableaux de taille constante pour contenir un nombre variable de vraies données.

Une autre option proposée est de renseigner le nombre effectif de voisins dans la première case du tableau, ce qui implique d'utiliser des entiers naturels (autant dire de zéro au nombre de sommets moins un) pour les noms de sommets.

Pour les matrices d'adjacence, il est vraiment préférable que les sommets soient les premiers entiers naturels, mais il existe un repli en utilisant des dictionnaires contenant des dictionnaires ayant des booléens pour valeurs, dans les deux cas les clés seront les sommets. Cette option ne sera pas retenue a priori, et toute autre représentation cohérente peut être envisagée afin de garder de la flexibilité par rapport aux besoins et aux consignes d'une épreuve.

Définition

Un graphe biparti est un graphe dont l'ensemble des sommets peut être partitionné en deux sous-ensembles S_1 et S_2, de sorte que deux sommets de S_1 ne soient jamais reliés, de même que deux sommets de S_2.

Les graphes bipartis sont souvent non orientés, mais ce n'est pas systématique. Le chapitre de deuxième année sur la théorie des jeux suggère fortement d'utiliser des graphes orientés bipartis, par ailleurs.

12.2 Chemins

Définition

Un chemin (ou une chaîne dans un graphe non orienté) dans un graphe est une liste de sommets $s_1 s_2 \dots s_k$ tel que, pour tout $1 \leq i < k$, $(s_i; s_{i+1})$ est un arc (une arête dans un graphe non orienté ; noter qu'une chaîne n'est pas orientée non plus et qu'on exclut en principe de prendre deux fois de suite une arête dans les deux sens). On appelle s_i le prédécesseur de s_{i+1} et s_{i+1} le successeur de s_i dans le chemin. La longueur du chemin est $k-1$, c'est le nombre d'arcs traversés.

Remarque : Il n'est pas interdit qu'un chemin passe deux fois par le même sommet, mais il existe alors un chemin strictement plus court de mêmes source et destination.

Exemple : Dans le graphe donné, 1 4 3 1 2 est un chemin de longueur 4.

On constate qu'il est possible de suivre les flèches pour parcourir ce chemin.

Définition

Un circuit (ou un cycle dans un graphe non orienté) dans un graphe est un chemin dont la destination est aussi la source.

Exemple : Dans le graphe donné, 1 4 3 1 est un circuit de longueur 3.

Définition

Une composante connexe d'un graphe non orienté (S, A) est un sous-ensemble S' de S (ou pour certains auteurs le sous-graphe induit) tel que pour tous $s, t \in S'$ il existe une chaîne reliant s à t dans le graphe en ne passant que par des sommets de S'.

Définition

Un graphe non orienté est dit connexe si l'ensemble de ses sommets forme une composante connexe selon notre définition de cette notion.

Certains auteurs ajoutent une condition de maximalité (pour l'inclusion) pour être une composante connexe (aucun ensemble contenant S' ne doit la vérifier), ainsi que pour la définition suivante.

Définition

Une composante fortement connexe d'un graphe orienté (S, A) est un sous-ensemble S' de S (même remarque) tel que pour tout couple $(s, t) \in S' \times S'$ il existe un chemin de s à t dans le graphe en ne passant que par des sommets de S'.

Définition

Un graphe orienté est dit fortement connexe si l'ensemble de ses sommets forme une composante fortement connexe.

Remarque : On peut aussi trouver la notion de graphe orienté connexe chez certains auteurs, signifiant que le graphe obtenu en ignorant l'orientation est connexe.

Définition

Un chemin hamiltonien est un chemin, nécessairement de longueur $\#S - 1$, passant une et une seule fois par chacun des sommets. Un circuit hamiltonien en est le prolongement par raccord, donc de longueur $\#S$.

Exemple : Dans le graphe donné, 1 2 4 3 est un chemin hamiltonien qui peut se prolonger en le circuit hamiltonien 1 2 4 3 1.

Définition

Un chemin eulérien est un chemin, nécessairement de longueur $\#A$, passant une et une seule fois par chacun des arcs. Un circuit eulérien est un chemin eulérien qui est en plus un circuit.

Exemple : L'intuition d'un circuit eulérien dans un graphe non orienté se retrouve dans les jeux qui consistent à dessiner une figure sans lever le crayon. On retrouve par ailleurs le problème de détermination de chemins hamiltoniens dans certains casse-tête de jeux vidéo...

Proposition

Un graphe non orienté non vide admet une chaîne eulérienne si, et seulement si, il est connexe (en excluant d'éventuels sommets de degré zéro) et tous ses sommets sont de degré pair, sauf éventuellement deux.

Proposition

Un graphe orienté non vide admet un chemin eulérien si, et seulement si, il est « connexe » (même remarque) et tous ses sommets sont de degré entrant égal au degré sortant, sauf éventuellement deux ayant des différences d'un en valeur absolue.

Proposition

Soit M^n la puissance usuelle n-ième de la matrice d'adjacence M d'un graphe (S, A). Alors la cellule à la ligne i et la colonne j de M^n contient le nombre de chemins distincts de longueur n du i-ième sommet du graphe au j-ième sommet du graphe. Par ailleurs, on compte le nombre de circuits de longueur n sur la diagonale.

Définition

La fermeture transitive d'un graphe (S, A) est un graphe dont l'ensemble des arcs T est la plus petite relation transitive telle que $A \subseteq T \subseteq S \times S$. L'ensemble T est aussi l'ensemble des couples $(s; t)$ tels qu'il existe dans le graphe un chemin de s à t.

Remarque : Le terme de clôture transitive (comme pour une relation, ce qui n'est pas étonnant) est équivalent et se rencontre parfois. Cette notion peut également s'appliquer spécifiquement à l'ensemble des arcs et non pas au graphe entier.

Exemple : La fermeture transitive du graphe donné est le graphe dit *complet* (chaque couple de sommets est relié par un arc). En effet, il existe un circuit hamiltonien dans le graphe, donc on peut aller à chaque autre sommet depuis chaque sommet.

Définition

La somme booléenne de deux entiers valant 0 *ou* 1 *correspond au « OU » logique et le produit booléen correspond au « ET » logique. Ainsi, la somme booléenne de* 1 *et* 1 *sera* 1*. Les autres résultats correspondent à la somme et au produit classiques.*

Remarque : La somme et le produit booléen peuvent être vus comme les lois de composition internes du semi-anneau {Vrai,Faux}.

Définition

Le produit booléen de deux matrices ne contenant que des 0 *et des* 1 *est le produit matriciel dans lequel les additions sont des sommes booléennes.*

Remarque : On peut aussi faire le produit booléen en calculant le produit usuel et en remplaçant toutes les valeurs non nulles par 1. Une autre méthode, qui s'explique mieux visuellement, consiste à remplir les lignes du produit booléen en scannant les lignes de la matrice de droite correspondant aux colonnes où la matrice de gauche a un 1 dans la ligne étudiée.

Définition

La puissance booléenne n-ième d'une matrice carrée est sa puissance n-ième par le produit booléen.

Proposition

Soit $M^{[n]}$ la puissance booléenne n-ième de la matrice d'adjacence M d'un graphe (S, A). Alors la cellule à la ligne i et la colonne j de $M^{[n]}$ contient un 1 *si, et seulement si, il existe un chemin de longueur n du i-ième sommet du graphe au j-ième sommet du graphe.*

Proposition

Pour un entier $n \geq 1$, soit P la somme booléenne de I_n et de la matrice d'adjacence M d'un graphe (S, A) à n sommets. Alors la puissance booléenne $n-1$-ième de P est égale à la somme booléenne $M \bigoplus M^{[2]} \bigoplus \cdots \bigoplus M^{[n-1]}$, et c'est aussi la matrice d'adjacence de la fermeture transitive du graphe.

Définition

Un graphe sans circuit ou graphe acyclique est un graphe... dans lequel il n'y a pas de circuit non trivial. Dans un graphe sans circuit, le niveau d'un sommet est la longueur du plus long chemin terminant à ce sommet. Ainsi, si un sommet n'a pas de prédécesseur (et au moins un tel sommet existe si le graphe est non vide, puisque celui-ci n'a pas de circuit), il aura le niveau 0.

Remarque : On calcule le niveau de chaque sommet de manière récursive. Une fois les sommets de niveau 0 trouvés, on attribue a priori le niveau 1 aux successeurs des sommets de niveau 0, puis le niveau 2 aux successeurs des sommets de niveau 1, même s'ils ont déjà le niveau 1, puis on continue jusqu'à ce que rien ne soit modifié. Le niveau maximal possible est le nombre de sommets moins un.

Le processus termine car il n'y a pas de circuit.

Définition

Le niveau d'un graphe sans circuit est le plus grand niveau d'un de ses sommets.

Remarque : On peut modifier la représentation graphique d'un graphe sans circuit de sorte que les sommets de même niveau soient alignés horizontalement.

Remarque : Un arbre est un graphe sans circuit dont un seul sommet, appelé racine, est de niveau 0, et tel qu'il existe un et un seul chemin, appelé branche, de la racine à chaque autre sommet. Le nombre d'arcs d'un arbre est alors le nombre de sommets moins un.

Dans le cas non orienté, tout graphe connexe ayant une arête de moins que le nombre de sommets peut être vu comme un arbre en choisissant arbitrairement parmi les sommets une racine.

Pour être complet, si le graphe n'est pas vide, deux des propriétés suivantes, peu importe lesquelles, entraînent la troisième et caractérisent un arbre :
- être acyclique ;
- être connexe ;
- avoir une arête de moins que le nombre de sommets.

On peut remarquer une correspondance de vocabulaire entre les graphes et les arbres : sommet/nœud, prédécesseur/père, successeur/fils, chemin/branche, niveau d'un sommet/profondeur du sommet, niveau du graphe/hauteur de l'arbre, sommet sans successeur/feuille (et les autres sommets sont des nœuds internes), sommet sans prédécesseurs/racine, degré/arité.

Définition

Une forêt est un ensemble, éventuellement vide, d'arbres.

On appellera par exemple forêt un graphe dont les composantes connexes sont toutes des arbres.

Les arbres en question ne seront a priori pas ordonnés, même si leur stockage sera très probablement dans une structure séquentielle.

12.3 Graphes pondérés

Commençons par une définition qui concerne encore les graphes habituels.

Définition

Un chemin dans un graphe est dit optimal en longueur s'il n'existe pas de chemin de même source, de même destination et de longueur strictement moindre. Un tel chemin n'est pas forcément unique, mais il en existe au moins un dès qu'il existe un chemin de la source en question à la destination en question.

Proposition

Tout sous-chemin d'un sous-chemin optimal en longueur est optimal en longueur, mais raccorder deux chemins optimaux en longueur ne donne pas forcément un chemin optimal en longueur.

Remarque : Par convention, on pourra considérer que les chemins optimaux en longueur d'un sommet à lui-même sont les chemins vides.

Définition

Un graphe pondéré est un graphe dont les arcs sont étiquetés par un nombre, habituellement entier et potentiellement négatif, appelé le poids. Il peut alors y avoir deux arcs de même source et même destination, mais avec un poids différent (cela dit, il est fort probable que l'un des deux soit alors inutile pour les applications qui nous concerneraient).

Remarque : Suivant le contexte, le poids est appelé coût, durée, etc.

Remarque : Il faut préciser le poids de chaque arc quand on écrit un chemin dans un graphe pondéré, précisément en raison d'une ambigüité possible.

Définition

Le poids d'un chemin dans un graphe pondéré est la somme des poids des arcs qui forment ce chemin.

Définition

Un chemin dans un graphe pondéré est dit optimal en valeur (ou en poids) s'il n'existe pas de chemin de même source, de même destination et de poids strictement moindre.

Remarque : Il est possible qu'il n'y ait pas de chemin optimal en valeur lorsque le graphe admet au moins un circuit de poids négatif. Par ailleurs, le calcul de chemins optimaux en valeur est plus difficile lorsqu'il y a des arcs de poids négatif. Dans le cas contraire, nous verrons dans la section suivante qu'on peut utiliser l'algorithme de Dijkstra, qui est plus simple que l'algorithme de Bellman-Ford pour le cas général.

Remarque : Il n'y a aucune raison pour qu'un chemin optimal en longueur soit optimal en valeur, et vice-versa. On peut facilement trouver un contre-exemple, d'ailleurs : il suffit de prendre un chemin de 10 arcs de poids 1 d'un sommet à un autre et en parallèle un unique arc de poids 20.

De nombreux algorithmes sont spécifiques aux graphes pondérés et leur étude forme l'essentiel du chapitre 13, ainsi que du TP associé.

Terminons par quelques suggestions de types pour représenter des graphes pondérés, adaptés de la section précédente.

En OCaml :

```
type graphep1 = { mutable sommets : string list;
mutable arcs : (string * int * string) list; };;

type graphep2 = (int * int) list array;;

type graphe3 = int array array;;
```

Dans la première représentation, les chaînes de caractères peuvent en pratique être remplacées par un autre type si on le souhaite.

Dans la deuxième représentation, les sommets sont des entiers de zéro au nombre de sommets moins un, et les arcs de la liste d'indice `i` dans le tableau sont représentés par des couples formés par le successeur du sommet `i` et le poids de l'arc en question.

Dans la troisième représentation, plutôt que d'utiliser des options d'entiers, l'absence d'arc sera matérialisée par la valeur `max_int`. Un principe similaire s'applique en C.

En C :

```
struct graphp { int nbsommets; int nbarcs; int* arcs; };
typedef struct graphp graphe_pondere;

struct arc_pondere { int sommet; int poids; };
typedef struct arc_pondere arcp;

struct t_p { int nb_successeurs; arcp* successeurs; };
typedef struct t_p tp;

struct grapho_t_p { int nb_sommets; tp* tableau_poids; };
typedef struct grapho_t_p graphe_tableau_poids;

struct grapho_m_p { int nb_sommets; int** matrice_poids; };
typedef struct grapho_m_p graphe_matrice_poids;
```

Dans la première représentation, le champ `arcs` est un tableau de taille trois fois le nombre d'arcs, avec une alternance origine - destination - poids. Bien entendu, et comme en OCaml, cette représentation a peu de chances d'être intéressante et sera a priori laissée de côté.

Chapitre 13

Algorithmique des graphes

Partie du cours : Algorithmique
Prérequis : Graphes
Applications : Toutes les applications des graphes nécessitent de maîtriser des algorithmes de base.
Compétence visée principale : Analyser et modéliser

Les opérations élémentaires sur les graphes font l'objet d'un TP préliminaire, l'objectif de ce chapitre est de présenter des algorithmes classiques sur des graphes ayant des applications pratiques.

Avant de commencer à implémenter les algorithmes qui seront ici écrits uniquement en pseudo-code, on pourra réfléchir aux raisons qui ont conduit au choix des structures de données employées, qui optimisent la complexité tout en permettant de programmer de manière aisée, mais aussi au choix de la représentation des graphes pour extraire les données de manière pratique.

L'algorithmique des graphes est scindée en deux parties, dont une seule est traitée en première année. Elle se compose de l'ancien programme d'option deuxième année en MP, à savoir les parcours de graphes et leurs applications, ainsi que les recherches de chemins optimaux en valeur dans des graphes pondérés.

13.1 Parcours de graphes

Parcours en largeur

Le *parcours en largeur* consiste à partir d'un sommet et à « visiter » d'abord tous ses successeurs, avant de visiter tous les successeurs (non encore visités) des successeurs visités lors de la première étape, et ainsi de suite.

Le squelette d'un tel algorithme est :

```
fonction parcours_largeur(graphe, origine)
{
    ouverts <- file_vide();
    enfiler(ouverts, origine);
    fermes <- [];
    tant que ouverts n'est pas vide
    {
        s <- defiler(ouverts);
        imprimer(s);
        # ou ajouter à la liste des sommets qu'on retournera
        fermes <- fermes U {s};
        pour tous les successeurs t de s
        {
            si t n'est ni dans ouverts ni dans fermes
            {
                enfiler(ouverts, t);
            }
        }
    }
}
```

Parcours en profondeur

Le *parcours en profondeur* consiste à partir d'un sommet et à explorer d'abord un chemin en ne visitant que des sommets non encore visités, jusqu'à être bloqué et à remonter dans le chemin pour explorer un autre chemin.

Il s'agit d'un cas particulier de retour sur trace.

On peut imaginer la différence entre les deux parcours comme la différence entre piles et files, mais en pratique transformer la file en une pile et la traiter de la même façon ne donne pas exactement un parcours en profondeur, on parlera éventuellement de « parcours par pile ».

On pourra par exemple chercher à construire un graphe où ces trois parcours traitent les sommets dans trois ordres différents.

Le squelette du parcours en profondeur dans sa version récursive (une version itérative revenant à simuler la pile d'appels) est :

```
fonction parcours_profondeur(graphe, origine)
{
    fermes <- [];
    fonction recursion(s)
    {
        si s n'est pas dans fermes
        {
            imprimer(s);
            fermes <- fermes U {s};
            pour tous les successeurs t de s
            {
                recursion(t);
            }
        }
    }
    recursion(origine);
}
```

Le parcours d'un graphe peut être l'occasion de traiter les sommets au moment de leur visite en fonction des besoins.

Comme dans le cas de la programmation dynamique, une information peut être récupérée en plus du traitement d'un sommet, c'est le sommet qui a permis sa découverte.

Pour le parcours en largeur, l'enfilement est le moment où ce sommet est connu, pour le parcours en profondeur, il s'agit du déclenchement de la récursion.

À partir de ces informations, on peut créer un graphe orienté ayant les propriétés d'un arbre d'arité quelconque, qu'on appellera l'*arborescence du parcours*.

13.2 Applications des parcours

Commençons par souligner qu'une utilité évidente des parcours est de déterminer l'existence ou non d'un chemin d'un sommet à un autre ou d'une chaîne entre deux sommets suivant que le graphe soit orienté ou non.

Autour de la connexité

Considérons ici une version des parcours où la fonction renvoie la séquence des sommets visités.

Proposition

Un graphe non orienté est connexe si, et seulement si, un algorithme de parcours depuis n'importe quel sommet renvoie un objet dont la taille est le nombre de sommets.

En ce qui concerne la forte connexité d'un graphe orienté, ce n'est pas aussi simple, car un trajet retour n'est pas forcément garanti quand l'aller existe. Dans ce cas, une condition nécessaire et suffisante est de fixer n'importe quel sommet s et de vérifier que tous les sommets soient accessibles depuis s, et que s soit accessible depuis tous les autres sommets.

Afin d'éviter d'avoir besoin d'autant de parcours qu'il y a de sommets, une astuce est de faire deux parcours depuis s, un dans le graphe tel quel, et un autre dans le graphe dont les arcs sont renversés.

Pour un graphe non connexe, la méthode s'adapte et permet de constituer les composantes connexes par des parcours successifs, en partant d'un sommet non encore visité et en récupérant les nouveaux sommets rencontrés.

La recherche de composantes fortement connexes d'un graphe orienté est au programme de deuxième année uniquement.

Détection de circuits ou cycles

Faisons un léger changement dans l'algorithme du parcours en plus de l'hypothèse précédente : on se débrouille désormais pour que l'origine ne soit pas mise dans les fermés lors de son premier traitement.

Proposition

Un graphe orienté possède un circuit, et seulement si, un algorithme de parcours avec le changement en question depuis au moins un sommet renvoie une séquence contenant ce sommet.

En effet, dans ce cas l'origine a pu être rejointe au cours du parcours, car elle n'était pas directement dans la liste `fermes`.

Cette fois tout de même, il faut commencer le parcours dans un sommet du circuit, car il est possible qu'aucun circuit ne soit accessible depuis une autre origine, et quand bien même il y en aurait au moins un d'accessible mais sans qu'aucun ne contienne l'origine, la méthode suggérée ci-avant ne détecterait rien.

Pour détecter un cycle dans un graphe non orienté, le principe habituel de créer deux arcs pour chaque arête fausserait les résultats car cela génèrerait des circuits ne pouvant pas compter comme des cycles.

Une suggestion est de s'interdire de prendre un arc correspondant à celui qu'on vient de prendre, mais dans le sens inverse, et donc de mémoriser à côté de chaque sommet le sommet précédent, à exclure de la liste des voisins dans la boucle « pour ».

Tri topologique

La notion de *tri topologique* est à rapprocher de l'agencement par niveaux d'un graphe sans cycle.

Il s'agit d'une séquence de sommets (presque jamais unique) telle que pour tout arc (u, v), le sommet u figure avant v dans la séquence.

Une première contrainte est donc que le graphe soit orienté, et un tel tri ne peut pas exister si le graphe admet un cycle.

Parmi les méthodes pour obtenir un tri topologique, il y a la construction d'une séquence depuis sa fin au fur et à mesure de parcours en profondeur jusqu'à ce que tous les sommets soient visités, en mettant les sommets dans la séquence quand leur exploration est finie (ordre postfixe) et en choisissant un sommet non encore visité à la fin d'un parcours tant qu'on peut en trouver.

Une autre méthode plus simple mais moins efficace revient à travailler sur une copie du graphe et de construire le tri topologique dans l'ordre cette fois, en mettant un sommet sans prédécesseur dans la séquence puis en supprimant ce sommet avec les arcs qui en sortent, puis de recommencer jusqu'à ce que la copie du graphe soit vide.

Recherche du plus court chemin

Proposition

L'arborescence que l'on peut obtenir en effectuant un parcours ***en largeur*** *depuis un certain sommet s dans un graphe place tous les sommets accessibles depuis s à une profondeur correspondant à la plus courte taille d'un chemin depuis s.*

13.3 Chemins optimaux en valeur

Pour la recherche d'un chemin optimal en valeur dans un graphe pondéré, plusieurs cas se présentent, favorisant l'un ou l'autre des trois algorithmes présentés ici :

- Si on cherche le plus court chemin depuis un sommet fixé vers un autre sommet (voire tous les autres sommets) dans un graphe sans arc de poids strictement négatif, on privilégiera l'algorithme de Dijkstra.
- S'il y a des arcs de poids strictement négatif, on utilisera l'algorithme de Bellman-Ford pour limiter la complexité.
- Si on veut disposer de toutes les distances minimales d'un sommet à un autre, on utilisera l'algorithme de Floyd-Warshall.

Algorithme de Dijkstra

L'*algorithme de Dijkstra* est en quelque sorte une version améliorée du parcours en largeur, où on choisit à tout moment pour sommet à visiter le sommet non encore visité dont la distance à l'origine est la plus petite.

Ceci implique qu'une fois un sommet visité, sa distance à l'origine ne peut plus diminuer, voilà pourquoi aucun arc ne doit avoir de poids strictement négatif.

Pour optimiser la complexité, les sommets non encore visités sont stockés dans un tas (pour disposer d'une file de priorité). On peut alors récupérer le sommet de distance minimale, ajouter un sommet, et baisser la distance d'un sommet.

Ainsi, chaque arc sera traité au plus une fois, causant éventuellement une opération d'insertion ou de remontée dans le tas, et chaque sommet sera traité au plus une fois, causant son retrait du tas. La complexité est alors un $\mathcal{O}((m+n)\log n)$.

Par construction, la distance à l'origine de sommets visités ne peut plus être modifiée, nous n'utilisons donc pas la liste `fermes` comme dans le parcours en largeur. L'algorithme ne termine alors pas s'il existe un circuit de poids strictement négatif (mais dans ces conditions il ne pourrait pas être correct).

Une version complète de cet algorithme mémorise aussi le prédécesseur de chaque sommet $t \neq s$ dans un chemin de poids minimal de s à t. Ceci utilise un tableau auxiliaire de taille $|S|$. En ne gardant que les arcs mémorisés, on obtient un arbre décrivant un chemin optimal de s à chaque sommet : le seul chemin disponible.

```
fonction dijkstra(graphe, origine) # On considère disposer du tableau poids.
{
    d <- tableau de taille |S| initialisé à l'infini; d[origine] = 0;
    ouverts <- creer_tas(); entasser(ouverts, origine);
# on entassera suivant la valeur de distances pour la clé
    tant que ouverts n'est pas Vide
    {
        s <- extraire_racine(ouverts);
        pour tous les successeurs t de s tels que d[s] + poids[s, t] < d[t]
        {
            d[t] <- d[s] + poids[s, t];
            si d[t] était l'infini { entasser(ouverts, t); }
            sinon { remonter(ouverts, t); }
        }
    }
    retourner d;
}
```

Algorithme de Floyd-Warshall

L'*algorithme de Floyd-Warshall* consiste à travailler sur une matrice carrée de taille $|S|$ dont la cellule (i, j) contient à la fin la distance entre le i-ième et le j-ième sommet.

En fait, $|S|$ itérations sont effectuées, de sorte qu'à l'itération k le contenu de la cellule (i, j) est le poids minimal d'un chemin entre le i-ième sommet et le j-ième sommet, en ne passant que par des sommets de numéro inférieur ou égal à k, d'où l'intérêt que les sommets soient les premiers entiers naturels.

L'écriture est alors très simple :

```
fonction floyd_warshall(graphe, n) # On considère disposer du tableau poids.
{
    dist <- matrice (n, n) initialisée à l'infini; # sommets : {0, ..., n-1}
    pour tout (s, t) dans A { dist[s][t] <- poids[s, t]; }
    pour h entre 0 et n-1 { si dist[h][h] >= 0 { dist[h][h] <- 0; } }
    répéter deux fois
    {
        pour k entre 0 et n-1
        {
            pour i entre 0 et n-1
            {
                pour j entre 0 et n-1
                {
                   si ni dist[i][k] ni dist[k][j] n'est l'infini
                       dist[i][j] <- min(dist[i][j], dist[i][k] + dist[k][j]);
                }
                si dist[i][i] < 0 { dist[i][i] <- -infini; }
            }
        }
    }
    retourner dist
}
```

On a écrit « répéter deux fois » pour transmettre un éventuel code $-\infty$ arrivé sur le tard partout où il peut arriver. **Ceci est une version personnelle de l'algorithme !**

Il suffit de songer à ce que devrait donner l'algorithme sur un graphe réduit à un circuit de poids strictement négatif, et à ce qu'il donnerait si on ne refaisait pas un passage entier.

En fait, une version plus classique de l'algorithme ne fait pas cette répétition ni la ligne testant les `dist[i][i]`, mais regarde à la fin si une valeur diagonale est strictement négative, auquel cas elle signale qu'il existe un circuit de poids strictement négatif non exploité et faussant donc les résultats.

De manière intéressante, on pourra associer l'algorithme de Floyd-Warshall à la programmation dynamique alors que l'algorithme de Dijkstra est plutôt un cas particulier d'algorithme glouton optimal nécessitant cependant beaucoup de calculs à chaque étape pour savoir quelle branche privilégier.

Algorithme de Bellman-Ford

L'*algorithme de Bellman-Ford*, dont la complexité est certes supérieure à celle de l'algorithme de Dijkstra, présente sur ce dernier l'avantage de fonctionner même s'il existe des arcs de poids strictement négatif.

En pratique, cet algorithme peut détecter des circuits de poids strictement négatif et donner, en leur absence, un plus court chemin entre un sommet donné et n'importe quel autre sommet, à un aménagement près fournissant aussi le prédécesseur dans le plus court chemin calculé.

Avec des circuits de poids strictement négatif, il faudrait faire comprendre quel circuit prendre et comment faire la jonction avec l'origine et la destination, ce qui reste faisable.

Le principe est de calculer les chemins de poids minimaux et de taille k d'un sommet particulier à tous les sommets pour k de 1 au nombre de sommets moins un.

Ensuite, si faire un tour de boucle supplémentaire améliore strictement l'un des poids, c'est qu'il y avait un circuit de poids strictement négatif (et des valeurs $-\infty$ à propager).

L'écriture en pseudo-code figure donc ci-après.

```
fonction bellman_ford(S,A,poids,origine)
{
    n <- taille de S;
    d <- tableau de taille n initialisé à l'infini; d[origine] = 0;
    pour i entre 0 et 2*n-1
    {
        nv_d <- copie de d;
        pour (s, t) dans A { nv_d[t] <- min nv_d[t] (d[s] + poids(s,t)); }
        si i < n { remplacer d par nv_d; }
        sinon
        {
            pour verif entre 0 et n-1
            {
                si nv_d[verif] < d[verif]
                {
                    d[verif] <- -infini;
                }
            }
        }
    }
}
```

Ouverture : algorithme A*

En deuxième année, un chapitre d'algorithmique est l'occasion de présenter l'*algorithme* A^*, qui propose une heuristique devinant avec plus ou moins de succès quel sommet privilégier en cas d'égalité de distance avec l'origine, suivant la destination (on peut penser à des nœuds repérés par des coordonnées cartésiennes, et un graphe dont les arcs sont des lignes parallèles à un axe, auquel cas la distance dans le graphe n'a aucune raison d'être forcément liée à la norme 1, mais c'est un bon pari que de chercher à explorer d'abord un sommet rapprochant de la destination suivant cette norme).

Chapitre 14

Exploration exhaustive

Partie du cours : Algorithmique
Prérequis : Connaissances de base
Applications : À utiliser par dépit quand les méthodes des autres chapitres d'algorithmique ne donnent rien
Compétence visée principale : Imaginer et concevoir une solution

---

Ce chapitre court est formé de considérations théoriques et d'exemples pratiques autour de la méthode dite de retour sur trace, aussi appelé par son nom anglais de *backtracking*. En deuxième année, la méthode de *séparation et évaluation* (*branch and bound*) sera présentée, pour un gain en efficacité permettant d'envisager de s'attaquer à des problèmes pour lesquels l'exploration exhaustive mettrait trop de temps à donner une réponse.

14.1 Principe

Définition

On parle de retour sur trace lorsqu'un algorithme, amené à faire des choix, teste successivement toutes les possibilités en poursuivant sa progression jusqu'à tomber sur une solution qui a les propriétés attendues pour arrêter le calcul ou jusqu'à être en mesure de dire qu'une solution ne peut plus être atteinte ainsi, faisant remonter au dernier choix fait et partant sur une autre possibilité non encore explorée ou, s'il n'en reste pas, remontant encore dans l'historique des choix faits, et ce de manière non chronologique.

Structurellement, on reconnaît alors deux choses :

— d'une part, il s'agit de parcourir en profondeur l'arbre des exécutions possibles, dont les nœuds sont les moments où un choix doit être fait, avec une arité arbitraire (les choix ne sont pas forcément booléens, mais si tel est le cas on peut imaginer l'arbre des exécutions possibles comme un arbre binaire) ;

— d'autre part, la gestion de ce parcours peut être faite par une pile, qui pour chaque choix à faire mémorisera l'état d'avancement dans l'exploration des possibilités, afin de déterminer quand l'ensemble des possibilités a été testé en vain, nécessitant un dépilement pour revenir en arrière.

On comprend à la lumière de ce principe qu'il s'agit effectivement d'exploration exhaustive, comme l'évoque le titre du chapitre.

Une caricature de ce principe pourrait revenir à dire qu'on trie une séquence par un retour sur trace naïf si on choisit de prendre successivement les éléments de la séquence pour construire la version triée, en revenant en arrière dès qu'une contradiction est constatée. Il s'agit d'une version sous-optimale du tri par sélection, où un élément est mis dans la version triée même s'il ne s'agit pas du minimum, mais l'ajout d'éléments ultérieurs permettra, éventuellement de nombreuses étapes plus tard, de se rendre compte que l'élément choisi n'était pas le bon...

Un algorithme procédant par retour sur trace donnera toujours la solution optimale (ou, suivant le contexte, conclura sur l'absence de solution à un problème posé), et sera caractérisé par une complexité souvent rédhibitoire, ce que la facilité relative d'écriture ne saurait compenser.

En contrepartie, l'ordre de traitement des choix fait que la complexité en espace restera raisonnable, puisqu'en reprenant l'idée d'explorer l'arbre des possibilités, seuls les nœuds parents et l'état d'avancement de leur traitement sont matériellement mis en attente, quand on n'a pas de solution pour déduire cette information et tout faire en espace constant.

En pratique, même si le retour sur trace n'est pas en soi un algorithme, il donne un squelette aux algorithmes à écrire, qui se ressembleront suffisamment pour justifier la facilité d'écriture mentionnée ci-avant.

14.2 Exemples

Tous les exemples ci-après sont rassemblés dans un TP portant sur ce chapitre.

Plus court chemin par un parcours en profondeur

Dans le chapitre d'algorithmique des graphes, les parcours en largeur et en profondeur sont étudiés, en signalant qu'un parcours en largeur permettait de trouver le plus court chemin, ce qui n'est pas la vocation du parcours en profondeur.

Effectivement, le parcours en profondeur s'arrête dès qu'il trouve un chemin, mais on peut aussi lui donner pour consigne de chercher le chemin le plus court par retour sur trace, en suivant le principe du *parcours en profondeur itéré* (IDS en anglais).

Il s'agit de lancer un parcours en profondeur qui renonce au choix courant dès que la profondeur atteint une valeur fixée, tout en s'interdisant de repasser par un sommet visité pour économiser un peu la complexité.

Le chemin le plus court est détecté en incrémentant la valeur maximale de la profondeur à chaque échec du parcours en profondeur pour la valeur maximale précédente.

La complexité en temps s'en ressent certes par rapport à un parcours en largeur.

Pour un sujet de concours traitant du parcours en profondeur itéré, entre autres, et pleinement dans le thème du chapitre, on pourra consulter l'épreuve d'informatique A du concours X-ENS 2017 en MP.

Perles de Dijkstra

Une fois de plus, nous pouvons nous intéresser à un problème intéressant posé par Dijkstra (et une fois de plus les couleurs du drapeau néerlandais seront remplacées par les valeurs `0`, `1` et `2` dans la présentation du problème).

Il s'agit de faire un collier de perles, c'est-à-dire une séquence d'entiers, en se servant de trois couleurs, c'est-à-dire les entiers mentionnés ci-avant.

La contrainte ajoutée est qu'aucune sous-séquence, quelle que soit sa taille, ne peut se répéter deux fois de suite. Ainsi, `00` est interdit, de même que `0101` et `21012101` par exemple.

En procédant au hasard parmi les valeurs autorisées à chaque étape, on peut être amené à devoir revenir sur une, voire plusieurs, valeurs mises précédemment, lorsque l'ensemble des valeurs autorisées s'avère vide pour l'étape en cours.

Par un retour sur trace, il est possible de produire une séquence de taille demandée, voire des séquences infinies grâce à une résolution mathématique du problème (la solution est facile à trouver sur internet).

Pour ajouter de l'intérêt au retour sur trace sans passer par la suite de Thue-Morse (spoiler...), on pourra demander la plus petite séquence de taille fixée selon l'ordre lexicographique.

Jeux de logique

Cet exemple constitue l'illustration la plus intuitive du principe de retour sur trace : un humain qui résout un jeu de logique va pouvoir employer des raisonnements divers et variés pour progresser dans la complétion de la grille associée (ou d'autre chose s'il ne s'agit pas d'une grille), mais il est possible qu'à un moment plus rien ne fonctionne, suggérant d'émettre des hypothèses.

Ces hypothèses correspondent exactement aux choix faits par un algorithme ne procédant pas à la moindre réflexion.

Dans ce cas, tomber sur une contradiction signifie que, en admettant toutes les hypothèses précédentes, la dernière hypothèse faite est absurde (sa négation est alors une déduction).

Dans ce contexte, l'unicité de la solution permet de s'arrêter dès qu'après un certain nombre d'hypothèses non encore prouvées on peut compléter la grille... qu'on cherche à résoudre tout de même en testant la négation des hypothèses émises par acquit de conscience, pour ne pas profiter de l'unicité promise.

Bien entendu, cette unicité est une propriété dont un algorithme procédant à un retour sur trace doit absolument profiter, s'il peut s'épargner l'exploration de tout l'arbre des possibilités grâce à un succès dans la première branche testée.

Chapitre 15

Algorithmique des textes

Partie du cours : Algorithmique
Prérequis : Connaissances de base
Applications : Langages formels, et on prend un peu plus l'habitude de réfléchir à l'efficacité des algorithmes écrits
Compétence visée principale : Imaginer et concevoir une solution

---

Ce chapitre s'articule autour de deux problèmes : la recherche de sous-chaîne dans une chaîne, qui sera étendue en deuxième année avec la notion d'expression régulière, et la compression de textes, premier aperçu d'un enjeu important en informatique sur le stockage condensé d'information si possible sans perte.

Tous les algorithmes de ce chapitre seront expliqués sans être écrits. Leur réalisation en C et OCaml fera l'objet du TD associé (et du TP dans un cas).

15.1 Recherche dans un texte

Le problème est le suivant : on dispose d'une chaîne de caractères `s`, en principe assez longue, et on veut déterminer si une autre chaîne de caractères `m`, appelée le motif, en est une sous-chaîne, c'est-à-dire qu'en extrayant une suite ininterrompue de caractères de `s`, on obtient exactement `m`.

Les applications de ce problème sont multiples, allant du simple « contrôle-F » à l'analyse de séquences en génétique.

On rapprochera également la détection de motifs dans un texte à la reconnaissance d'images et au traitement du signal.

En pratique, bien que les chaînes de caractères soient des objets de dimension une, les algorithmes employés s'adaptent bien à la dimension deux (voire plus), un peu comme le flashcode étend le code-barres.

Le problème qui nous intéressera pourra se formuler de plusieurs façons : est-ce que le motif apparaît ? où commence sa première occurrence ? combien y en a-t-il ? Finalement, la formulation qui permet de déduire toutes les autres est la suivante : quelles sont tous les indices de départ des occurrences du motif dans la chaîne ?

Algorithme naïf

Dans un premier temps, nous allons considérer un algorithme naïf, qui récupère simplement toutes les tranches possibles de `s` ayant la taille de `m` et qui les compare à `m`.

Leur nombre est un de plus que la différence des tailles, ramené à zéro si `s` est strictement plus court que `m`.

Il est bon de connaître cette valeur pour éviter des débordements d'indice et bien baliser les boucles, la clarté du code en sera améliorée.

Le nombre de comparaisons de caractères, qui sera l'unité de complexité pour tous les algorithmes en pratique, est qualifié de quadratique, il s'agit précisément, dans le pire des cas, du produit de la taille du motif et du nombre de tranches, pouvant être atteint si la chaîne n'a qu'une lettre différente, tout comme le motif, et que cette lettre soit la même.

Pour le problème revenant simplement à détecter la présence du motif ou sa première occurrence, la complexité dans le pire des cas nécessite une adaptation de cet exemple, en remplaçant la dernière lettre du motif par une autre lettre.

Il est bon, pour limiter la complexité moyenne en temps, de s'assurer que les comparaisons entre le motif et la tranche s'arrêtent dès qu'un caractère est différent d'une part.

Quant à la complexité en espace, on peut la limiter en faisant les comparaisons à la volée plutôt que d'extraire toute une tranche.

On notera qu'en faisant les comparaisons à la volée, il y a un risque que les indices soient mal gérés : dans l'algorithme naïf, deux indices coexistent, pour simplifier c'est un pour signifier le début de la recherche (qui ne bouge pas pendant le test d'une tranche) et un pour l'exploration du motif.

Une question se pose alors : ne peut-on pas profiter des informations obtenues pendant le test d'une tranche pour l'étude de la suivante ? C'est l'objet des deux optimisations à suivre...

Algorithme de Boyer-Moore

Remarque : Le programme officiel invite à écrire une version simplifiée. La version plus efficace sera présentée en cours en faisant le travail à la main.

Une première amélioration de l'algorithme naïf, l'*algorithme de Boyer-Moore*, permet un gain conséquent dans les cas favorables, d'une part en procédant aux comparaisons entre caractères de la droite vers la gauche pour qu'un caractère de la chaîne qui serait absent du motif élimine tout un bloc au lieu de faire passer à l'indice de départ suivant, d'autre part en décalant malgré tout l'indice de départ d'autant que possible en cas d'échec d'une comparaison, grâce à une étude préliminaire du motif.

Pour ce faire, une structure d'appui un peu technique sera construite, permettant de créer une fonction de décalage. Il s'agit de déterminer, après avoir constaté au j-ième caractère en partant de la fin du motif qu'on cherchait à trouver à partir du i-ième caractère de la chaîne, à partir du combientième caractère de la chaîne la recherche sera relancée, entre i plus un (ça arrive) et i plus la taille du motif (idéalement).

Dans la version simplifiée, le décalage sera tel que le caractère de la chaîne qui aura différé du caractère correspondant dans le motif sera aligné avec la dernière position de ce caractère dans le motif qui soit à gauche de la position étudiée (plus petit décalage possible), donc pas parmi les j derniers en reprenant le contexte du paragraphe précédent.

Concrètement, pour tous les caractères possibles et tous les indices possibles dans le motif, il faut donner une valeur à ce décalage.

La question de la structure de données qui peut stocker cette information se pose : d'une part il est clair qu'on aura un tableau pour l'indexation parallèle au motif, d'autre part il y a potentiellement beaucoup de caractères absents du motif pour lesquels la réponse sera toujours la même (décalage maximal), donc mieux vaut disposer de la possibilité de donner une valeur par défaut, sans pour autant utiliser une fonction qui refera toujours les mêmes tests conditionnels.

La solution est donc d'utiliser un dictionnaire, et le choix pratique lors de l'implémentation en TP sera la table de hachage (du module idoine en OCaml, faite à la main en C).

La complexité dans le pire des cas en temps est atteinte pour une instance miroir de celle déjà présentée pour l'algorithme naïf : il s'agit d'une chaîne formée d'un seul caractère répété autant de fois qu'on souhaite et d'un motif ayant un premier caractère différent puis le caractère commun de la chaîne répété également autant de fois qu'on le souhaite.

Algorithme de Rabin-Karp

Dans un registre totalement différent, l'*algorithme de Rabin-Karp* optimise également la recherche dans le meilleur des cas et le cas moyen en déterminant rapidement si cela vaut la peine de tester une tranche.

Pour ce faire, une empreinte de la tranche va être comparée à l'empreinte du motif, et cette empreinte va correspondre au haché.

Ainsi, si la tranche a un haché différent du motif, la comparaison ne se fait pas, et sinon on commence à comparer caractère par caractère.

Une fonction de hachage idéale pour ce problème aurait la propriété de permettre de passer du haché d'une tranche au haché de la tranche suivante en temps constant.

C'est exactement ce qu'on peut faire en utilisant une évaluation de fonction polynomiale utilisant le code ASCII des caractères comme coefficients et en mettant le degré maximal au début de la chaîne.

Le haché du motif sera alors $\sum_{k=0}^{l-1} m_k b^{l-1-k}$, où l est la taille du motif, les m_k sont ses caractères et b est la valeur où la fonction polynomiale est évaluée.

En fonction de la taille du motif et de la valeur en laquelle la fonction polynomiale sera évaluée (a priori un nombre premier), des dépassements arithmétiques menacent, suggérant de raisonner à tout moment modulo un nombre (si possible premier aussi).

Dans tous les cas, modulo un entier ou non, le passage de $x := \sum_{k=0}^{l-1} s_{i+k} b^{l-1-k}$ à $\sum_{k=0}^{l-1} s_{(i+1)+k} b^{l-1-k}$, où les s_k sont des caractères de la chaîne et i est l'indice de début de la tranche en plus des notations précédentes, se fait en multipliant x par b, en retirant $s_i b^l$ (ou $s_i b^{l-1}$ avant la multiplication) et en ajoutant s_{i+l}, sachant que b^l peut être stocké pour n'être calculé qu'une fois, d'où une complexité constante à chaque étape et linéaire à la première (penser à la méthode de Horner).

Avec beaucoup de malchance sur la fonction de hachage et sur la chaîne, il faudra néanmoins faire un nombre de comparaisons entre caractères de l'ordre du produit de la taille de la chaîne et de la taille du motif.

Par exemple, avec le motif `"jjjjjjjj"` et une évaluation polynomiale en 73, la chaîne `"jjjjjjk!"` aura le même haché que `"jjjjjjjj"`, car l'évaluation polynomiale occasionne une différence de $1 \times 73^1 - 73 \times 73^0$ (la différence entre les codes de `'k'` et `'j'` est d'un, et de l'autre côté celle entre les codes de `'j'` et `'!'` est de 73).

On peut alors imaginer une chaîne qui répète cette séquence de même haché, et sept fois sur huit (au moins !) le haché correspondra et il faudra encore comparer en moyenne la moitié des caractères, d'où la complexité.

Il peut être intéressant de remarquer que cet algorithme s'adapte très bien à la recherche de plusieurs motifs à la fois, pour peu qu'ils soient de même longueur.

15.2 Compression

L'essentiel des fichiers multimédia sur l'ordinateur sont encodés et compressés, afin que leur stockage prenne aussi peu de place que possible sans préjudice sur le contenu.

Avant tout, il faut souligner que la compression n'est pas de la magie : il y a toujours la possibilité que la compression sans perte, quelle qu'elle soit, augmente la taille d'un fichier. En effet, il n'existe pas d'injection d'un ensemble fini dans un ensemble de cardinal strictement moindre.

De manière plus pratique et moins mathématique : si la compression permettait toujours un gain strict de place, comprimer le résultat de la compression de manière répétitive ramènerait tout à une taille nulle... (Normalement, le deuxième tour de compression est déjà inutile.)

L'efficacité d'un algorithme de compression se mesure en faisant le rapport entre les tailles, sachant que les données en entrée seront généralement découpées en octets alors que les données compressées seront construites bit par bit.

Ici, la compression se limitera à du texte, plus précisément à des chaînes de caractères dont les éléments seront tous issus de la table ASCII.

Une première idée naïve de compression peut revenir à décider une fois pour toutes qu'on n'aura qu'au plus soixante-quatre caractères différents dans le texte, toujours les mêmes par ailleurs, et donc au lieu de prendre un octet entier par caractère on redécoupe le tout en paquets de six bits.

La règle de décompression étant connue, aucun stockage n'est nécessaire par ailleurs, et le taux de compression sera alors de $\frac{3}{4}$.

Malheureusement, rien que les lettres de l'alphabet en capitales et en minuscules prennent cinquante-deux caractères.

On peut s'en sortir en se limitant à vingt-sept le nombre de caractères : l'alphabet plus un déclencheur de capitales (ou un interrupteur).

Ceci étant, à présent, un caractère nécessitera parfois douze bits, car la lettre sera précédée d'un code spécial. Et on court le risque que l'efficacité de la compression s'en ressente sur des textes où la casse est particulièrement peu adaptée.

Algorithme de Huffman

À la suite de l'introduction de cette section, on a pu accepter l'idée que les caractères n'allaient plus forcément être encodés sur un octet.

Allons encore plus loin : dans l'*algorithme de Huffman*, le nombre de bits pour chacun des caractères n'a pas besoin d'être toujours le même.

Cette propriété occasionne cependant un risque : comment être certain que la séquence de bits en cours de lecture, pouvant correspondre à un caractère, représente bien celui-ci et ne doit pas se prolonger pour en former un autre ?

Pour éviter ceci, on s'assurera qu'aucune séquence de bits correspondant à un caractère ne soit le préfixe d'une autre, levant toute ambigüité dans la lecture.

On parle de *code préfixe*, ce qui est une condition suffisante pour l'injectivité de la compression (mais pas forcément nécessaire).

Le nombre de bits associé à un caractère sera évidemment d'autant plus faible si le caractère est fréquent, et le codage sera construit en suivant un algorithme glouton, revenant à construire un arbre binaire à partir d'une forêt d'arbres initialement formée d'une racine (vue comme une feuille) par caractère, chacune ayant en information supplémentaire la fréquence du caractère.

À chaque étape, il s'agit de construire un nouvel arbre en choisissant deux arbres dont les racines respectives sont associées à deux fréquences minimales (en cas d'égalité le choix parmi les égalités est arbitraire) en créant un nœud ayant ces deux arbres comme fils (dans un ordre au choix) et la somme des fréquences comme information supplémentaire.

L'algorithme s'arrête quand il ne reste plus qu'un arbre.

On remarque qu'il s'agit d'un exemple d'arbre binaire où les feuilles et les nœuds internes sont de types différents.

La fréquence peut être obtenue par un pré-calcul, en stockant le nombre d'apparitions de chaque caractère dans un dictionnaire. L'arbre obtenu sera alors particulièrement adapté au texte ainsi traité (on montrera en TD qu'il s'agit d'un algorithme glouton optimal parmi les algorithmes encodant un texte avec un code préfixe).

L'inconvénient de cette méthode est que la règle d'encodage déduite de l'arbre devra être fournie en plus dans le texte compressé, prenant une place d'autant plus significative que le texte est court.

La manière d'encoder l'arbre sera discutée dans le TP associé au codage de Huffman et à la sérialisation.

Une autre solution ne consommant pas d'espace supplémentaire (ou au plus un espace constant) au niveau du texte revient à décider d'un encodage particulier, avec les fréquences canoniques des caractères (dont la ponctuation, et par extension toute la table ASCII) de la langue dans laquelle le texte est écrit.

Dans ce cas, l'arbre peut être construit une fois pour toutes et intégré aussi à l'algorithme de décompression.

Pour avoir un peu de flexibilité dans le choix de l'arbre, on peut imaginer un certain nombre d'arbres possibles, et celui qui correspond à la langue peut alors être renseigné sur les premiers bits du texte compressé.

Encore une autre idée : le premier bit peut correspondre à un booléen permettant de déduire parmi les deux méthodes possibles laquelle a été choisie.

Les fréquences seront a priori des flottants, mais rien n'interdit d'arrondir au millionème près et de multiplier par un million pour ne manipuler que des entiers. Dans la première méthode, après avoir compté les occurrences, il est par ailleurs préférable de s'en tenir à des entiers.

Une fois l'arbre terminé, le code de chaque caractère se lit en suivant le chemin depuis la racine : celle-ci correspond au mot vide et ne sera jamais une feuille (donc on ne fait pas de codage de Huffman s'il n'y a pas au moins deux caractères différents), puis le code du fils gauche (resp. droit) de tout nœud est le code du nœud suivi d'un zéro (resp. un).

On stockera ces codes dans un dictionnaire en vue de la compression, mais pour la décompression l'arbre est plus adapté, car on lit les bits l'un après l'autre, en produisant le caractère associé à chaque feuille rencontrée, puis on reviendra à la racine pour continuer la lecture.

Le fichier produit pourra être lu octet par octet, mais devra être traité bit par bit.

La difficulté principale réside dans le fait que le dernier octet avant la fin du fichier peut avoir des bits à ignorer, et il faut également prendre garde au stockage de l'arbre qui termine potentiellement en plein milieu d'un octet.

Une solution élégante à la gestion de la fin du fichier sera étudiée en TP.

Algorithme de Lempel-Ziv-Welch

L'algorithme de Huffman a permis d'identifier une stratégie pertinente pour la compression : on souhaite représenter ce qui apparaît fréquemment de manière concise.

Dans le cas de l'*algorithme de Lempel-Ziv-Welch* (LZW, pour faire court [!!!]), plutôt que d'utiliser des codes courts pour les caractères fréquents, les codes resteront toujours de même taille, et en l'occurrence il s'agira d'entiers à tout moment, mais ils pourront représenter des enchaînements de caractères arbitrairement longs.

Le principe de l'algorithme est de créer un code pour des séquences de caractères consécutifs la première fois qu'elles sont rencontrées en prolongeant d'un caractère une séquence dont le code existe déjà, et d'émettre à chaque fois le code de la plus longue séquence possible déjà enregistrée.

En fait, contrairement à ce qu'on pourrait anticiper, il n'est pas question de commencer par compter le nombre d'occurrences de séquences par un pré-calcul puis de faire la compression dans une passe ultérieure, mais de créer les codes à la volée. L'idée est qu'en pratique on n'a pas forcément le loisir de lire plusieurs fois un fichier, et on peut même imaginer une lecture en plusieurs fois sans préjudice de fonctionnement de l'algorithme.

Un parallèle dans la vie réelle permet de cerner l'intuition : à la lecture de copies, une erreur classique donne lieu à une remarque similaire à écrire plusieurs fois. Une fois qu'on l'a écrite dans une première copie, on peut être tenté de la recopier dans un rapport associée à un numéro. Pour les copies ultérieures, le numéro seul suffira. Et la cerise sur le gâteau est que pour la décompression, le rapport n'est pas nécessaire si on peut lire toutes les copies, mais ceci peut paraître obscur sans avoir eu un aperçu préalable de l'algorithme.

Il existe plusieurs versions de l'algorithme LZW, développées à partir de l'idée initiale datant de la fin des années 1970. En particulier, le travail peut tout aussi bien se faire sur les caractères que sur les bits qui composent la chaîne. En pratique, toute idée que l'on pourrait avoir en découvrant le programme peut correspondre à une variante déjà inventée...

Dans l'algorithme présenté ici, on démarre avec un « dictionnaire » (représenté par un tableau redimensionnable de chaînes de caractères supportant uniquement recherches et insertions) initialisé à partir des 256 caractères du code ASCII sur 8 bits.

Le déroulement de l'algorithme est le suivant : On crée une chaîne `b` constituée du premier caractère de la chaîne `s` à encoder, cette chaîne `b` servant de buffer en plus des variables locales déjà évoquées.

Dans la boucle principale, chacun des caractères suivants de `s` (à l'exclusion du caractère de fin de chaîne) est collé à la fin du buffer (explicitement ou non, suivant le langage et la gestion des chaînes de caractères qu'on souhaite faire), et au moment de l'ajout de ce caractère noté `c` on procède à un test : si la « fusion » de `b` et de `c` correspond à une chaîne déjà présente dans le dictionnaire, alors on passe au tour de boucle suivant en tenant compte de l'ajout, sinon on ajoute cette fusion de `b` et de `c` en tant que clé du dictionnaire **pour une utilisation à partir de la prochaine fois en particulier**, on écrit dans la pseudo-sortie le code de `b` et on remplace `b` par la chaîne limitée au caractère `c` pour le tour de boucle suivant.

Quand on a fini de lire `s`, le code de `b` est écrit dans la pseudo-sortie pour finir.

Il peut être surprenant que l'essentiel des clés du dictionnaire soient créées pour rien vu qu'on ne s'en sert pas la première fois, mais l'efficacité de l'algorithme n'en demeure pas moins très bonne, notamment car **le dictionnaire n'a pas besoin d'être transmis** et peut être reconstruit lors de l'étape de décodage, comme on le verra juste après.

Pour préserver l'efficacité justement, il s'agit de ne pas écrire dans la sortie ce qui figure dans la pseudo-sortie en tant qu'entiers, ce qui consommerait quatre octets pour chacun des entiers alors qu'il est fort probable que la taille du dictionnaire ne dépasse pas quelques milliers.

Une possibilité consiste à regarder la plus grande taille d'un code en binaire et d'utiliser systématiquement cette taille pour éviter toute ambigüité, mais il faut penser à transmettre cette information d'une manière ou d'une autre. La solution « maison » sera la suivante : le premier octet de la sortie est un entier court (non signé, même si cette précaution est a priori inutile) indiquant la taille de tous les codes, sachant qu'il n'y a aucun risque de dépasser la dizaine en pratique.

En ce qui concerne la décompression, on observe que chaque émission d'un code (sauf la dernière) correspond à la création d'une nouvelle clé dans le dictionnaire, donc le dictionnaire doit être reconstruit à la volée en produisant la clé associée, ce qui nécessite de consulter la chaîne correspondant au décodage du prochain code.

Attention, il peut arriver que lors du décodage, le prochain code soit encore absent du dictionnaire. Dans ce cas, il correspondra nécessairement au code actuel suivi de sa première lettre.

Pour illustrer cette difficulté, décoder les instances fournies en tant qu'exemple dans le TD 15 permet de faire face au problème. La fin de ce même TD demande à implémenter l'algorithme de LZW, on pourra consulter la correction pour une version reprenant les idées données ici.

Chapitre 16

Syntaxe des formules logiques

Partie du cours : Logique
Prérequis : Induction
Applications : Découvrir les applications des ensembles inductifs, construire une syntaxe
Compétence visée principale : Décrire et spécifier

Cette partie présente la logique de manière plus approfondie que le simple calcul avec des booléens, en tant qu'un des fondements de l'informatique théorique.

On note dans ce chapitre, pour une utilisation principalement dans le suivant, $\mathcal{B}$ l'ensemble {vrai, faux} des booléens, associés respectivement aux symboles $\top$ et $\bot$. On considère également un ensemble dénombrable appelé $\mathcal{V}$, contenant tous les noms de variables propositionnelles envisagés.

16.1 Introduction

La logique propositionnelle est le fragment de base de la logique et se limite à des opérations entre booléens.

On le construit à l'aide des constantes $\top$ et $\bot$, de variables propositionnelles (booléennes), prises dans un ensemble dénombrable $\mathcal{V}$, et de connecteurs, fonctions de $\mathcal{B}^n$ dans $\mathcal{B}$ pour $n \geq 1$, formant les ensembles $\mathcal{F}_n$.

On pourrait en pratique considérer $\top$ et $\bot$ comme des fonctions de $\mathcal{B}^0$ dans $\mathcal{B}$.

Ainsi, la *syntaxe* de la logique propositionnelle, c'est-à-dire la façon d'utiliser les constructeurs pour former des formules qui ont un sens, est la suivante : une formule est soit une constante, soit une variable, soit l'application d'un connecteur n-aire à n formules (appelées sous-formules de la formule), ce qui s'écrit avec des notations usuelles en informatique : $\varphi ::= \top \mid \bot \mid p \in \mathcal{V} \mid f(\underbrace{\varphi, \ldots, \varphi}_{n})$ pour $f \in \mathcal{F}_n$.

Cette ligne est à comprendre comme « ce qu'on appelle une formule et qu'on note par la suite φ peut s'écrire sous la forme « tautologie » ou sous la forme « contradiction » ou sous la forme d'un élément de $\mathcal{V}$ ou sous la forme d'un élément de $\mathcal{F}_n$ suivi d'une parenthèse ouvrante et de n formules définies inductivement de la même façon, séparées par des virgules, avec une parenthèse fermante à la fin ».

Un connecteur f est défini par sa *table de vérité*, dont une représentation est un tableau à 2^n cellules indiquant pour chaque n-uplet de booléens $(x_1, \ldots, x_n)$ la valeur de $f(x_1, \ldots, x_n)$.

L'ensemble des formules propositionnelles (avec nos connecteurs de base) peut en fait être défini par le haut et par le bas, en tant que cas particulier d'ensemble inductif.

Par le haut : c'est le plus petit sous-ensemble $\mathcal{F}$ des expressions que l'on peut engendrer avec la syntaxe précédente.

Par le bas : c'est la réunion sur $\mathbb{N}$ des ensembles de formules E_n définis par récurrence, avec $E_0 = \mathcal{V} \cup \{\top, \bot\}$, et pour $n \in \mathbb{N}$,

$$E_{n+1} = E_n \cup \{f(\varphi_1, \ldots, \varphi_n) \mid \text{les } \varphi_i \in E_n, f \text{ est un connecteur } n\text{-aire}\}.$$

Cette première façon de voir les choses ne nous conviendra pas, dans la mesure où elle donne à la fois trop de liberté (toutes les fonctions booléennes peuvent être considérées comme déjà définies, donc on déduirait une fois la sémantique abordée qu'il est inutile de construire des formules compliquées, il suffit d'imaginer une fonction équivalente à la formule et lui donner un nom) et trop de contraintes (le contrecoup de la liberté ici évoquée est le besoin de disposer de ces fonctions et de les fixer conventionnellement).

16.2 Syntaxe usuelle de la logique propositionnelle

Conventionnellement, on va limiter le nombre de connecteurs (un théorème ultérieur justifiera pourquoi), et utiliser une syntaxe simplifiée et pratique, de sorte qu'on dira désormais que l'ensemble des formules propositionnelles est (par le haut) le plus petit sous-ensemble $\mathcal{F}$ des expressions utilisant les éléments de $\mathcal{V}$, les symboles de connecteurs $\{\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow\}$ et les parenthèses $\{(,)\}$, tel que $\mathcal{V} \subseteq \mathcal{F}$, si $\varphi \in \mathcal{F}$, alors $\neg\varphi \in \mathcal{F}$ et si $\varphi, \psi \in \mathcal{F}$, alors $(\varphi\ C\ \psi) \in F$ pour tout $C \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$.

Les constantes booléennes ont été retirées par commodité, mais leur ajout ne changerait rien, au prix de l'adaptation des théorèmes à suivre.

On notera qu'une variante remplaçant $(\varphi\ C\ \psi)$ par $(\varphi)\ C\ (\psi)$ est également envisageable, et dans la pratique les parenthèses non nécessaires, comme la sémantique l'établira, ne sont pas écrites, un abus essentiel pour la lisibilité. Il est intéressant de remarquer l'absence de parenthèses associées au symbole $\neg$.

La caractérisation par le bas se déduit aisément selon le même principe que celui utilisé dans le cadre de la syntaxe générale.

On a alors un théorème de lecture unique de formules avec cette syntaxe restreinte (qui serait aussi valable avec la syntaxe générale de la section précédente, par ailleurs).

Théorème

Pour toute formule $\varphi \in \mathcal{F}$, un et un seul des trois cas suivants se présente :
- $\varphi \in \mathcal{V}$;
- *il existe une unique formule $\psi \in \mathcal{F}$ telle que $\varphi = \neg\psi$;*
- *il existe un unique couple de formules $(\psi, \theta) \in \mathcal{F}^2$ et un unique connecteur $C \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$ tel que $\varphi = (\psi\ C\ \theta)$.*

16.3 Lien entre formules et arbres

Les cas inductifs du théorème de lecture unique permet d'introduire la notion de sous-formule d'une formule φ. Une première définition possible est de considérer qu'il s'agit d'une formule apparaissant telle quelle dans φ.

De manière plus explicite, il convient tout d'abord de dire qu'on peut représenter les formules propositionnelles comme des arbres, chose qu'on fera dans un TP associé à ce chapitre, en mettant les connecteurs au niveau des nœuds internes et les variables au niveau des feuilles, sachant que les parenthèses ne seront plus nécessaires car la représentation géométrique clarifiera leur délimitation.

Dans ce cas, une sous-formule d'une formule φ est une formule représentée par un arbre qui est un descendant de la racine de l'arbre représentant φ.

À partir de ce lien entre formules et arbres, on définit classiquement la **taille** d'une formule comme le nombre de connecteurs utilisés. Cela revient à compter le nombre de nœuds internes de l'arbre représentant la formule.

On comprendra aisément comment se définit la **hauteur** d'une formule : il s'agit de la hauteur de l'arbre en question.

Attention : Ces définitions de taille et hauteur ne sont pas forcément universellement admises, et un énoncé de concours pourra en imposer une à laquelle il faudrait évidemment se tenir.

16.4 Quantificateurs et vocabulaire associé

Déjà étudiés en mathématiques, les quantificateurs feront ici l'objet de considérations théoriques afin de mettre des mots sur les principes intuitifs qu'il est toujours bon de consolider.

On appelle *quantificateur universel* le symbole $\forall$ et *quantificateur existentiel* le symbole $\exists$. En logique propositionnelle, ils ne peuvent quantifier que sur les booléens. En logique du premier ordre (hors programme), la possibilité de travailler sur un ensemble quelconque de constantes (typiquement $\mathbb{N}$) étend l'usage des quantificateurs, en logique monadique du second ordre, les ensembles interviennent, avec l'opérateur d'appartenance et la possibilité de quantifier sur des ensembles, et en logique du second ordre, toute relation d'arité quelconque peut être quantifiée.

L'extension de la syntaxe de la logique propositionnelle est la suivante : pour toute formule φ contenant déjà éventuellement des quantificateurs, les formules $\forall v, \varphi$ et $\exists v, \varphi$ en sont également. Les virgules n'ont qu'une valeur esthétique et on peut les omettre sans trop nuire à la lisibilité.

Dans une formule de la forme $\forall v, \varphi$ avec φ pouvant contenir des quantificateurs, on dit que v est une *variable liée*. C'est plus généralement le cas de toute variable immédiatement à droite d'un quantificateur à un endroit de la formule. Au contraire, toute variable apparaissant à un endroit de la formule qui ne fait partie d'aucune sous-formule introduite par un quantificateur suivi du nom de la variable en question est *libre*.

Attention par conséquent, la formule $(v \wedge \forall v, v)$ est syntaxiquement correcte mais sera à bannir, en raison de l'utilisation d'un même nom pour une variable liée et une variable libre.

La notion de variable libre ou liée se retrouve en mathématiques, où il n'est pas question de faire exister la variable introduite par une somme en dehors de celle-ci, ainsi qu'en programmation, avec la portée des variables dans les fonctions et les boucles (en notant des différences suivant le langage).

16.5 Substitutions

Soient φ et ψ deux formules propositionnelles et v une variable propositionnelle. On appelle substitution de v par ψ dans φ et on note $\varphi[v/\psi]$ (notation non universelle) la formule obtenue à partir de φ en remplaçant toutes les occurrences de v par ψ. En principe, on exclut que v apparaisse dans ψ.

Au niveau de la représentation des formules par des arbres, cela revient à remplacer toutes les feuilles étiquetées par v par l'arbre représentant ϕ dans l'arbre représentant φ.

On peut également faire plusieurs substitutions, de manière simultanée avec la syntaxe $\varphi[v_1/\psi_1, v_2/\psi_2, \ldots, v_n/\psi_n]$ ou consécutive avec une succession de crochets. Noter que ce n'est pas forcément équivalent ni commutatif à plus forte raison.

On considèrera comme syntaxiquement équivalentes la formule $\forall v, \psi$ et la formule $\forall w, \psi[v/w]$, où w est une variable propositionnelle absente de ψ.

Chapitre 17

Sémantique de vérité du calcul propositionnel

Partie du cours : Logique
Prérequis : Le chapitre précédent, sur la syntaxe
Applications : En SPE, les raisonnements seront construits avec des arborescences...
Compétence visée principale : Décrire et spécifier

17.1 Introduction

La *sémantique* de la logique propositionnelle, c'est-à-dire le sens à donner à une formule, dépend d'une *interprétation* (on dit aussi *valuation*) I, soit une affectation de chaque variable apparaissant dans la formule à un booléen, et permet de déduire la valeur de vérité de la formule.

On peut en pratique voir une interprétation comme une fonction $\mathcal{V} \to \mathcal{B}$ dont seule la restriction aux variables apparaissant dans la formule qu'on évalue importe.

La valeur de vérité de $\top$ est vrai et celle de $\bot$ est faux, la valeur de vérité d'une variable booléenne est donnée par l'interprétation, la valeur de vérité d'une formule obtenue par l'application d'un connecteur se lit dans la table de vérité de celui-ci en déterminant la valeur de vérité des sous-formules en argument.

Les connecteurs standards de la logique propositionnelle, présentés avec la syntaxe restreinte, sont $\neg$ (la négation, unaire), $\wedge$ (la conjonction, binaire [1]), $\vee$ (la disjonction, binaire), $\Rightarrow$ (l'implication, binaire) et $\Leftrightarrow$ (l'équivalence, binaire).

La sémantique est donc (toujours avec les parenthèses nécessaires vu que les connecteurs sont représentés de manière infixe) :

$$
\begin{array}{rcl}
I, \top & \models & \text{vrai} \\
I, \bot & \models & \text{faux} \\
I, p & \models & I(p) \\
I, \neg f & \models & \text{vrai si } I, f \models \text{faux, faux sinon} \\
I, (f_1 \wedge f_2) & \models & \text{vrai si } I, f_1 \models \text{vrai et } I, f_2 \models \text{vrai, faux sinon} \\
I, (f_1 \vee f_2) & \models & \text{vrai si } I, f_1 \models \text{vrai ou } I, f_2 \models \text{vrai, faux sinon} \\
I, (f_1 \Rightarrow f_2) & \models & \text{vrai si } I, f_2 \models \text{vrai dès que } I, f_1 \models \text{vrai, faux sinon} \\
I, (f_1 \Leftrightarrow f_2) & \models & \text{vrai si } I, f_1 \models \text{vrai ssi } I, f_2 \models \text{vrai, faux sinon}
\end{array}
$$

Le symbole ci-avant se lit « thèse » et on dit que l'intepprétation I est un modèle de la formule φ (ou « satisfait φ ») si, et seulement si, $I, \varphi \models$ vrai. Il est intéressant de noter que le nom anglais du symbole est précisément *models*.

On tâchera de ne pas confondre $\top$ et $\bot$, qui sont des symboles (éléments de syntaxe) avec des valeurs de vérité qu'on se force à écrire en toutes lettres.

Une formule est une *tautologie* si toute interprétation en est un modèle, une *contradiction* si aucune interprétation n'en est un modèle, et *satisfaisable* si au moins une interprétation en est un modèle. Ainsi, une formule est satisfaisable si et seulement si elle n'est pas une contradiction. Deux formules sont *équivalentes* si elles s'évaluent au même booléen quelle que soit l'interprétation.

On dit aussi que φ est une conséquence logique de la formule ψ, et on note $\psi \models \varphi$, si tout modèle de ψ est un modèle de φ, en d'autres termes, si $\psi \Rightarrow \varphi$ est une tautologie. Par extension, on dit que φ est une conséquence logique de l'ensemble de formules Γ si tout modèle de l'ensemble des formules de Γ est un modèle de φ, donc si la conjonction des formules de Γ implique φ.

1. Les connecteurs $\wedge$ et $\vee$ sont associatifs, donc autant imposer qu'ils soient binaires.

Les théorèmes suivants correspondent à des résultats somme toute intuitifs.

Tout d'abord, le théorème d'unicité des interprétations :

Théorème

Pour toute interprétation I, vue comme une fonction de $\mathcal{V}$ dans $\{$vrai, faux$\}$, il existe une unique application de $\mathcal{F}$ dans $\{$vrai, faux$\}$ prolongeant I, c'est-à-dire respectant les règles de la sémantique des connecteurs telles que définies plus tôt.

Dans la même veine, un théorème permettant de limiter la description d'une interprétation une fois une formule donnée :

Théorème

Soit φ une formule, soit $\{p_1, \ldots, p_n\}$ l'ensemble des variables qui y apparaissent et soient deux interprétations I_1 et I_2 telles que pour tout k entre 1 et n on ait $I_1(p_k) = I_2(p_k)$. Alors I_1 satisfait φ si, et seulement si, I_2 satisfait φ.

Ensuite, le théorème de substitution, qui a l'avantage de rendre valides nos règles de déduction pour n'importe quelle formule dès lors qu'on les prouve pour toutes les constantes booléennes :

Théorème

Soient I une interprétation, n un entier naturel, $\varphi, \psi_1, \ldots, \psi_n$ des formules et $p_1, \ldots, p_n$ des variables propositionnelles deux à deux distinctes et n'apparaissant dans aucune des formules ψ_k. La valeur de vérité de $\varphi[p_1/\psi_1, \ldots, p_n/\psi_n]$ avec l'interprétation I est la même que la valeur de vérité de φ avec une interprétation qui correspond à I pour les variables propositionnelles hors $p_1, \ldots, p_n$ et qui affecte à chaque p_i la valeur de vérité avec I du ψ_i correspondant.

Pour ce théorème, il faut prendre garde à ne pas confondre, car l'interprétation de base correspond à la formule modifiée, et la formule de base est évaluée selon l'interprétation modifiée !

Terminons par le théorème d'interpolation dû à William Craig.

Théorème

Soient n un entier naturel non nul, $p_1, p_2, \ldots, p_n$ des variables propositionnelles deux à deux distinctes, et φ, ψ deux formules ayant $p_1, p_2, \ldots, p_n$ comme variables propositionnelles communes. Les deux propriétés suivantes sont équivalentes :

- *La formule $(\varphi \Rightarrow \psi)$ est une tautologie.*
- *Il existe au moins une formule θ, ne contenant aucune variable propositionnelle en dehors de $p_1, \ldots, p_n$, appelée interpolante entre φ et ψ, et telle que les formules $\varphi \Rightarrow \theta$ et $\theta \Rightarrow \psi$ soient des tautologies.*

17.2 Cas des formules quantifiées

Pour déterminer la valeur de vérité d'une formule propositionnelle sans quantificateurs, il est intuitif que seules les variables propositionnelles y apparaissant effectivement ont une valeur booléenne qui nécessite d'être précisée.

En pratique, pour des formules quantifiées, ce besoin se limite aux variables libres. S'il n'y en a pas, la formule est alors forcément soit vraie soit fausse sans avoir besoin de considérer la moindre interprétation.

Par ailleurs, ceci permet de considérer comme équivalents le fait qu'une formule soit satisfaisable et que la formule obtenue en quantifiant existentiellement toutes ses variables libres soit vraie.

Puisqu'on a dit au chapitre précédent qu'une variable dans la portée d'un quantificateur était liée, il s'avère qu'on peut remplacer toute formule introduite par un quantificateur par une formule équivalente, deux fois plus longue, de la manière suivante : si φ s'écrit $\forall v, \psi$, alors φ est équivalente à $(\psi[\top/v] \wedge \psi[\bot/v])$, en remplaçant $\top$ (resp. $\bot$) par n'importe quelle tautologie (resp. contradiction) si on a refusé de considérer ce symbole comme élément de base de la syntaxe. Pour un quantificateur existentiel, on remplace $\wedge$ par $\vee$.

En répétant ce processus, on montre que toute formule quantifiée est équivalente à une formule sans quantificateur.

Complétons cette section par un petit détour hors programme :

Une formule quantifiée est en *forme prénexe* si elle s'écrit comme un enchaînement de quantificateurs suivi d'une formule sans quantificateurs. Par exemple, $\forall x, \exists y, (x \Leftrightarrow y)$ est en forme prénexe, mais $(\forall x, x \vee \forall x, \neg x)$ n'en est pas une.

De par les règles de déduction données plus tard, on peut établir que toute formule quantifiée est équivalente à une formule en forme prénexe.

Il existe également la forme de Skolem, qui consiste à remplacer toutes les quantifications existentielles par l'introduction de fonctions artificielles remplaçant les variables quantifiées par un $\exists$ en prenant comme arguments les variables quantifiées avant chaque $\exists$ retiré.

17.3 Formes normales

Dans cette section, nous allons introduire du vocabulaire grâce auquel nous formulerons un théorème majeur de la logique propositionnelle.

Un *littéral* est une variable propositionnelle (*littéral positif*) ou la négation d'une variable propositionnelle (*littéral négatif*). Une *clause* est une disjonction de littéraux, par exemple $p \vee q \vee \neg r$. Une formule est dite en *forme normale conjonctive* si elle s'écrit comme une conjonction de clauses, et en *forme normale disjonctive* si elle s'écrit comme une disjonction de conjonction de littéraux (sans nom spécifique).

Théorème

Toute formule de la logique propositionnelle peut s'écrire en forme normale conjonctive ou en forme normale disjonctive.

Seules les formules non quantifiées sont considérées ici, les formules quantifiées pouvant subir une élimination de quantificateurs pour commencer.

Ce théorème se prouve de la façon suivante pour la forme normale disjonctive : on considère une formule f de la logique propositionnelle. On note $v_1, v_2, \ldots, v_n$ les variables qui y apparaissent.

Tester la valeur de vérité de f pour les 2^n interprétations possibles des variables permet de dresser la table de vérité de f.

Soit X l'ensemble des n-uplets de constantes booléennes correspondant aux interprétations pour lesquelles f est vraie.

On considère un élément $(x_1, x_2, \ldots, x_n)$ de X : si $v_i = x_i$ pour $1 \leq i \leq n$, alors f est vraie.

La formule $x_1 \wedge x_2 \wedge \cdots \wedge x_n$ implique donc f.

On pose maintenant, pour $1 \leq i \leq n$, l_i le littéral v_i si x_i est vrai, et $\neg v_i$ sinon.

La formule $l_1 \wedge l_2 \wedge \cdots \wedge l_n$ est une conjonction de littéraux, et elle vraie à la condition expresse que tous les v_i valent les x_i respectifs.

En écrivant la disjonction des conjonctions de littéraux obtenues pour chaque élément de X, on obtient une formule φ en forme normale disjonctive.

De plus, pour toute interprétation I, si $I, f \models$ vrai, alors I correspond à un élément de X, donc l'une des disjonctions est vraie, donc φ est vraie.

Réciproquement, toujours pour toute interprétation, si φ est vraie, c'est qu'au moins une des conjonctions de littéraux est vraie (exactement une, en fait), donc chaque variable v_i vaut le x_i respectif d'un élément de X, donc f est vraie.

Les formules f et φ étant simultanément vraies ou fausses pour chaque interprétation, elles sont équivalentes.

Le lecteur motivé pourra faire la preuve pour la forme normale conjonctive, facilitée par les lois de De Morgan.

Corollaire

On peut se contenter des variables booléennes et des connecteurs $\neg$, $\vee$ et $\wedge$ pour écrire une formule équivalente à n'importe quelle formule de la logique propositionnelle (même si elle utilisait la syntaxe non restreinte).

On dit que $(\neg, \vee, \wedge)$ forme un système **complet**.

Il est important de souligner qu'on peut effectivement obtenir des tautologies et des contradictions sans se servir de vrai ni faux, respectivement avec $p \vee \neg p$ et $p \wedge \neg p$, pour que les définitions ne créent pas de souci.

Ceci justifie la validité du remplacement des quantifications même lorsque $\top$ et $\bot$ ne sont pas des éléments de base de la syntaxe.

D'autres systèmes complets existent : $(\neg, \vee)$ et $(\neg, \wedge)$ d'après les lois de De Morgan, mais aussi $(\bot, \Rightarrow)$ et (NAND) (preuve en exercice pour ces deux derniers).

17.4 Règles de déduction

Remarque : Les systèmes de déduction permettant d'écrire des preuves de formules sous forme d'arbres sont nombreux, un aperçu en est donné en deuxième année.

On utilise ici les connecteurs standards de la logique propositionnelle, et on donne les règles de déduction les plus habituelles, permettant de déduire des formules (généralement plus concises) à partir d'autres.

Pour toutes ces règles, φ, ψ et autres symboles représentent des formules quelconques (voir cependant le théorème de substitution).

Seules les parenthèses utiles sont écrites, en considérant que la négation est prioritaire sur tous les autres connecteurs.

Une première vague concernera des formules non quantifiées :

— Remplacement de l'implication : $\varphi \Rightarrow \psi$ est équivalente à $\neg\varphi \vee \psi$.
— Remplacement de l'équivalence : $\varphi \Leftrightarrow \psi$ est équivalente à $(\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$ et à $(\varphi \wedge \psi) \vee (\neg\varphi \wedge \neg\psi)$.
— Double négation : $\neg\neg\varphi$ est équivalente à φ[2].

2. Il se trouve que certains systèmes ne permettent pas d'utiliser cette règle.

- Lois de De Morgan : $\neg(\varphi \wedge \psi)$ est équivalente à $\neg\varphi \vee \neg\psi$, et $\neg(\varphi \vee \psi)$ est équivalente à $\neg\varphi \wedge \neg\psi$.
- Idempotence : $\varphi \vee \varphi$ et $\varphi \wedge \varphi$ sont équivalentes à φ, et identité : $\varphi \Rightarrow \varphi$ est une tautologie.
- Commutativité : $\varphi \vee \psi$ est équivalente à $\psi \vee \varphi$ et $\varphi \wedge \psi$ est équivalente à $\psi \wedge \varphi$.
- Associativité : $\varphi \vee (\psi \vee \theta)$ est équivalente à $(\varphi \vee \psi) \vee \theta$ et $\varphi \wedge (\psi \wedge \theta)$ est équivalente à $(\varphi \wedge \psi) \wedge \theta$.
- Distributivité : $\varphi \vee (\psi \wedge \theta)$ est équivalente à $(\varphi \vee \psi) \wedge (\varphi \vee \theta)$ et $\varphi \wedge (\psi \vee \theta)$ est équivalente à $(\varphi \wedge \psi) \vee (\varphi \wedge \theta)$.
- Contradiction : $\varphi \wedge \neg\varphi$ est une contradiction, et tiers exclu : $\varphi \vee \neg\varphi$ est une tautologie.
- Absorption : $\varphi \vee (\varphi \wedge \psi)$ et $\varphi \wedge (\varphi \vee \psi)$ sont équivalentes à φ.
- $(\varphi \Rightarrow \psi) \vee (\psi \Rightarrow \varphi)$ est une tautologie.
- Raisonnement par distinction de cas : $(\varphi \Rightarrow \psi) \wedge (\neg\varphi \Rightarrow \psi)$ est équivalente à ψ.
- Raisonnement par contraposée : $\varphi \Rightarrow \psi$ est équivalente à $\neg\psi \Rightarrow \neg\varphi$.
- Raisonnement par l'absurde : $\varphi \Rightarrow \bot$ est équivalente à $\neg\varphi$.
- Loi de Peirce : $(\varphi \Rightarrow \psi) \Rightarrow \varphi$ implique φ.
- Modus ponens : $(\varphi \Rightarrow \psi) \wedge \varphi$ implique ψ, et modus tollens : $(\varphi \Rightarrow \psi) \wedge \neg\psi$ implique $\neg\varphi$ (syllogismes).
- Transitivité de l'implication (ou modus barbara) : $(\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \theta)$ implique $\varphi \Rightarrow \theta$.
- Distributivité à gauche : $\varphi \Rightarrow (\psi \wedge \theta)$ est équivalente à $(\varphi \Rightarrow \psi) \wedge (\varphi \Rightarrow \theta)$ et $\varphi \Rightarrow (\psi \vee \theta)$ est équivalente à $(\varphi \Rightarrow \psi) \vee (\varphi \Rightarrow \theta)$;
- ... mais $(\varphi \wedge \psi) \Rightarrow \theta$ est équivalente à $(\varphi \Rightarrow \theta) \vee (\psi \Rightarrow \theta)$ et $(\varphi \vee \psi) \Rightarrow \theta$ est équivalente à $(\varphi \Rightarrow \theta) \wedge (\psi \Rightarrow \theta)$;

Par ailleurs, l'implication n'est pas associative, mais l'équivalence l'est.

Attention, $\varphi_1 \Leftrightarrow \varphi_2 \Leftrightarrow \cdots \Leftrightarrow \varphi_n$, quel que soit le parenthésage, ne doit alors pas être interprété comme « toutes les formules sont équivalentes ».

En fait, une notation avec un grand symbole d'équivalence à l'image des sommes, produits, conjonctions, disjonctions, etc. est à éviter en raison de cette ambiguïté.

On prouve que $\varphi_1 \Leftrightarrow \varphi_2 \Leftrightarrow \cdots \Leftrightarrow \varphi_n$ est vraie si, et seulement si, le nombre de formules fausses est pair.

Les règles suivantes s'appliquent à des formules quantifiées et servent à trouver une formule équivalente en forme prénexe à toute formule quantifiée.

En pratique, elles peuvent s'étendre aux cas où la quantification couvre un autre ensemble de constantes que $\mathcal{B}$, et donc s'appliquer en mathématiques.

— Négation d'un quantificateur (très classique) : $\neg\forall v, \varphi$ est équivalente à $\exists v, \neg\varphi$, et $\neg\exists v, \varphi$ est équivalente à $\forall v, \neg\varphi$.
— Élimination d'un quantificateur (déjà vue par avant) : $\forall v, \varphi$ est équivalente à $(\varphi[v/\top] \wedge \varphi[v/\bot])$ et $\exists v, \varphi$ est équivalente à $(\varphi[v/\top] \vee \varphi[v/\bot])$.
— Renommage : $\forall v, \varphi$ est équivalente à $\forall w, \varphi[v/w]$ où w n'apparaît pas dans φ.
— Capture d'une conjonction ou d'une disjonction : $\varphi \wedge \forall v, \psi$ est équivalente à $\forall v, (\varphi \wedge \psi)$, à condition que v n'apparaisse pas dans φ (commencer par un renommage autrement). Il en va de même en échangeant l'ordre autour des $\wedge$, mais aussi en remplaçant les $\wedge$ par des $\vee$, ainsi qu'en remplaçant les $\forall$ par des $\exists$.
— Capture d'une implication par la droite : $\varphi \Rightarrow \forall v, \psi$ est équivalente à $\forall v, (\varphi \Rightarrow \psi)$, avec la même condition sur v. Il en va de même en remplaçant les $\forall$ par des $\exists$.
— Gestion d'une implication par la gauche (remarquer l'analogie avec la distributivité de l'implication sur conjonction et disjonction) : $(\forall v, \varphi) \Rightarrow \psi$ (parenthésée de manière claire plutôt qu'en suivant la syntaxe) est équivalente à $\exists v, (\varphi \Rightarrow \psi)$, et $(\exists v, \varphi) \Rightarrow \psi$ est équivalente à $\forall v, (\varphi \Rightarrow \psi)$.
— Distributivité partielle avec la conjonction : $\forall v, (\varphi \wedge \psi)$ est équivalente à $\forall v, \varphi \wedge \forall v, \psi$, mais $\exists v, (\varphi \wedge \psi)$ implique seulement $\exists v, \varphi \wedge \exists v, \psi$ sans réciproque garantie.
— Distributivité partielle avec la disjonction : $\exists v, (\varphi \vee \psi)$ est équivalente à $\exists v, \varphi \vee \exists v, \psi$, mais $\forall v, \varphi \vee \forall v, \psi$ implique seulement $\forall v, (\varphi \vee \psi)$ sans réciproque garantie.
— Distributivité partielle avec l'implication : $\forall v, (\varphi \Rightarrow \psi)$ implique $\forall v, \varphi \Rightarrow \forall v, \psi$. Avec des quantificateurs existentiels, c'est dans l'autre sens : $\exists v, \varphi \Rightarrow \exists v, \psi$ implique $\exists v, (\varphi \Rightarrow \psi)$.
— Passage des quantificateurs universels à gauche : $\exists v, \forall w, \varphi$ implique $\forall w, \exists v, \varphi$.

17.5 Problème SAT

On appelle SAT le problème suivant : étant donné une formule de la logique propositionnelle, cette formule est-elle satisfaisable ?

Pour résoudre ce problème, il suffit de l'évaluer avec toutes les interprétations possibles.

Il n'existe pas à l'heure actuelle d'algorithme asymptotiquement plus efficace (c'est aussi un problème NP-complet, comme vu en deuxième année), mais des algorithmes appelés SAT-solvers ont été développés pour travailler en des temps très acceptables en moyenne.

L'un des plus connus, datant de la moitié du vingtième siècle, est DPLL, du nom de ses quatre auteurs (Davis, Putnam, Logemann et Loveland).

Traditionnellement, on demande à ce que la formule donnée en instance du problème SAT soit en FNC, mais ce n'est pas nécessaire et cela n'a pas d'impact sur la complexité.

Cependant, le problème serait trivial avec une formule en FND (vérifier qu'au moins une disjonction de conjonctions existe sans être une contradiction).

Ceci n'est pas aberrant car la formule en FND équivalente à une formule quelconque obtenue en suivant la preuve ci-avant peut être exponentiellement plus grosse que la formule de départ.

Des variantes du problème SAT sont les problèmes n-SAT, où n est un entier fixé. Cette fois-ci la FNC est obligatoire, et les clauses doivent être de taille n au plus (ou n exactement, quitte à répéter des littéraux pour compléter).

Pour $n \geq 3$, le problème reste NP-complet, mais il se résout facilement pour $n = 2$, comme on le verra dans le TD associé.

Une autre variante est le problème MAX-SAT, qui n'est pas un *problème de décision* (c'est-à-dire un problème pour lequel la réponse est vrai ou faux), mais qui consiste à déterminer le plus grand nombre de clauses d'une formule en FNC qui peuvent être satisfaites par la même interprétation.

Chapitre 18

Vocabulaire des bases de données

Partie du cours : Bases de données
Prérequis : Connaissances de base
Applications : Comprendre le contexte autour des requêtes écrites (voir chapitre suivant)
Compétence visée principale : Analyser et modéliser

Un des fondements et motivations de l'étude des bases de données relationnelles et, du point de vue théorique, de l'algèbre relationnelle est l'existence de problèmes algorithmiques plus ou moins concrets pour lesquels les structures de données étudiées auparavant ne sont pas aussi efficaces que l'on peut l'espérer.

Avant tout, une question en lien avec notre sujet : on souhaite trier un ensemble de personnes. Selon quels critères ?

Alors que les listes d'entiers se triaient dans presque tous les cas suivant l'ordre canonique (croissant ou décroissant), il n'y a pas d'ordre de référence pour les humains.

C'est bien évidemment parce que chacun a ses particularités, qu'on représentera pour des objets quelconques par des attributs.

De tels attributs permettent un regroupement (on peut par exemple mettre ensemble toutes les personnes ayant des lunettes / des lentilles / subi une chirurgie au laser / etc.) voire un classement (tri par la taille, l'âge, etc.).

En pratique, le classement est toujours imaginable dans la mesure où on peut associer à une valeur d'un attribut une chaîne de caractères qui la décrit et considérer par exemple l'ordre lexicographique.

Le problème des structures de données que l'on connaît, c'est que si on tente de présenter des données regroupées selon un certain attribut, faire un regroupement selon un autre sera très complexe.

Imaginons qu'on mette les étudiants de SPE d'un lycée dans un tableau, ce tableau contenant un sous-tableau par classe et dans ce sous-tableau les élèves sont regroupés par classe l'année précédente.

Récupérer la liste des anciens MP2I nécessite un parcours de chaque sous-tableau pour accéder au sous-sous-tableau de la MP2I.

Bien entendu, à ce moment-là, récupérer la liste des MP* est rapide car elle est déjà fournie (on peut avoir besoin d'aplatir les sous-sous-tableaux suivant l'énoncé). L'idée est cependant de garder la même simplicité pour tous les problèmes de ce genre.

Ainsi, nous allons étudier dans ce chapitre les moyens d'exécuter des opérations (ou requêtes, sur un modèle concret de bases de données) sur des structures d'objets pourvus d'attributs.

18.1 Algèbre relationnelle

Cette section a été retirée du programme de classes préparatoires en 2021. Elle est laissée dans les notes de cours mais ne sera pas traitée en classe telle quelle. En pratique, il s'agit de définir en cours les notions sans les mettre dans le contexte de l'algèbre relationnelle mais directement sur leur effet pratique dans les bases de données.

L'*algèbre relationnelle* est un modèle théorique développé dans les années 1970, en grand lien avec la théorie des ensembles.

Nous allons nous appuyer sur cette dernière pour définir notre vocabulaire, et dans la section suivante un parallèle sera fait entre chaque opération présentée ici et chaque requête introduite dans le modèle concret des bases de données.

Soit un objet caractérisé par un certain nombre d'*attributs*. On va considérer cet objet comme le n-uplet formé par les valeurs de ces attributs dans un ordre fixé pour chaque objet.

Il est possible que des doublons soient alors créés dans un ensemble d'objets, ce qu'on autorise dès lors qu'on ne met pas en place une sorte d'identifiant sur lequel nous reviendrons.

Une *relation* est un ensemble fini d'objets (appelés ici *valeurs* ou *enregistrements*, dont le nombre est le *cardinal* de la relation, noté par un croisillon) ainsi définis. On précisera habituellement la structure de cette relation (voir deux définitions plus loin), tout comme on précise en théorie des ensembles sur quels ensembles une relation binaire est construite. L'homonymie n'est par ailleurs pas due au hasard.

Le *domaine* d'un attribut est l'ensemble de ses valeurs possibles (et non pas nécessairement l'ensemble des valeurs prises, tout comme une fonction réelle peut ne pas être surjective quand bien même son ensemble d'arrivée est défini comme étant $\mathbb{R}$ conventionnellement).

Un *schéma relationnel* est la précision de la structure d'une relation, sous la forme de n-uplet formé des couples (attribut, domaine) dans l'ordre (on pourra omettre le domaine dans certains cas).

Afin de permettre une représentation efficace d'une relation, on utilisera simplement un tableau (au sens habituel du terme), d'où le nom de *table* que l'on verra dans la section sur les bases de données.

Les opérations sur les relations, ou *opérateurs relationnels*, sont essentiellement des recherches d'enregistrements ou se fondent dessus en effectuant un traitement sur les résultats.

Produire un même résultat peut se faire de plusieurs façons différentes, mais pas nécessairement avec la même complexité, que ce soit du point de vue algorithmique ou de celui de la taille de la formule mathématique correspondante (les deux sont souvent en relation).

Soient deux relations R_1 et R_2 de même schéma. L'*union* (ou réunion) de R_1 et R_2, notée $R_1 \cup R_2$, est l'ensemble des enregistrements de R_1 et de ceux de R_2.

Attention, un élément apparaissant au moins une fois dans R_1 et au moins une fois dans R_2 apparaîtra exactement une fois dans $R_1 \cup R_2$, d'après la convention classique.

Par analogie, on définit l'*intersection* de R_1 et R_2, notée $R_1 \cap R_2$, comme l'ensemble, lui aussi a priori sans doublon, des enregistrements communs à R_1 et R_2.

Enfin, la *différence* de R_1 et R_2, notée $R_1 - R_2$, est l'ensemble des enregistrements de R_1 qui ne sont pas dans R_2.

Dans les trois cas, le schéma de la relation « composée » est le schéma commun aux deux relations.

Ceci étant, si on n'impose pas que les schémas soient communs mais seulement *compatibles* (c'est-à-dire les noms d'attributs peuvent ne pas être les mêmes, mais les domaines doivent être identiques, et à plus forte raison le nombre d'attributs doit être identique pour les deux relations).

On note que certaines propriétés de la théorie des ensembles ne sont plus forcément vraies en algèbre relationnelle : en raison de la présence possible de doublons, il n'est pas exclu que $\#(R_1 \cup R_2) < \max(\#R_1, \#R_2)$ et que $\#(R_1 \cap R_2) > \min(\#R_1, \#R_2)$ (si la convention est différente et que chaque égalité entre un élément de R_1 et un élément de R_2 donne un élément dans l'intersection). De même, la formule $\#(R_1 \cup R_2) = \#R_1 + \#R_2 - \#(R_1 \cap R_2)$ est potentiellement fausse.

Soient une relation R de schéma S et S' un sous-ensemble de S. La *projection* de R selon S', notée $\pi_{S'}(R)$, est la relation de schéma S' dont les enregistrements ne sont alors définis que par les valeurs pour les attributs dans S'.

Ici, des doublons peuvent être créés, et le cardinal de la projection est identique au cardinal de R.

Soient R_1 et R_2 deux relations de schémas respectifs S_1 et S_2. Le *produit cartésien* de R_1 et R_2, noté $R_1 \times R_2$, est l'ensemble des couples d'enregistrements (e_1, e_2), où $e_1 \in R_1$ et $e_2 \in R_2$.

Le schéma de ce produit cartésien est la réunion avec doublons de S_1 et de S_2, et son cardinal est le produit des cardinaux des relations.

Si de plus $S_2 \subseteq S_1$, on peut définir la *division cartésienne* de R_1 par R_2, notée $R_1 \div R_2$, par la relation, de schéma $S_1 \setminus S_2$, dont les enregistrements e sont tels que pour tout enregistrement e_2 de R_2, la fusion de e et de e_2 donne un enregistrement de R_1.

On note que $(R_1 \times R_2) \div R_2 = R_1$, mais $(R_1 \div R_2) \times R_2 \subseteq R_1$ sans garantie d'égalité (et il faut aussi que $R_1 \div R_2$ existe, pour commencer).[1]

Soient une relation R de schéma S et A' un attribut de domaine D commun avec un attribut A dans S. Le *renommage* de A en A', noté $\rho_{A \to A'}(R)$, donne une nouvelle relation dont le schéma est S'.

On peut imaginer un renommage multiple, par ailleurs.

Puisque le nom de l'attribut n'a pas vraiment d'influence sur les enregistrements, on justifie ici le fait de nécessiter des schémas compatibles pour l'union, l'intersection et la différence, plutôt que nécessairement des schémas identiques.

Soient S un schéma relationnel et R une relation de schéma S. Étant donné un *prédicat* P, c'est-à-dire une fonction dépendant d'un certain nombre d'arguments et retournant un booléen, d'argument un enregistrement de R, la *sélection* de R selon P est l'ensemble des enregistrements e dans R tels que $P(e)$ soit vrai, qu'on écrit simplement $\sigma_P(R) = \{e \in R \mid P(e)\}$.

Un prédicat de base est la comparaison de la valeur pour un certain attribut avec un certain élément de son domaine.

Remarque : Le rapport étroit entre la logique et la théorie des ensembles apparaît dans les propriétés suivantes.

— $\sigma_{P \text{ ET } Q}(R) = \sigma_P(R) \cap \sigma_Q(R)$ (privé des doublons) ;
— $\sigma_{P \text{ OU } Q}(R) = \sigma_P(R) \cup \sigma_Q(R)$ (idem) ;
— $\sigma_{\text{NON } P}(R) = R - \sigma_P(R)$ (idem) ;

La *jointure symétrique* de R_1 et R_2 selon l'égalité d'attributs $A_1 = A_2$ (où chaque A_i est dans le schéma de R_i), notée $R_1 \bowtie_{A_1 = A_2} R_2$, est l'ensemble des enregistrements du produit cartésien $R_1 \times R_2$ dont les valeurs pour les attributs A_1 et A_2 sont égales.

1. On peut faire un parallèle avec la division euclidienne. La relation $R_1 \div R_2$ est la plus grande relation R_q pour laquelle il existe une relation R_r telle que la réunion de R_r et de $R_q \times R_2$ soit R_1.

On notera qu'il n'est pas nécessaire que le domaine des attributs soient les mêmes, et que la jointure sera vide si l'intersection des domaines est vide, par exemple.

Par ailleurs, puisqu'on introduit ici une redondance, on peut se permettre de ne retenir qu'un attribut entre A_1 et A_2 dans le produit cartésien. C'est ce que font certains opérateurs de jointures dans le langage enseigné à la section suivante.

Ainsi, $R_1 \bowtie_{A_1=A_2} R_2 = \sigma_{A_1=A_2}(R_1 \times R_2)$, avec éventuellement l'application d'une projection retirant A_1 ou A_2 au résultat.

La *jointure externe à gauche* repose sur le même principe que la jointure symétrique, à ceci près que si un élément de la première table ne trouve aucun pendant dans la deuxième table au niveau de l'égalité d'attribut, il figurera néanmoins dans le résultat de la jointure avec la valeur spéciale `NULL` pour tous les attributs issus de la deuxième table. Les autres jointures ne sont pas au programme.

Restent les fonctions d'agrégation, que nous verrons plus particulièrement dans une section spécifique.

Toute l'algèbre relationnelle repose sur la composition d'opérations présentées ci-avant.

18.2 Bases de données relationnelles

Une grande partie du vocabulaire défini à la section précédente est à connaître du point de vue des bases de données. Il n'est pas rappelé ici mais présenté de manière adaptée en cours.

Une base de données est une implémentation d'un ensemble de relations (appelées ici *tables*) telles que décrites précédemment. Le stockage en lui-même de la base de données dépend du système de gestion, l'une des possibilités étant du texte clair, ou de façon plus pratique un tableur.

Les opérations de l'algèbre relationnelle sont bien entendu implémentées elles aussi (du moins les principales), ainsi que des opérations de modification, telles que l'ajout d'un *enregistrement* (appelé ici *entrée*), sa suppression ou sa modification, de même que la modification de la structure d'une table (ce qui modifie au passage les entrées), voire l'ajout ou la suppression de tables, entre autres.

18.2.1 Clés

Dans une base de données, il peut être utile de disposer de contraintes sur les entrées assurant qu'on puisse identifier chacune par un index, notamment en vue de faire des relations entre tables, et surtout afin de garantir l'unicité d'une valeur pour un attribut, ou éventuellement seulement d'un n-uplet.

C'est le concept de *clé*. Une clé (en SQL, cela se repère par un champ `UNIQUE` dans la structure) est un sous-ensemble d'attributs tel que l'insertion ou la modification d'une entrée ne soit possible que si le n-uplet de valeurs pour ces attributs ne se retrouve dans aucun autre enregistrement.

Si la table n'a pas de doublon, son schéma est en pratique une clé, mais bien entendu on préfère que le nombre d'attributs soit aussi restreint que possible.

En pratique, on ajoute souvent à la structure d'une table un attribut spécial, dont le nom fait référence à un identifiant, prévu pour être une clé, on dispose même d'une fonctionnalité dite `AUTO INCREMENT`, grâce à laquelle on n'a pas besoin de renseigner la valeur de l'identifiant, car un compteur associé à la table sait quelle sera la prochaine valeur à affecter.

Une *clé primaire* est une clé **pour laquelle la recherche est optimisée** sans autre caractérisation ; tout ceci se fait au sein de la base de données, donc avec les éléments du programme rien ne permet formellement de distinguer une clé primaire d'une autre clé.

En particulier, rien n'impose qu'une clé primaire se limite à un attribut, et une clé limitée à un attribut n'est pas forcément primaire.

Le concept de *clé étrangère* est en lien avec celui que nous venons de présenter, à ceci près qu'il n'y a pas d'unicité dans la table où une telle clé figure. En fait, une clé étrangère est un attribut correspondant normalement à une clé (souvent la clé primaire, pour profiter de l'optimisation) dans une autre table, là aussi en vue de faire des associations.

Par exemple, si on dispose d'une table contenant une liste d'élèves et d'une table contenant une liste d'épreuves, on peut créer une troisième table avec trois attributs : une clé étrangère correspondant à l'identifiant d'un élève, une autre correspondant à l'identifiant d'un devoir, et finalement la note obtenue.

C'est le contenu de cette troisième table qui peut être étudié afin de faire des calculs, et les résultats seront traités à l'aide des deux premières tables : une fois la moyenne, le maximum, etc. calculés, il s'agit de retrouver à qui ou à quelle épreuve l'information obtenue correspond.

Ces trois tables se retrouveront dans la base de données utilisée dans le TP 14.

18.3 Associations

Développé quelques années après le modèle relationnel, le *modèle entité-association* permet de faciliter la création des tables d'une base de données en mettant en forme les liens à faire entre les données de manière quasiment automatique.

On peut voir ceci comme une analyse grammaticale à partir de phrases décrivant les données contenues dans la base de données. Les noms dans les phrases seront des entités, et donneront a priori lieu à des tables. Les adjectifs associés seront des attributs (noter la coïncidence, qui n'est vraiment pas due au hasard) dans ces tables. Les verbes seront des associations, et donneront aussi a priori lieu à des tables où des clés étrangères seront mis en jeu, typiquement une pour le sujet et une pour le complément. Les adverbes seront alors des attributs de ces dernières tables.

Prenons l'exemple de la gestion d'une bibliothèque.

Les *entités* seront au moins les clients, les livres et les auteurs, avec divers attributs comme les informations personnelles du client, le format du livre, etc. Les *associations* proviendront des phrases « le client emprunte un livre » et « l'auteur écrit un livre ». On peut imaginer l'adverbe « longtemps » pour la première phrase, suggérant un attribut pour la date d'emprunt...

Il y a en pratique plusieurs types d'associations, notées `1-1`, `1-*` et `*-*` pour les principales. Il s'agit de déterminer combien d'entités peuvent être mises en relation avec combien d'entités par l'association formée (le symbole `*` signifie que le nombre peut être arbitrairement grand, en pratique, et on le retrouvera dans d'autres contextes).

Pour le cas de la bibliothèque, un client peut emprunter plusieurs livres à la fois, mais un seul client réalise l'emprunt, c'est donc une association `1-*`. Quant à l'écriture d'un livre, elle peut se faire par un collectif d'auteurs (*!*), et un auteur peut en écrire plusieurs, d'où l'association `*-*`.

Chapitre 19

Requêtes en SQL

Partie du cours : Bases de données
Prérequis : Notions de bases de données
Applications : Gestion de données sans écrire tous les algorithmes ou manipuler des tableurs
Compétence visée principale : Mettre en œuvre une solution

19.1 Introduction

Afin de communiquer avec un serveur de bases de données, un langage spécifique est nécessaire. Ce langage consiste à formuler des requêtes, reprenant les opérations de l'algèbre relationnelle, et à en récupérer les résultats.

Aux requêtes classiques de recherche et de gestion du contenu des tables s'ajoutent des requêtes d'administration, moins souvent utilisées (et qu'on préfèrera largement faire faire par un intermédiaire). Ces requêtes d'administration ne sont pas à connaître, pas même les manipulations de données modifiant le contenu de tables.

Un langage quasi incontournable pour exécuter ces requêtes est SQL [1].

En pratique, de nombreux systèmes de gestion de bases de données (ou SGBD) utilisent des surcouches de SQL, avec du sucre syntaxique.

1. pour *Structured Query Language*, qui se traduit par « langage de requête structurée »

On citera les systèmes MySQL, PostgreSQL, Oracle, etc.

Quant à SQLite, dont le nom est tout aussi connu [2], il ne s'agit pas d'un SGBD mais d'un moteur de bases de données (qu'il y a dans les SGBD, donc), nécessitant de mettre un peu plus les mains dans le cambouis.

19.2 Requêtes

Les requêtes principales concernant les données se regroupent en quatre catégories : insertion, recherche, mise à jour et suppression. [3]

L'insertion se contente d'ajouter une entrée dans la table, en renseignant tous les attributs nécessaires (si des attributs manquent, il faut préciser quelles valeurs correspondent à quels attributs pour lever l'ambiguïté, ce qui permet également de renseigner les attributs dans l'ordre que l'on souhaite, heureusement).

Les autres requêtes nécessitent de préciser à l'aide de conditions sur quel(le)s entrée(s) l'opération doit être effectuée (il s'agit de la projection). Ces conditions portent sur les attributs de la table : comparaisons, tests d'égalité, voire recherche de sous-chaîne, etc.

À ce titre, il est possible de faire appel à des fonctions prédéfinies dans le langage dont certaines sont données ci-après (fonctions d'agrégation).

La syntaxe est la suivante :

- Insertion : `INSERT INTO <table>(<attributs>) VALUES (<valeurs>)`, la partie renseignant les attributs entre parenthèses étant optionnelle si on les fournit tous ;
- Recherche : `SELECT <attribut>,...,<attribut> FROM <table>`, suivi de `WHERE <condition>` la plupart du temps, avec le joker `*` pour demander tous les attributs, ou la version sans doublon `SELECT DISTINCT` ;
- Mise à jour : `UPDATE <table> SET <attribut>=<valeur>,...,<attribut>=<valeur> WHERE <condition>` ;
- Suppression : `DELETE FROM <table> WHERE <condition>`.

2. et qui est parfois intégré à des environnements de développement intégrés pour certains langages

3. Notez le rapprochement avec les opérations sur des structures de données abstraites composées.

Ordinairement, les attributs sont entourés d'accents graves (convention de MySQL, non généralisée) et les valeurs sont entourées d'apostrophes (les guillemets sont tolérés par l'essentiel des SGBD mais non canoniques) quand il s'agit de chaînes de caractères.

En pratique, utiliser systématiquement des apostrophes est autorisé quel que soit le type, les conversions vers le bon type se font si besoin comme on l'observe en Python.

La partie commençant par `WHERE` est toujours optionnelle, et si elle est absente on fait la requête sur toutes les entrées.

Un point-virgule, séparant deux requêtes consécutives est fréquemment mis en toute fin, mais il n'y est pas nécessaire.

Attention, dans une requête de recherche la sélection vient du mot-clé `WHERE`, et `SELECT` induit une projection !

Exemples :

- `INSERT INTO ‘Etudiants‘(‘Nom‘,‘Prenom‘) VALUES ("Martin","Jean");` insère une entrée dans une table nommée `Etudiants` dont tous les champs sauf éventuellement `Nom` et `Prenom` précisés ici sont optionnels ;
- `INSERT INTO ‘Notes‘ VALUES (1,1,12);` insère cette fois une entrée dans une table à exactement trois champs ;
- `SELECT ‘Etudiant‘ FROM Notes WHERE ‘Epreuve‘ = 1 AND ‘Note‘ >= 10` récupère dans la table mentionnée les identifiants des étudiants ayant eu la moyenne à l'épreuve d'identifiant `1` ;
- `SELECT * FROM Etudiants WHERE Prenom REGEXP "J"` récupère tous les attributs des étudiants dont le prénom contient la lettre (capitale ou non) `J` ;
- `UPDATE ‘Notes‘ SET ‘Note‘ = ‘Note‘ + 2 WHERE ‘Epreuve‘ = 2` ajoute deux points à toutes les entrées correspondant à la deuxième épreuve ;
- `DELETE FROM Etudiants WHERE Nom = "Martin"` supprime tous les élèves dont le patronyme est `Martin`. Attention aux doublons ![4]

4. phpMyAdmin signale le nombre de lignes affectées par de telles requêtes, ce qui peut être utile pour arranger une catastrophe immédiatement.

19.3 Syntaxe supplémentaire pour la présentation

Le langage SQL dispose de mots-clés impactant la façon de présenter les résultats de la requête, et certains des paramètres ci-après, ne relevant pas de l'algèbre relationnelle, sont parfois propres à un système de gestion de bases de données.

Pour trier les résultats selon un attribut spécifique, on ajoutera (obligatoirement après la condition, si elle existe) `ORDER BY <attribut>` (ordre croissant, `ASC` étant sous-entendu) ou `ORDER BY <attribut> DESC` (ordre décroissant), avec d'autres possibilités dont le notable `ORDER BY RAND()` pour laisser le hasard décider.

Il est possible de n'afficher qu'un nombre limité de résultats (ce qui se combine bien avec ce qui précède) en écrivant `LIMIT <nombre> OFFSET <premier>`, ce qui donne les résultats à partir de celui dont l'indice (à partir de `0`) est précisé en "*offset*" et au nombre précisé en "*limit*" ou en s'arrêtant avant s'il n'y en a pas assez.

19.4 Syntaxe supplémentaire pour l'administration

Les requêtes concernant la structure des tables, ou des bases de données en général, sont soumises à la même remarque que les requêtes d'administration (qui gèrent les utilisateurs, entre autres) : dans de nombreux cas, on ne les effectuera pas en les saisissant mais par exemple à l'aide d'une interface.

On mentionnera simplement `TRUNCATE <table>` pour vider une table, `DROP TABLE <table>` pour la supprimer, ainsi que `DROP DATABASE <BDD>` pour supprimer une base de données entière, `RENAME TABLE <table> TO <nom>` pour renommer une table, `ALTER TABLE <table> ORDER BY <attribut>` pour la trier et `ALTER TABLE <table> ADD PRIMARY KEY(<attribut>)` pour y ajouter une clé primaire.

Et il est impossible de résister au plaisir de citer cette planche du webcomic XKCD : `https://xkcd.com/327/`!

19.5 Opérateurs pour les conditions

Les opérateurs à connaître pour l'arithmétique sont `+`, `-`, `x` et `/`. La division produit des flottants comme l'opérateur identique en Python.

Pour produire des booléens par des comparaisons, on retrouve `<`, `<=`, `>` et `>=`, mais le test d'égalité se fait avec `=` (d'autres opérateurs existent, mais sont hors programme) et le test de différence avec `<>`, sachant que `!=` est implémenté dans certaines versions de SQL, mais sera théoriquement compté faux au concours.

On trouvera éventuellement le mot-clé `LIKE` au lieu de `=` pour le test d'égalité, qui fait intervenir des expressions régulières (hors programme pour ce qui concerne SQL).

Les opérateurs booléens dans les conditions introduites par `WHERE` s'écrivent en toutes lettres en anglais : il s'agit de `AND`, `OR` et `NOT`, auxquelles on ajoute `XOR`, correspondant à soit ... soit ..., mais on regrette l'absence de `NAND` et `NOR`, qui s'obtiennent néanmoins aisément par composition.

19.6 La valeur `NULL`

Il existe en SQL une valeur absente de tous les domaines et servant de valeur exceptionnelle comme `None` ou `null` dans d'autres langages. On la note `NULL`.

Elle apparaît typiquement lors de la création d'enregistrements quand des attributs n'ont pas de valeur fournie.

Ceci étant, au moment de la création d'une table, il faut préciser pour chaque attribut si la valeur `NULL` est autorisée ou non, et si elle ne l'est pas, le système de gestion de bases de données donne la possibilité de choisir une valeur par défaut qui sera incorporée si aucune valeur n'est précisée.

Si on écrit une condition simple en SQL introduite par `WHERE` et faisant intervenir une valeur `NULL` comparée à l'aide d'opérateurs de comparaison de la section précédente, aucun résultat ne sera obtenu, et il en va de même pour la négation d'une telle condition.

Ainsi, tester ... `WHERE Annee <> NULL` ne donnera aucun résultat, que l'attribut `Annee` de l'enregistrement soit `NULL` ou pas, tester ... `WHERE Numero = NULL` n'en donnera pas non plus, que l'attribut `Numero` de l'enregistrement soit `NULL` ou pas, et d'un autre côté tester ... `WHERE Annee <> 2022` ne fera apparaître que les enregistrements dont l'attribut `Annee` est différent de `2022` **parmi ceux dont l'attribut n'est cependant pas** `NULL`.

Pour comparer à la valeur `NULL`, il faut écrire `IS NULL` ou `IS NOT NULL`, et incorporer ce genre de comparaisons dans une opération composée sur les booléens si on veut obtenir des résultats avec des valeurs existantes et d'autres avec `NULL`.

19.7 Rassembler des tables

Si on souhaite plutôt faire une union, une intersection ou une différence de résultats de requêtes, il s'agit de combiner les requêtes en les séparant par les mots-clés `UNION`, `INTERSECT` ou `EXCEPT` (les deux derniers sont rejetés par certaines versions de SQL).

La jointure utilise le mot-clé `JOIN` entre deux noms de tables, la condition associée s'exprimant juste après par `ON <condition>` (a priori une conjonction d'égalités entre des attributs, souvent une seule). Le produit cartésien se fait par la mention de tables séparées par le mot-clé `JOIN`, car il s'agit en fait d'une jointure sans condition.

Une autre jointure au programme est la jointure externe à gauche, de même syntaxe que la jointure au remplacement près du mot-clé `JOIN` par `LEFT JOIN`, qui contient tous les enregistrements qu'il y aurait dans la jointure symétrique plus tous les enregistrements de la première table pour lesquels aucun élément de la deuxième table ne peuvent être appariés par la condition de la jointure (**qui est en particulier nécessaire** donc le `ON` ne peut pas être omis ici), laissant tous les attributs de la deuxième table à `NULL` dans ces enregistrements-là.

Il s'agit typiquement des cas d'association `1 - *` du chapitre précédent.

Bien entendu, la jointure externe à droite existe avec une syntaxe intuitive.

De manière analogue à ce qu'on observe en mathématiques, les produits cartésiens peuvent s'enchaîner (les jointures aussi), et un produit cartésien ou plus généralement une jointure peut concerner deux fois la même table.

Il est à noter qu'en faisant une jointure entre une table et elle-même, on crée une ambiguïté entre tous les attributs, ce qui nous amène à introduire le renommage.

Le renommage au sein d'une requête se fait par le mot-clé `AS` après un nom d'attribut. Ceci est notamment utile lorsque la requête fait intervenir des fonctions d'agrégation, où exploiter le résultat nécessite quasiment de lui donner un nom d'attribut pratique.

La même syntaxe permet de renommer une table pour l'exécution d'une requête, ce qui n'a donc aucun impact sur les données enregistrées. Dans les deux cas, écrire le vrai nom puis l'alias sans `AS` est aussi autorisé mais moins esthétique.

Puisque les attributs peuvent provenir de deux tables, les noms d'attributs étant parfois identiques d'une table à l'autre (à éviter sauf dans le cas de clés étrangères), d'éventuels noms d'attributs redondants sont à préfixer par le nom de leur table, il est même envisageable de tout préfixer.

Exemple de préfixage systématique : `SELECT classe.nom FROM classe JOIN edt ON classe.id = edt.classe WHERE edt.salle = 218 AND edt.date = "mardi" AND edt.heure = 8`, pour les noms des étudiants qui ont cours en 218 le mardi à 8 h.

En évitant la jointure : `SELECT nom FROM classe WHERE id_classe IN (SELECT classe FROM edt WHERE id_salle = 218 AND date = "mardi" AND heure = 8)`.

19.8 Agrégation

La notion d'*agrégation* réfère au fait qu'on applique une fonction à un nombre a priori indéterminé d'arguments, qui sont des valeurs d'attributs d'un certain nombre d'enregistrements rassemblés ; les enregistrements sur lesquels la fonction est appliquée forment ce qu'on appelle un *agrégat*.

Le regroupement se fait en écrivant `GROUP BY <attribut>`, après quoi il ne reste qu'une entrée pour chaque valeur ou n-uplet de valeurs du ou des attributs en question (agissant comme une clé), entrée qui est l'agrégat en question dont on peut sélectionner le résultat de l'application d'une fonction présentée dans la suite.

Le programme mentionne cinq fonctions d'agrégation à connaître.[5] Il s'agit du comptage (`COUNT`), du minimum (`MIN`), du maximum (`MAX`), de la somme (`SUM`) et de la moyenne (`AVG`).

L'agrégation crée de nouveaux attributs (dont on a dit qu'ils sont renommés pour faciliter leur utilisation).

5. Une liste plus complète est donnée dans le mémento SQL accessible à la page `http://sql.sh/wp-content/uploads/2013/02/mysql-aide-memoire-sql-950px.png`.

Elle peut alors se composer avec des sélections (ou regroupements) après avoir été appliquée, ce qui n'est pas la même chose que les regroupements ou sélections avant d'appeler la fonction.

Par exemple, si on veut trouver les étudiants qui ont au dessus de 12 en moyenne dans une table dont les attributs sont un identifiant d'étudiant, un identifiant d'épreuve et une note (voir le TD associé), il faut faire un regroupement des entrées avec le même identifiant d'étudiant, un calcul de moyenne puis une sélection dans la table obtenue, ce qui s'écrit par exemple `SELECT Etudiant FROM (SELECT Etudiant, AVG(Note) AS Moyenne FROM notes GROUP BY Etudiant) AS Id WHERE Moyenne > 12.`[6]

Une autre possibilité est de faire un filtrage en aval, dans la mesure où la clause introduite par `WHERE` est un filtrage en amont, donc avant le regroupement, donc sans pouvoir utiliser le résultat de fonctions d'agrégations.

Le mot-clé pour le filtrage en aval est `HAVING`, il concerne des champs agrégés en priorité. Évidemment, on met dans ce cas `HAVING` après `GROUP BY`, et la requête précédente se réécrit donc `SELECT Etudiant FROM notes GROUP BY Etudiant HAVING AVG(Note) > 12`.

Bien entendu, les fonctions d'agrégations peuvent être appelées plus simplement. Par exemple, le cardinal d'une table s'obtient par `SELECT COUNT(*) FROM <table>`, et l'enregistrement maximisant un attribut : `SELECT MAX(<attribut>) FROM <table>`, qu'on peut projeter avec des attributs « compatibles ».

Dans ce cas, tout se passe comme s'il n'y avait qu'un agrégat, composé de tous les éléments de la table, ou de tous les résultats qui ont été récupérés par un filtrage avec `WHERE`.

Notons pour terminer que, pour tous les mots-clés du langage SQL, les lettres capitales sont essentiellement esthétiques et donc non obligatoires, mais pour des raisons de lisibilité et de clarté on les retrouvera toujours.

6. La partie `AS Id` est imposée par MySQL (utilisé en TP), car il faut faire un renommage lorsque l'on utilise ce qu'il appelle une « table dérivée ». En SQL standard, ce n'est pas nécessaire.

Troisième partie

Lexique

Lexique

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V