Pascal Lienhardt

L2

Bases en algorithmique et en programmation

Cours et exercices corrigés



ellipses

Références sciences

Bases en algorithmique et en programmation

Cours et exercices corrigés

Pascal Lienhardt

L2

ellipses

Collection Références sciences

dirigée par Paul de Laboulaye
paul.delaboulaye@editions-ellipses.fr

Retrouvez tous les livres de la collection et des extraits sur www.editions-ellipses.fr

ISBN 9782340-069671
©Ellipses Édition Marketing S.A., 2022
8/10 rue la Quintinie 75015 Paris

DANGER LE PHOTOCOPILLAGE TUE LE LIVRE

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5.2° et 3°a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective », et d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit constituerait une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

www.editions-ellipses.fr

Introduction

Contexte

Schématiquement, le domaine de l'*Algorithmique - Programmation* est celui du *calcul*. On cherche à *exprimer* et à *effectuer* des calculs, en vue de contribuer à l'élaboration de solutions à des problèmes.

Étant donné un problème, les questions suivantes se posent, qui nécessitent d'*analyser* le problème afin de pouvoir y répondre :

1. que calcule-t-on, et à partir de quoi ? Il s'agit ici d'identifier précisément quels sont les *résultats* attendus, et quelles sont les *données* ; en termes de *fonction*, il s'agit d'en identifier le résultat et le paramètre ;
2. comment calcule-t-on ? Il s'agit ici d'identifier un mode de calcul du résultat à partir du paramètre ; en d'autres termes, il s'agit de *concevoir et d'exprimer un algorithme*, ce qui nécessite un *langage d'algorithmique* ;
3. l'algorithme est-il correct ? Il s'agit ici de *prouver la correction* de l'algorithme, ou *a minima* de *tester* l'algorithme ;
4. quel est le coût de l'algorithme ? Il s'agit ici d'étudier la *complexité* de l'algorithme, c'est-à-dire les ressources nécessaires à son exécution.

Pour effectuer un calcul sur un ordinateur, il est nécessaire de programmer [1] l'algorithme décrivant ce calcul. Ici aussi, plusieurs questions se posent :

1. en supposant qu'il y ait le choix, quel langage de programmation utilise-t-on ? Il s'agit ici d'identifier le ou les langages de programmation offrant les mécanismes les plus adaptés compte-tenu de l'algorithme que l'on souhaite implanter ;
2. quelles notions du langage utilise-t-on ? En effet, il arrive fréquemment qu'un langage de programmation offre différents mécanismes permettant d'implanter une même notion algorithmique ;

1. Nous utilisons indifféremment les termes *programmation* et *codage*.

3. compte-tenu des choix d'implantation, le programme est-il correct par rapport à l'algorithme qu'il implante ? De même, quel est son coût d'exécution [1] ?

Bien d'autres questions se posent, liées par exemple à des questions très importantes comme la *maintenance* des codes, qui ne sont pas abordées dans cet ouvrage [2].

Objectif

L'objectif de ce livre est l'étude de *notions fondamentales en Algorithmique-Programmation*, qui concernent :

1. les *structures linéaires*, les *arbres* et les *graphes* ; ces notions permettent de représenter des informations *structurées*, ce qui est essentiel pour de nombreux problèmes, afin de représenter les données d'un problème, les résultats calculés, voire des résultats intermédiaires. Ces aspects sont liés principalement [3] à la question « Que calcule-t-on » ;
2. des *algorithmes* classiques utilisant ces structures, par exemple des algorithmes de tri utilisant des structures linéaires ou arborescentes, des algorithmes d'extraction d'informations à partir de graphes, *etc.* Ces aspects sont liés en particulier à la question « Comment calcule-t-on ? » ;
3. des notions de base en *complexité des algorithmes*, liées en particulier à la question « À quel coût calcule-t-on ? » ;
4. des mécanismes de base permettant d'*implanter* les structures linéaires, les arbres et plus généralement les graphes, ainsi que les contraintes liées à ces mécanismes, et leurs conséquences sur les complexités des opérations de manipulation de ces structures. Ces aspects sont bien sûr liés aux questions relatives à la programmation.

Les relations entre *récursivité* et *itérativité* [4] sont aussi étudiées. Ces deux mécanismes sont disponibles dans de nombreux langages d'algorithmique et de

1. Il est très fréquent en effet que les choix d'implantation limitent le domaine de validité de l'algorithme, ou modifient son coût d'exécution. Par exemple, un algorithme manipulant des entiers (au sens mathématique) peut être implanté en utilisant un type de base du langage qui ne permet de ne représenter qu'un sous-ensemble fini d'entiers (comme par exemple le type `int` en langage `C`), ce qui peut modifier à la fois son domaine de validité et sa complexité.

2. On peut aussi noter que si chacune des activités - Algorithmique, Programmation - a ses propres spécificités, il n'y a pas de frontière stricte entre les deux, ne serait-ce que parce qu'un choix d'implantation peut avoir des conséquences sur la validité ou la complexité d'un algorithme.

3. Avec des conséquences éventuelles sur les questions « Comment calcule-t-on ? » et « À quel coût calcule-t-on ? ».

4. Le terme « itérativité » ne fait pas partie du vocabulaire français. Nous l'employons par analogie avec le terme « récursivité ».

programmation. Il est intéressant de connaître les liens qui les unissent, et qui s'appuient sur certaines des notions mentionnées précédemment.

Langage

Pour simplifier la présentation, nous utilisons un unique langage, à la fois comme langage d'algorithmique et comme langage de programmation. Il s'agit d'un *sous-ensemble* du langage `OCaml` (`http://www.ocaml.org/`), accompagné de *conventions d'écriture* (*cf.* chapitre 1 page 11). Les notions présentes dans ce sous-ensemble de `OCaml` sont communes à de nombreux langages de programmation : la quasi-totalité des algorithmes [1] que nous écrirons peuvent donc être traduits de manière directe dans d'autres langages de programmation.

Nous utilisons aussi des *fonctions utilitaires*, dont le code figure sur le site des éditions Ellipses [2] : `http://www.editions-ellipses.fr`

Ces fonctions sont décrites dans l'annexe page 487 ; certaines d'entre elles, liées aux notions étudiées dans ce livre, sont définies dans les chapitres correspondants.

Prérequis - Avertissement

Pour la compréhension des notions étudiées dans ce livre, il est nécessaire que le lecteur ait des bases sur les notions de *type, fonction, conditionnelle, récursivité et itérativité*, rappelées dans le chapitre 1 page 11. À défaut, le lecteur peut consulter par exemple le livre [7].

Comme son nom l'indique, l'objectif de ce livre est l'étude de notions et d'algorithmes qui sont *à la base* de nombreuses autres notions en Algorithmique et en Programmation. Les notions fondamentales sont présentées dans les parties de cours, et illustrées sur des exemples simples. L'étude des notions est approfondie au travers d'exercices corrigés, qui occupent une place importante dans ce livre.

Certains exercices concernent la mise en œuvre d'algorithmes, dont les preuves ne sont pas toujours explicitées, car elles sortent du cadre de ce livre. Le lecteur désireux d'aller au-delà peut consulter en particulier les ouvrages [1], [6], [2], [9], [3], [5] et [4].

1. Un algorithme est toujours écrit comme une fonction, pouvant faire elle-même appel à d'autres fonctions : en conséquence, les termes « algorithme » et « fonction » sont synonymes par la suite.

2. Il en est de même pour tous les codes présentés dans ce livre.

Remerciements

J'adresse toute ma reconnaissance à Chiara Oberti, pour son aide précieuse et constante.

Tous mes remerciements aux autres personnes ayant contribué à la réalisation de ce livre, en particulier à Samuel Peltier, Laurent Fuchs et Agnès Arnould, ainsi qu"à Daniel Méneveaux, Eric Andres et Gilles Subrenat.

Table des matières

Introduction **3**

1 Préliminaires **11**
1.1 Notions de base 11
1.2 Démarche de conception - développement 29
1.3 Types abstraits 36
1.4 (Multi-)ensembles : implantation utilisée 51

2 Notions de complexité **57**
2.1 Introduction 58
2.1.1 Exemples d'algorithmes 58
2.1.2 Que calcule-t-on ? 59
2.1.3 À partir de quoi calcule-t-on ? 61
2.2 Complexité « théorique » d'un algorithme 62
2.2.1 Complexité au mieux, au pire, en moyenne 62
2.2.2 Notations de Landau 64
2.2.3 Comment calcule-t-on ? 69
2.3 Outils pour l'expérimentation 89
2.4 Exercices corrigés 95

3 Structures linéaires **139**
3.1 Files 141
3.1.1 Type abstrait 142
3.1.2 Implantation utilisée 144
3.2 Piles 146
3.2.1 Type abstrait 147
3.2.2 Implantation utilisée 148
3.3 Listes 150
3.3.1 Type abstrait 151
3.3.2 Implantation utilisée 154

3.4 Tableaux . . . 155
3.4.1 Type abstrait . . . 155
3.4.2 Implantation OCaml . . . 156
3.5 Exemples . . . 156
3.5.1 L'attente à un guichet . . . 156
3.5.2 Le labyrinthe . . . 159
3.6 Exercices corrigés . . . 165

4 Arbres **199**
4.1 Introduction . . . 199
4.2 Arbres binaires . . . 202
4.2.1 Terminologie . . . 202
4.2.2 Arbres binaires particuliers . . . 203
4.2.3 Type abstrait . . . 204
4.2.4 Implantation utilisée . . . 206
4.2.5 Manipulations élémentaires : parcours, construction . . . 207
4.3 Arbres et forêts . . . 210
4.4 Exemples . . . 213
4.4.1 Arbres binaires de recherche, arbres *AVL* . . . 213
4.4.2 Arbres d'expression . . . 215
4.4.3 Calcul d'arbre géométrique . . . 218
4.5 Exercices corrigés . . . 222

5 Graphes **271**
5.1 Terminologie partielle . . . 272
5.2 Graphes orientés . . . 275
5.2.1 Type abstrait . . . 276
5.2.2 Implantation utilisée . . . 282
5.2.3 Manipulations de base . . . 284
5.2.4 Indicateurs de complexité . . . 291
5.3 Graphes non orientés . . . 294
5.3.1 Type abstrait . . . 294
5.3.2 Implantation utilisée . . . 297
5.3.3 Exemple . . . 298
5.4 Exercices corrigés . . . 301

6 Implantation **351**
6.1 Types sommes, filtrage . . . 351
6.1.1 Types sommes . . . 351
6.1.2 Opérations . . . 353
6.1.3 Implantation des piles avec un type somme . . . 358

6.2 Tableaux, accès indicé 360
6.2.1 Tableaux 360
6.2.2 Exemples d'implantation utilisant des tableaux 361
6.3 Enregistrements et pointeurs 367
6.3.1 Principe 367
6.3.2 Implantation utilisée 370
6.3.3 Implantation des piles 372
6.4 Remarques 380
6.4.1 Remarques générales 380
6.4.2 Variables mutables, tableaux 383
6.4.3 Type `list` 385
6.5 Exercices corrigés 386

7 Récursif et Itératif **451**
7.1 Récursivité terminale et boucles 451
7.2 Du récursif à l'itératif 454
7.3 Exercices corrigés 465

Annexe **487**

Ouvrages cités **515**

Index **517**

Chapitre 1

Préliminaires

L'étude des notions présentées dans ce livre suppose que le lecteur dispose de connaissances [1] sur les notions de *type, variable, fonction, conditionnelle, récursivité, itérativité*. Celles-ci sont rappelées succinctement en section 1.1, dans le langage utilisé par la suite. Les exemples d'exécution utilisent l'interpréteur `ocaml` (*cf.* `www.ocaml.org`). La démarche classique de *conception et de développement d'un algorithme* suivie dans ce livre est présentée en section 1.2 page 29. La notion de *type abstrait*, utilisée en particulier dans les chapitres 3, 4 et 5, est explicitée en section 1.3 page 36. Les types et fonctions utilitaires que nous utilisons pour la manipulation des ensembles sont présentés en section 1.4 page 51.

1.1 Notions de base

1.1.1 Types et opérations de base

type	**exemples de valeurs**	**remarques**
`int`	`0`, `-5`, `222`	entiers compris entre -2^{62} et $2^{62}-1$
`float`	`.2`, `-3.14`, `5.2e3`	flottants
`char`	`'a'`, `'?'`, `'1'`	caractères
`string`	`"Hello World!"`	chaînes de caractères
`bool`	`true`, `false`	booléens
`unit`	`()`	valeur unique « vide »

type	**opérations**	**remarques**
`int`	`+`, `-`, `*`, `/`, `mod`	`mod` : reste de la division entière
`float`	`+.`, `-.`, `*.`, `/.`, `**`	`**` : exponentiation
`char`		pas d'opérations

1. Le lecteur voulant acquérir ces connaissances peut consulter par exemple l'ouvrage [7].

`string`	`^`	concaténation
`bool`	`&&`, `||`, `not`	respectivement *et*, *ou* et *non* logiques

Voici quelques exemples d'expressions et de valeurs : la première ligne contient l'expression saisie [1] dans l'interpréteur `ocaml`, la deuxième ligne contient la réponse de l'interpréteur, c.-à-d. la valeur de l'expression [2] :

```
# 5 / 2 ;;
- : int = 2
# 5 mod 2 ;;
- : int = 1
# 3.2 /. 5.0 ;;
- : float = 0.64
# 3.2 ** 2. ;;
- : float = 10.240000000000002
# "Hello World" ^ " : " ^ "Bonjour \n" ;;
- : string = "Hello World : Bonjour \n"
```

Les opérateurs de comparaison sont :

=, <>, >, >=, < et <=

Ces opérateurs permettent de comparer deux valeurs *de même type*, pour tout type de base, comme pour les types construits (*cf.* section 1.1.2).

```
# "alpha" < "alphabetique" ;;
- : bool = true
# "alpha" < "alentour" ;;
- : bool = false
# 2 > 3 && 'c' < 'k'
- : bool = false
# 2 > 3 || 'c' < 'k'
- : bool = true
# not(2 > 3) || ('c' > 'k' && 'c' < 'z')
- : bool = true
```

1.1.2 Types construits

1.1.2.1 Types produits

Les valeurs des *types produits* sont des n-uplets, et la comparaison de deux n-uplets de même type est faite selon l'ordre lexicographique :

```
# (4, 9) ;;
- : int * int = (4, 9)
# ('a', 5, "hello", 7.2) ;;
- : char * int * string * float = ('a', 5, "hello", 7.2)
```

1. Le symbole `#` en début de ligne ne fait pas partie de l'expression : il indique que l'interpréteur `ocaml` attend une « phrase » écrite en langage `OCaml`.

2. Plus précisément, le symbole - en début de ligne indique qu'aucun nom n'est défini ; on trouve ensuite le type de l'expression et sa valeur.

```
# (5, 7, 3) < (6, 1, 2) ;;
- : bool = true
# (5, 7, 3) < (5, 7, 4) ;;
- : bool = true
# (5, 7, 3) < (5, 7, 2) ;;
- : bool = false
```

La déclaration d'un type produit se fait suivant la syntaxe suivante :

`type` *type_name* `=` *type_1* `*` *type_2* `*` $\cdots$ `*` *type_n*

Le mot-clé `type` indique la définition d'un type, ici de nom `type_name`. Toute valeur du type produit ainsi défini est un n-uplet, dont le $i^{ème}$ composant est de type `type_i`, pour tout i compris entre `1` et `n`.

1.1.2.2 Types structurés

Une valeur d'un *type structuré* est similaire à un n-uplet dont les différents composants sont *nommés*. Il est nécessaire de déclarer un type structuré avant de l'utiliser.

`type` *type_name* `={`*field_1* `:` *type_1* `;` $\cdots$ `;` *field_n* `:` *type_n*`}`

Pour tout i compris entre `1` et `n`, le nom du $i^{ème}$ *champ* est `field_i`, et il est de type `type_i`.

Une valeur de ce type s'écrit :

`{`*field_1* `=` *value_1* `;` $\cdots$ `;` *field_n* `=` *value_n*`}`

où `value_i` est de type `type_i`, pour tout i compris entre `1` et `n`, par exemple :

```
# type t_point = {x : float; y : float} ;;
type t_point = { x : float; y : float; }
# {x = 0.0; y = 0.0} ;;
- : t_point = {x = 0.; y = 0.}
```

1.1.2.3 Types énumérés

Un type *énuméré* est déclaré de la manière suivante :

`type` *type_name* `=` *VALUE_1* `|` *VALUE_2* `|` ... `|` *VALUE_n*

où `VALUE_1`, $\cdots$, `VALUE_n` sont les valeurs du type, par exemple :

```
# type t_mycolor = BLUE | RED | GREEN ;;
type t_mycolor = BLUE | RED | GREEN
# BLUE ;;
- : t_mycolor = BLUE
```

1.1.3 Expression

En `OCaml`, toute *expression* a un type et une valeur, par exemple :

```
# not(3 + 7 > 9) || ((5 - 3 ) = 1) ;;
- : bool = false
# {x = 3.0 *. 7.2 ; y = 1.0 +. 2.0 *. 3.2} ;;
- : t_point = {x = 21.6; y = 7.4}
```

1.1.4 Variables non mutables

Intuitivement, une *variable non mutable* (aussi appelée *constante*) est un nom associé à une valeur [1], qu'il faut déclarer avant de l'utiliser ; la déclaration est caractérisée par le mot-clé `let`.

`let` *var_name* `:` *type_name* `=` *value_of_this_type* `;;`

par exemple :

```
# let x : int = 3 ;;
val x : int = 3
# let y : int = x + 2;;
val y : int = 5
# let col : t_mycolor = BLUE ;;
val col : t_mycolor = BLUE
# let (v_int, v_bool) : int * bool =  (3 mod 2, 4 + 5 > 7) ;;
val v_int : int = 1
val v_bool : bool = true
# let p : t_point = {x = 0.0; y = 1.0} ;;
val p : t_point = {x = 0.; y = 1.}
# p.x ;;
- : float = 0.
# p.y ;;
- : float = 1.
```

Le mot-clé `val` indique qu'un nom est associé à la valeur, et bien sûr à son type ; par exemple `val x : int = 3` est la réponse de l'interpréteur `ocaml` indiquant la déclaration d'une variable de nom `x`, de type `int` et de valeur `3`.

On peut noter qu'il est possible de définir des n-uplets de variables, ce qui revient à définir n variables, comme `(v_int, v_bool)` dans l'exemple ci-dessus. On peut noter aussi que l'accès à un champ d'une variable de type structuré s'écrit `var_name.field_name` : *cf.* `p.x` et `p.y` dans l'exemple ci-dessus.

Il est possible de déclarer une variable *locale* à une expression, voire plusieurs variables locales à une expression :

1. Par la suite, nous utilisons généralement le terme « variable ».

```
let var_name_1 : type_1 = expression_1 in expression
```

```
let var_name_1 : type_1 = expression_1
      and var_name_2 : type_2 = expression_2
      and ...
      and var_name_n : type_n = expression_n
in expression
```

Si l'on souhaite déclarer une variable dont la valeur est définie à partir d'une autre variable, il faut enchaîner les déclarations, par exemple :

```
# let x : int = 3 * 12 ;;
val x : int = 36
# let x : int = 3 and y : int = 2 in
      x * y
  ;;
- : int = 6
# let x : int = 3 in
    let y : int = x * 2 in
      y + x
  ;;
- : int = 9
# x ;;
- : int = 36
```

1.1.5 Fonctions

Une fonction est déclarée en utilisant le mot-clé `let`, et sa signature s'écrit :

```
fun_name(pf_1, ..., pf_n : t_1 * ...* t_n) : t'_1 * ... * t'_k
```

où `(pf_1, ... , pf_n)`, de type `t_1 * ... * t_n`, est le paramètre formel de la fonction, `t'_1 * ... * t'_k` est le type de son résultat, et `t_1 * ... * t_n -> t'_1 * ... * t'_k` est le type de la fonction, par exemple :

```
# let add_diff(a, b : int * int) : int * int =
    (a + b, a - b)
;;
val add_diff : int * int -> int * int = <fun>
# let (sum, diff) : int * int = add_diff(5, 4) ;;
val sum : int = 9
val diff : int = 1
```

Comme pour les variables, le mot-clé `val` indique la déclaration d'un nom, par exemple `val add_diff : int * int -> int * int = <fun>` indique la déclaration de la fonction de nom `add_diff`, de type `int * int -> int * int` et dont la valeur, c'est-à-dire sa définition, est symbolisée par `<fun>`.

En `OCaml`, il existe de nombreuses fonctions prédéfinies, par exemple des fonctions de conversion de types :

`int_of_float : float → int`	`float_of_int : int → float`
`int_of_char : char → int`	`char_of_int : int → char`
`string_of_int : int → string`	`string_of_float : float → string`

1.1.6 Expression conditionnelle

Une expression conditionnelle est caractérisée par les mots-clés :

`if` *boolean_expression*
`then` *expression_of_type_t*
`else` *expression_of_same_type_t*

par exemple :

```
# let min2int(a, b : int * int) : int =
    if a < b
    then a
    else b
;;
val min2int : int * int -> int = <fun>
# min2int(4, 3) ;;
- : int = 3
```

Gestion des préconditions : lorsqu'une fonction n'est pas définie pour toutes les valeurs de son paramètre, nous utilisons la fonction `failwith`, dont le paramètre est une chaîne de caractères. L'exécution de cette fonction entraîne la levée d'une erreur et l'affichage de la chaîne de caractères, par exemple :

```
# let div_int(a, b : int * int) : int =
    if b = 0
    then failwith("erreur div_int : division par 0")
    else a / b
;;
val div_int : int * int -> int = <fun>
# div_int(3, 0) ;;
Exception: Failure "erreur div_int : division par 0".
```

1.1.7 Composition de fonctions

Le corps d'une fonction peut bien sûr contenir, en plus de déclarations de variables locales, des appels à des fonctions définies antérieurement, par exemple [1] :

```
# let min3int(a, b, c : int * int * int) : int =
    let min_ab : int = min2int(a, b) in
      min2int(min_ab, c)
;;
val min3int : int * int * int -> int = <fun>
```

1. Les symboles `(*` et `*)` délimitent un commentaire en `OCaml`.

```
# min3int(4, 2, 5) ;;
- : int = 2
(* version alternative *)
# let min3int(a, b, c : int * int * int) : int =
    min2int(min2int(a, b), c)
;;
val min3int : int * int * int -> int = <fun>
```

La commande `#trace` permet de suivre des enchaînements d'appels de fonctions. La commande `#untrace` permet de supprimer l'effet de la commande `#trace`, par exemple :

```
# #trace min3int ;;
min3int is now traced.
# #trace min2int ;;
min2int is now traced.
# min3int(4, 2, 5) ;;
min3int <-- (4, 2, 5)
min2int <-- (4, 2)
min2int --> 2
min2int <-- (2, 5)
min2int --> 2
min3int --> 2
- : int = 2
```

Ce sont des commandes de l'interpréteur `ocaml`, et non des fonctionnalités du langage `OCaml`. La syntaxe est :

`#trace` *function_name* `;;`	`#untrace` *function_name* `;;`

À noter qu'il est possible de déclarer des fonctions *locales*[1], de manière similaire à la déclaration de variables locales.

1.1.8 Fonctions récursives

La déclaration d'une fonction récursive est caractérisée par le mot-clé `rec`. Par exemple, la fonction `pow` calcule x^n pour un entier donné `x` et un entier positif `n` :

```
# let rec pow_aux(x, n : int * int) : int =
    if n = 0
    then 1
    else x * pow_aux(x, n-1)
;;
val pow_aux : int * int -> int = <fun>
# let pow(x, n : int * int) : int =
    if (n < 0) || ((x = 0) && (n = 0))
    then failwith("erreur pow : parametres invalides")
```

1. Cette possibilité n'est généralement pas utilisée dans ce livre, la commande `#trace` ne permettant pas de tracer l'exécution de fonctions locales.

```
    else pow_aux(x, n)
;;
val pow : int * int -> int = <fun>
# #trace pow_aux ;;
pow_aux is now traced.
# pow(3, 4) ;;
pow_aux <-- (3, 4)
pow_aux <-- (3, 3)
pow_aux <-- (3, 2)
pow_aux <-- (3, 1)
pow_aux <-- (3, 0)
pow_aux --> 1
pow_aux --> 3
pow_aux --> 9
pow_aux --> 27
pow_aux --> 81
- : int = 81
```

La fonction ci-dessous calcule le même résultat, mais en récursivité *terminale*[1] :

```
# let rec pow_aux(x, n, ind, res : int * int * int * int) : int =
    if ind = n
    then res
    else pow_aux(x, n, ind + 1, x * res)
;;
val pow_aux : int * int * int * int -> int = <fun>
# let pow(x, n : int * int) : int =
    if (n < 0) || ((x = 0) && (n = 0))
    then failwith("erreur pow : parametres invalides")
    else pow_aux(x, n, 0, 1)
;;
val pow : int * int -> int = <fun>
# #trace pow_aux ;;
pow_aux is now traced.
# pow(3, 4) ;;
pow_aux <-- (3, 4, 0, 1)
pow_aux <-- (3, 4, 1, 3)
pow_aux <-- (3, 4, 2, 9)
pow_aux <-- (3, 4, 3, 27)
pow_aux <-- (3, 4, 4, 81)
pow_aux --> 81
pow_aux --> 81
pow_aux --> 81
pow_aux --> 81
```

1. Schématiquement, une fonction est récursive terminale si, quel que soit le cas d'exécution, soit il n'y a aucun appel récursif, soit la fonction s'appelle elle-même une fois et, dans ce cas, il s'agit de la dernière opération effectuée (pour plus de précisions, voir le chapitre 7).

```
pow_aux --> 81
- : int = 81
```

1.1.9 Listes

Une liste peut être considérée comme une « suite » de valeurs du même type, par exemple :

```
# [3; 6; 1] ;;
- : int list = [3; 6; 1]
# ["hello"; "world"] ;;
- : string list = ["hello"; "world"]
# [] ;;
- : 'a list = []
```

Le type d'une liste est `type_name list`, où `type_name` est le type des éléments de la liste. En `OCaml`, `'a` désigne n'importe quel type : dans l'exemple ci-dessus, comme la liste `[]` est la liste vide, l'interpréteur ne peut définir un type précis pour ses éléments, puisqu'il n'y en a pas.

Deux listes peuvent être comparées en utilisant les comparateurs usuels. La comparaison utilise l'ordre lexicographique, par exemple :

```
# let m1 : int list =  [4; 0; -1; 7] ;;
val m1 : int list = [4; 0; -1; 7]
# let m2 : int list = [] ;;
val m2 : int list = []
# let m3 : int list = [4; 0; 0] ;;
val m3 : int list = [4; 0; 0]
# m2 < m1 ;;
- : bool = true
# m2 < m3 ;;
- : bool = true
# m1 < m3 ;;
- : bool = true
# m2 = [] ;;
- : bool = true
```

Dans cet ouvrage, les listes sont manipulées par les fonctions utilitaires suivantes (*cf.* annexe page 487). La longueur d'une liste est son nombre d'éléments, retourné par la fonction `len`.

```
len(x : 'a list) : int
```

par exemple :

```
# len(m1) ;;
- : int = 4
# len(m2) ;;
- : int = 0
# len(m3) ;;
- : int = 3
```

La fonction `empty` retourne une liste vide, et la fonction `isempty` teste si une liste est vide ou non.

`empty() : 'a list`	`isempty(l : 'a list) : bool`

par exemple :

```
# empty() ;;
- : 'a list = []
# isempty(empty()) ;;
- : bool = true
```

L'accès aux éléments utilise les fonctions `fst` (accès au premier élément, s'il existe), `lst` (accès au dernier élément, s'il existe), `nth` (accès à l'élément d'indice `k`, s'il existe, où `k` est compris entre `0` et la longueur de la liste diminuée d'une unité).

`fst(l : 'a list) : 'a`
`lst(l : 'a list) : 'a`
`nth(l, k : 'a list * int) : 'a`

par exemple :

```
# fst([3; 2; -5; 1] );;
- : int = 3
# lst([3; 2; -5; 1]) ;;
- : int = 1
# nth([3; 2; -5; 1],2) ;;
- : int = -5
```

Le calcul d'une liste contenant un élément `e` ainsi que les éléments d'une autre liste `l` utilise les fonctions `add_fst` (`e` est le premier élément de la liste résultat), `add_lst` (`e` est le dernier élément de la liste résultat), `add_nth` (`e` est l'élément d'indice `k` de la liste résultat, où `k` est compris entre `0` et `len(l)`).

`add_fst(l, e : 'a list * 'a) : 'a list`
`add_lst(l, e : 'a list * 'a) : 'a list`
`add_nth(l, e, k : 'a list * 'a * int) : 'a list`

par exemple :

```
# add_fst([3; 2; -5; 1], 7) ;;
- : int list = [7; 3; 2; -5; 1]
# add_lst([3; 2; -5; 1], 9) ;;
- : int list = [3; 2; -5; 1; 9]
# add_nth([3; 2; -5; 1], 9, 2) ;;
- : int list = [3; 2; 9; -5; 1]
```

Le calcul d'une liste contenant un élément de moins qu'une liste `l` utilise les fonctions `rem_fst` (la liste résultat ne contient pas le premier élément de la liste `l`), `rem_lst` (la liste résultat ne contient pas le dernier élément de la liste

`l`), `rem_nth` (la liste résultat ne contient pas l'élément d'indice `k` de la liste `l`, où `k` est compris entre `0` et `len(l) - 1`).

`rem_fst(l : 'a list) : 'a list`
`rem_lst(l : 'a list) : 'a list`
`rem_nth(l, k : 'a list * int) : 'a list`

par exemple :

```
# rem_fst([3; 2; -5; 1]) ;;
- : int list = [2; -5; 1]
# rem_lst([3; 2; -5; 1]) ;;
- : int list = [3; 2; -5]
# rem_nth([3; 2; -5; 1], 2) ;;
- : int list = [3; 2; 1]
```

À noter que, de même que calculer l'expression `x + 1` ne change pas la valeur de l'entier `x`, l'application d'une des opérations ci-dessus à une liste ne change pas la valeur de la liste : les listes sont non mutables.

La fonction ci-dessous calcule la somme des éléments d'une liste d'entiers :

```
# let rec sum_int_list(l : int list) : int =
    if isempty(l)
    then 0
    else
      let (f, r) : int * int list = (fst(l), rem_fst(l)) in
        f + sum_int_list(r)
;;
val sum_int_list : int list -> int = <fun>
# sum_int_list([3; 2; -5; 1]) ;;
- : int = 1
# let l : int list = [4; -3; -1] ;;
val l : int list = [4; -3; -1]
# sum_int_list(l) ;;
- : int = 0
# l ;;
- : int list = [4; -3; -1]
```

1.1.10 Variables mutables, affectation

Une valeur d'un type *mutable* `type_name ref` peut être considérée comme un *contenant* pouvant contenir une valeur de type `type_name` (un type mutable est caractérisé par le mot `ref`). Une variable d'un type mutable, ou *variable mutable*, peut être considérée comme une variable d'un langage de programmation impératif. Les opérateurs sont :

Opérateur	Action
`!`	accès au contenu
`:=`	affectation

La valeur d'une expression d'affectation est la valeur « vide » `()` du type `unit`. Par la suite, nous appelons *instruction* une expression dont le résultat est de type `unit`.

```
# ref 8 ;;
- : int ref = {contents = 8}
# let x : int ref = ref 10 ;;
val x : int ref = {contents = 10}
# !x ;;
- : int = 10
# x := !x + 3 ;;
- : unit = ()
# !x ;;
- : int = 13
# let y : int = !x ;;
val y : int = 13
# y = x ;;
Error: This expression has type int ref
   but an expression was expected of type int
```

1.1.11 Séquence

Une séquence (ou bloc) est une suite d'expressions séparées par des points-virgules. Par convention, si la séquence contient plus d'une expression, nous la délimitons par des parenthèses ; toutes les expressions d'une séquence sont des instructions, sauf éventuellement la dernière, qui définit la valeur et le type de la séquence, par exemple :

```
# let (a, b) : int ref * int ref = (ref 3, ref 7) ;;
val a : int ref = {contents = 3}
val b : int ref = {contents = 7}
# let exchange(x, y : 'a ref * 'a ref) : unit =
    let z : 'a = !x in
      (
      x := !y ;
      y := z
      )
;;
val exchange : 'a ref * 'a ref -> unit = <fun>
# exchange(a, b) ;;
- : unit = ()
# (a, b) ;;
- : int ref * int ref = ({contents = 7}, {contents = 3})
```

Le parenthésage permet aussi de contrôler la portée des variables [1] :

```
# let x : int ref = ref 3 ;;
val x : int ref = {contents = 3}
# let z : int = 8 in
    (
    x := !x + z ;
    z
    )
;;
- : int = 8
# x ;;
- : int ref = {contents = 11}
# (
  (let z : int = 8 in x := !x + z) ;
  z
  )
;;
Error: Unbound value z
```

où `Unbound value z` signifie que le nom `z` est inconnu. En fait, la variable locale `z` est déclarée et connue uniquement dans l'expression parenthésée :

```
(let z : int = 8 in x := !x + z)
```

Elle est inconnue en dehors de cette expression parenthésée.

1.1.12 Entrées-sorties

Les fonctions ci-dessous permettent d'effectuer des entrées-sorties.

Signature	**affichage à l'écran**
`print_int` : int -> unit	d'un entier
`print_char` : char -> unit	d'un caractère
`print_float` : float -> unit	d'un flottant
`print_string` : string -> unit	d'une chaîne de caractères
`print_newline` : unit -> unit	d'un saut de ligne

Signature	**lecture au clavier**
`read_line` : unit -> string	d'une chaîne de caractères
`read_int` : unit -> int	d'un entier
`read_float` : unit -> float	d'un flottant
`read_char` : unit -> char	d'un caractère

La fonction `read_char` n'est pas prédéfinie en `OCaml`, c'est une fonction utilitaire : *cf.* annexe page 487.

1. Ceci est parfois nécessaire : par exemple en `OCaml`, une déclaration locale est *a priori* suivie d'une séquence, et non d'une unique expression.

1.1.13 Instructions itératives

La syntaxe des instructions itératives, appelées aussi *boucles*, est :

for *index_name* = *start_value* to *end_value* do   *sequence_of_instructions* done ;
while *boolean_expression* do   *sequence_of_instructions* done ;

L'instruction `for` existe aussi avec `downto`. Voici quelques exemples :

```
(* calcul de factorielle : version iterative *)
let fact_iter(n : int) : int =
  if n < 0
  then failwith("erreur fact_iter : parametre negatif")
  else
    (
    let f : int ref = ref 1 in
      (
      for i = 1 to n
      do f := i * !f
      done ;
      !f
      )
    )
;;
let print_alphabet() : unit =
  let lower_a : int = int_of_char('a')
    and upper_a : int = int_of_char('A')
  in
    (
    for i = 0 to 25
    do print_char(char_of_int(i + lower_a))
    done ;
    print_newline() ;
    for i = 25 downto 0
    do print_char(char_of_int(i + upper_a))
    done ;
    print_newline()
    )
;;
let read_lower_case() : char =
  let c : char ref = ref '1' and the_end : bool ref = ref false in
```

```
    (
    while not(!the_end)
    do
      print_string("Veuillez saisir une lettre minuscule, svp : ") ;
      c := read_char() ; print_newline() ;
      the_end := ((!c >= 'a') && (!c <= 'z'))
    done ;
    !c
    )
;;
```

1.1.14 Tableaux

Un tableau peut être considéré comme une « suite » de valeurs du même type, par exemple :

```
# [| 3 ; 9 ; 10 ; 2 |] ;;
- : int array = [|3; 9; 10; 2|]
# [||] ;;
- : 'a array = [||]
# let v : int array = [|4 ; 10 ; 3 |] ;;
val v : int array = [|4; 10; 3|]
```

Le type d'un tableau est `type_name array`, où `type_name` est le type des éléments du tableau. Un tableau est manipulé via les fonctions et opérateurs suivants :

Opérateur	**Action**
`arr_make : int * 'a -> 'a array`	création
`arr_len : 'a array -> int`	longueur
`v.(i)`	accès
`v.(i) <- x`	affectation

La fonction `arr_make` a pour résultat un tableau dont toutes les cases contiennent la même valeur ; il est donc nécessaire en général d'*initialiser un tableau après sa création*. Les cases du tableau sont indicées à partir de `0` ; l'indice maximal est donc égal à la longueur du tableau diminuée d'une unité. Par exemple :

```
# let v : int array = arr_make(5, 0) ;;
val v : int array = [|0; 0; 0; 0; 0|]
# arr_len(v) ;;
- : int = 5
# for i = 0 to 4
  do v.(i) <- 2 * i
  done ;;
- : unit = ()
```

```
# v ;;
- : int array = [|0; 2; 4; 6; 8|]
# v.(5) ;;
Exception: Invalid_argument "index out of bounds".
```

Contrairement à une liste, un tableau est mutable, dans la mesure où il est possible de modifier les valeurs contenues dans le tableau ; en revanche, il n'est pas possible de modifier le nombre d'éléments d'un tableau.

Pour la manipulation de tableaux bidimensionnels, on peut utiliser [1] :

Type	`'a matrix`
Opérateur	**Action**
`mat_make : int * int * 'a -> 'a matrix`	création
`arr_len(m)`	nombre de lignes
`arr_len(m.(0))`	nombre de colonnes
`m.(i).(j)`	accès
`m.(i).(j) <- v`	affectation

La fonction `mat_make(ln_x, ln_y, v : int * int * 'a)` a pour résultat une matrice de `ln_x` lignes et `ln_y` colonnes, dont toutes les cases contiennent la valeur `v`. Il est donc nécessaire en général d'initialiser une matrice après sa création. En `OCaml`, une matrice est en fait un tableau de tableaux.

Voici des exemples de manipulation de tableaux :

```
(* recherche de l'indice de la premiere valeur negative *)
# let find_neg_in_array(myarr : int array) : int * bool =
    let max_ind : int = arr_len(myarr) - 1
      and found : bool ref = ref false and i : int ref = ref (-1)
    in
    let the_end : bool ref = ref (max_ind < 0) in
      (
      while not(!the_end)
      do
        if !i >= max_ind
        then the_end := true
        else
          (
          i := !i + 1 ;
          found := (myarr.(!i) < 0) ;
          the_end := !found
          );
      done ;
```

1. L'expression définissant le nombre de colonnes n'est définie que si le nombre de lignes est non nul.

```
      (!i, !found)
      )
;;
val find_neg_in_array : int array -> int * bool = <fun>
# find_neg_in_array([|3; 8; 2; -4; 5; 7; -1|]) ;;
- : int * bool = (3, true)

(* calcul du triangle de pascal dans une matrice carree *)
# let pascal(n : int) : int matrix =
  let m_pas : int matrix = mat_make(n + 1, n + 1, 0) in
    (
    for i = 0 to n
    do
      m_pas.(i).(0) <- 1 ;
      for j = 1 to i
      do m_pas.(i).(j) <- m_pas.(i-1).(j-1) + m_pas.(i-1).(j)
      done
    done ;
    m_pas
    )
;;
val pascal : int -> int matrix = <fun>
# pascal(6) ;;
- : int matrix =
[|[|1; 0; 0; 0; 0; 0; 0|];
  [|1; 1; 0; 0; 0; 0; 0|];
  [|1; 2; 1; 0; 0; 0; 0|];
  [|1; 3; 3; 1; 0; 0; 0|];
  [|1; 4; 6; 4; 1; 0; 0|];
  [|1; 5; 10; 10; 5; 1; 0|];
  [|1; 6; 15; 20; 15; 6; 1|]|]
```

Comme dit ci-dessus, il est en général nécessaire d'initialiser un tableau une fois celui-ci créé par l'application de la fonction `arr_make` ; cette remarque est valable aussi lors de la création d'une matrice par la fonction `mat_make`. En effet, *toutes les cases du tableau ou de la matrice ont exactement le même contenu*, ce qui n'est en général pas ce qui est souhaité. En particulier, si le contenu des cases est mutable, ou possède un composant mutable, la modification de la valeur contenue dans cette partie mutable affecte toutes les cases, puisqu'elles ont toutes le même contenu, par exemple :

```
# type t_person = {name : string ; age : int ref} ;;
type t_person = { name : string; age : int ref; }
# let my_people : t_person array =
         arr_make(3, {name = "myname" ; age = ref 10}) ;;
val my_people : t_person array =
  [|{name = "myname"; age = {contents = 10}};
```

```
    {name = "myname"; age = {contents = 10}};
    {name = "myname"; age = {contents = 10}}|]
# my_people.(1) ;;
- : t_person = {name = "myname"; age = {contents = 10}}
# my_people.(1).age ;;
- : int ref = {contents = 10}
# my_people.(1).age := 7 ;;
- : unit = ()
# my_people ;;
- : t_person array =
[|{name = "myname"; age = {contents = 7}};
  {name = "myname"; age = {contents = 7}};
  {name = "myname"; age = {contents = 7}}|]
```

1.1.15 Quelques conventions d'écriture (liste non exhaustive)

Les identificateurs : pour les noms de variables, de fonctions, *etc.*, nous utilisons le plus souvent des noms d'identificateurs explicites, écrits en minuscules et en anglais ; si l'identificateur nécessite plusieurs mots, ils sont en général séparés par un underscore.

Les noms de type [1] sont préfixés par `t_`, par exemple `t_point`, `t_vector`.

Les valeurs de types énumérés sont écrites entièrement en majuscules [2].

Les espaces sont placés avant et après un symbole d'opération ; les séparateurs, comme la virgule ou le point-virgule, sont suivis par un espace.

Une variable est toujours définie avec son type et sa valeur.

La fonction `failwith` est écrite avec un message d'erreur de la forme `erreur nom_de_fonction : description_de_l'erreur`, indiquant la fonction dans laquelle l'erreur s'est produite, ainsi que la cause probable de celle-ci.

Les commentaires : si besoin, une fonction, ou une déclaration de type, est précédée d'un commentaire qui en précise certains aspects : `(* ceci est un commentaire OCaml *)` ;

Les indentations sont aussi utilisées pour structurer les expressions, les fonctions, *etc.* En particulier, les mots-clés `for` (resp. `while`), `do`, `done` sont alignés, de même que `if`, `then`, `else`.

Les fonctions locales ne sont généralement pas utilisées, car leurs exécutions ne sont pas tracées lors de l'utilisation de la commande `#trace`.

Les séquences sont généralement parenthésées, sauf si elles sont déjà délimitées, par exemple par les mots-clés `do` et `done` d'une instruction itérative.

---

1. Notons que les types *produit* peuvent ne pas être déclarés explicitement en `OCaml` ; néanmoins, les déclarer permet de les nommer, ce qui peut faciliter la lecture du code.

2. En `OCaml`, seule la première lettre doit être une majuscule.

1.2 Démarche de conception - développement

1.2.1 Présentation

Étant donné un problème « simple » que l'on souhaite résoudre par un calcul, la démarche suivie est essentiellement la suivante :

1. *identifier la signature* de la fonction :
   — ceci nécessite d'identifier clairement le résultat et le paramètre de la fonction : pour ce faire, il est souvent nécessaire de *concrétiser* le problème par un ou plusieurs exemples. Notons que le résultat, de même que le paramètre, peut être composé de zéro, une ou plusieurs valeurs ;
   — une fois ces valeurs identifiées, on peut commencer à *formaliser*, au minimum préciser les *types* du résultat et du paramètre afin de définir la signature de la fonction. Il est aussi nécessaire de *nommer* les valeurs composant le paramètre, afin de pouvoir les manipuler ;
2. *concevoir un algorithme* permettant de calculer le résultat à partir du paramètre :
   — tous les coups sont permis : adapter un algorithme connu, traduire un raisonnement en algorithme, déduire un algorithme d'une expérimentation sur un ou plusieurs exemples, *etc.*
   — attention : suivant le problème posé, le nombre d'algorithmes pouvant calculer le résultat attendu à partir du paramètre donné peut être zéro, un ou plusieurs... Pour les problèmes étudiés dans le cadre de ce livre, il existe toujours au moins un algorithme ;
3. *vérifier la correction* de l'algorithme :
   — il existe des méthodes et des outils permettant de *prouver* un algorithme, mais ceci dépasse le cadre de ce livre. Néanmoins, il est toujours utile, pour un algorithme donné, de chercher des arguments permettant de conclure sur sa correction ;
   — la vérification est donc faite ici en *testant* l'algorithme. Il est nécessaire de savoir tester un algorithme aussi bien *manuellement* que *sur machine*. Bien sûr, des tests ne constituent pas une preuve en règle générale, mais des tests bien choisis permettent souvent de détecter des erreurs ;
4. *estimer la complexité* de l'algorithme ; en fait, nous nous focalisons ici sur le coût en espace et en temps de l'algorithme. Schématiquement, le coût en espace correspond à la place mémoire utilisée lors de l'exécution de l'algorithme, et le coût en temps mesure le temps mis pour son exécution ;
5. *implanter l'algorithme* dans un langage de programmation choisi. Comme nous utilisons un même langage comme langage d'algorithmique et comme

langage de programmation, seules sont étudiées dans ce livre les implantations de structures fondamentales comme les piles et les files, les arbres et les graphes.

1.2.2 Exemple

1.2.2.1 Énoncé du problème

Il est prévu d'organiser un examen sur table dans deux amphithéatres `A1` et `A2`, situés côte à côte dans le même bâtiment. La procédure prévoit que les étudiants sont appelés selon l'ordre croissant de leurs numéros d'étudiants. Pour chaque amphithéatre, les surveillants disposent de la liste des numéros concernés, dans l'ordre d'appel. Le jour de l'examen, une fuite d'eau empêche d'utiliser l'amphithéatre `A2`. Comme aucun autre amphithéatre n'est disponible, il est décidé, pour éviter de reporter l'examen, d'accueillir tous les étudiants concernés dans l'amphithéatre `A1`, qui, heureusement, peut les accueillir tous. Pour respecter la procédure d'appel des étudiants, il est donc nécessaire de reconstituer la liste complète des numéros des étudiants, dans l'ordre, à partir des deux listes d'appel prévues initialement.
La question posée est la suivante : comment simuler cette reconstitution sur machine ?

1.2.2.2 Identification de la signature de la fonction

Il est demandé de simuler, c'est-à-dire d'écrire un algorithme permettant de mimer le processus de reconstitution. Dans l'énoncé, le résultat est clairement défini : il s'agit de la liste complète des numéros des étudiants. On déduit facilement que les seules informations nécessaires pour calculer cette liste complète sont les listes partielles initialement prévues pour les deux amphithéatres.

Pour concrétiser le problème, nous choisissons par exemple les listes partielles suivantes [1] : `[1; 5; 9]` et `[2; 3; 6; 8]`.
Le résultat est donc : `[1; 2; 3; 5; 6; 8; 9]`.
Pour être complet, il faudrait considérer d'autres exemples, en particulier où l'une ou l'autre des listes est vide.

1. Nous choisissons des listes d'entiers, car dans l'énoncé :
   — il n'y a aucune indication sur la constitution d'un numéro d'étudiant ;
   — la seule propriété importante que présentent ces numéros est le fait qu'ils sont ordonnés.

Les entiers, ordonnés par nature, peuvent donc être utilisés pour simuler les numéros indiqués dans l'énoncé du problème.

Le paramètre est un couple de listes d'entiers, le résultat est une liste d'entiers ; la signature de la fonction (que l'on choisit d'appeler ici "merge") est : `merge(stud_A1, stud_A2 : int list * int list) : int list`.

1.2.2.3 Conception d'un algorithme

L'algorithme ci-dessous provient des remarques suivantes :

1. la première valeur du résultat est la plus petite des premières valeurs des deux listes. Sur l'exemple `merge([1; 5; 9],[2; 3; 6; 8])`, on voit que la valeur `1` est la plus petite des premières valeurs des deux listes (dit autrement : `1 < 2`), et cette valeur `1` se retrouve être la première valeur de la liste résultat ;
2. le reste de la liste résultat est simplement la « reconstitution » d'une liste ordonnée à partir des valeurs restantes des deux listes paramètres ; c'est ici que l'on voit apparaître la récursivité de l'algorithme. Sur l'exemple, il s'agit de `[2; 3; 5; 6; 8; 9] = merge([5; 9],[2; 3; 6; 8])`.

Il faut bien sûr faire attention aux cas où il n'est pas possible de comparer les premières valeurs des deux listes, c'est-à-dire aux cas (basiques) où une liste est vide. Une manière d'écrire la fonction est :

```
let rec merge(stud_A1, stud_A2 : int list * int list) : int list =
  if isempty(stud_A1)
  then stud_A2
  else
    if isempty(stud_A2)
    then stud_A1
    else
      let (f1, r1) : int * int list = (fst(stud_A1), rem_fst(stud_A1))
          and (f2, r2) : int * int list
                                = (fst(stud_A2), rem_fst(stud_A2))
      in
        if f1 < f2
        then add_fst(merge(r1, stud_A2), f1)
        else add_fst(merge(stud_A1, r2), f2)
;;
```

1.2.2.4 Vérification de la correction

Une première vérification consiste à tester manuellement la fonction sur l'exemple choisi (une exécution manuelle permet souvent de repérer facilement des erreurs d'analyse ou d'écriture) :

• `merge([1; 5; 9],[2; 3; 6; 8])` `= [1; 2; 3; 5; 6; 8; 9]`

```
-> 1 < 2 => add_fst(merge([5; 9], [2; 3; 6; 8]), 1)
```

```
• merge([5; 9], [2; 3; 6; 8])          = [2; 3; 5; 6; 8; 9]
-> 2 < 5 => add_fst(merge([5; 9], [3; 6; 8]), 2)
   • merge([5; 9], [3; 6; 8])          = [3; 5; 6; 8; 9]
   -> 3 < 5 => add_fst(merge([5; 9], [6; 8]), 3)
      • merge([5; 9], [6; 8])          = [5; 6; 8; 9]
      -> 5 < 6 => add_fst(merge([9], [6; 8]), 5)
         • merge([9], [6; 8])          = [6; 8; 9]
         -> 6 < 9 => add_fst(merge([9], [8]), 6)
            • merge([9], [8])          = [8; 9]
            -> 8 < 9 => add_fst(merge([9], []), 8)
               • merge([9], [])        = [9]
```

L'ordre d'exécution se lit dans un premier temps du haut vers le bas (partie en noir). Elle retrace pour chaque appel de fonction, par exemple celui décrit sur la première ligne, ce qui est exécuté, en l'occurence ce qui est décrit sur la deuxième ligne. Arrivé au dernier appel, le résultat est retourné directement, ce qui va permettre de donner les résultats de chaque appel (du bas vers le haut, en gris), jusqu'au résultat du premier appel. On peut voir aussi qu'à un moment de l'exécution, l'une des deux listes paramètre est vide, ce qui permet de vérifier aussi ce cas. Du fait de la symétrie des deux listes du paramètre, il est inutile ici de dérouler manuellement d'autres tests.

1.2.2.5 Évaluation de la complexité

Les notions de base relatives à la complexité, que nous utilisons dans ce livre, sont précisées dans le chapitre 2. Les étapes d'évaluation de la complexité de la fonction `merge` sont décrites ici succinctement.

1. En toute généralité, la complexité de la fonction `merge` est dépendante des longueurs des listes paramètres, et des valeurs qui y sont contenues. Soient `l1 = len(stud_A1)`, `l2 = len(stud_A2)` et `n = l1 + l2` : le nombre d'appels récursifs engendrés lors de l'évaluation du résultat de la fonction est *au pire* `n`. En effet, chaque appel engendre un appel sur des listes qui contiennent un élément de moins : le nombre d'éléments décroit donc de 1 en 1 depuis la valeur initiale `n` jusque (au pire) `1`, ce qui est le cas dans l'exemple ci-dessus. Le nombre d'appels est *au mieux* égal à `min(l1, l2)`. En effet, les appels cessent quand l'une des deux listes a été entièrement parcourue : le nombre d'appels est donc au mieux égal à la longueur de la plus petite des listes, ce qui est le cas quand tous les éléments d'une liste sont inférieurs au plus petit des éléments de l'autre liste, par exemple `merge([1; 3; 4], [7; 8; 10; 12; 15])`.

2. La *complexité en espace* est *au pire* proportionnelle à la somme des longueurs de toutes les listes partielles, si, à une étape de l'exécution, toutes les informations concernant les appels précédents ont été mémorisées (*cf.* l'exemple d'exécution). La complexité en espace est donc *au pire* proportionnelle à $\sum_{i=1}^{n} i$, ce qui est égal à $n*(n+1)/2$. La complexité en espace est *au mieux* proportionnelle à `n`, si seules les listes initiales sont mémorisées, en utilisant le fait que les listes partielles sont contenues dans les listes initiales.
3. Si l'on considère que les opérations de base sont exécutées en temps constant, la *complexité en temps* de la fonction `merge` est proportionnelle au nombre d'appels, c'est-à-dire au pire à `n` et au mieux à `min(l1, l2)`.

Remarque : il s'agit ici d'un problème très classique de *fusion de listes ordonnées* par ordre croissant. L'algorithme est adaptable très facilement pour traiter des listes ordonnées de tous types, voire des listes ordonnées par ordre décroissant.

1.2.2.6 Première variante

Une autre idée pour le calcul consiste à considérer 3 listes : initialement, les deux premières sont les listes paramètre, la troisième liste est vide. Une étape du calcul consiste à simuler le "transfert" de la plus petite des premières valeurs des deux premières listes vers la troisième liste.

La signature de la fonction `merge` a été identifiée lors de la première phase d'analyse du problème, elle n'a donc pas à être modifiée parce qu'on imagine un autre mode de calcul : il est alors nécessaire d'introduire une fonction auxiliaire [1]. Si l'on déroule l'exemple précédemment choisi, on obtient :

```
• merge([1; 5; 9], [2; 3; 6; 8])
-> merge_aux([1; 5; 9], [2; 3; 6; 8], [])
   • merge_aux([1; 5; 9], [2; 3; 6; 8], [])
   -> 1 < 2 => merge_aux([5; 9], [2; 3; 6; 8], [1])
      • merge_aux([5; 9], [2; 3; 6; 8], [1])
      -> 2 < 3 => merge_aux([5; 9], [3; 6; 8], [1; 2])
         • merge_aux([5; 9], [3; 6; 8], [1; 2])
         -> 3 < 5 => merge_aux([5; 9], [6; 8], [1; 2; 3])
            • merge_aux([5; 9], [6; 8], [1; 2; 3])
            -> 5 < 6 => merge_aux([9], [6; 8], [1; 2; 3; 5])
               • merge_aux([9], [6; 8], [1; 2; 3; 5])
               -> 6 < 9 => merge_aux([9], [8], [1; 2; 3; 5; 6])
```

1. Cette fonction `merge_aux` pourrait être locale à la fonction `merge`. Elle est ici déclarée comme une fonction à part entière afin de pouvoir en tracer l'exécution.

et ainsi de suite. Les fonctions peuvent alors être écrites [1] :

```
let rec merge_aux(stud_A1, stud_A2, res :
                     int list * int list * int list) : int list =
  if isempty(stud_A1)
  then concat(res, stud_A2)
  else
    if isempty(stud_A2)
    then concat(res, stud_A1)
    else
      let (f1, r1) : int * int list = (fst(stud_A1), rem_fst(stud_A1))
          and (f2, r2) : int * int list
                                = (fst(stud_A2), rem_fst(stud_A2))
      in
        if f1 < f2
        then merge_aux(r1, stud_A2, add_lst(res, f1))
        else merge_aux(stud_A1, r2, add_lst(res, f2))
;;
let merge(stud_A1, stud_A2 : int list * int list) : int list =
  merge_aux(stud_A1, stud_A2, empty())
;;
```

On peut noter que le nombre d'appels récursifs est le même que celui de la première version. Le nombre d'éléments des listes paramètres est `n` à chaque appel récursif (les trois listes paramètres contiennent tous les éléments des deux listes initiales). Avec le même raisonnement que précédemment, la complexité en espace est donc au pire proportionnelle à `n` multiplié par le nombre d'appels récursifs, c'est-à-dire au pire à n^2. Comme précédemment, la complexité en temps dépend aussi des complexités des opérations et des fonctions de base utilisées, qui sont étudiées dans le chapitre suivant.

1.2.2.7 Deuxième variante

Cette variante fusionne des tableaux dont les éléments sont ordonnés par ordre croissant. L'algorithme est très proche de la variante précédente : simplement, il est généralisé pour ne tenir compte que de *parties* des tableaux paramètres, compris entre des indices donnés (on suppose que le tableau résultat est assez grand pour contenir les éléments à fusionner).

La fonction `copy` recopie la partie du tableau `myarr` comprise entre les indices `min_ind` et `max_ind` dans le tableau `res` à partir de l'indice `r_ind`.

La fonction `merge` fusionne la partie du tableau `myarr_1` comprise entre les indices `min_ind_1` et `max_ind_1` et la partie du tableau `myarr_2` comprise entre les indices `min_ind_2` et `max_ind_2` dans le tableau `res` à partir de l'indice

1. La fonction `concat : 'a list * 'a list -> 'a list` est une fonction utilitaire dont le résultat est la concaténation des deux listes paramètres : *cf.* annexe page 487.

`r_ind`. Les variables `i1`, `i2` et `ir` sont utilisées pour parcourir les tableaux `myarr_1`, `myarr_2` et `res`. Lors d'un passage dans la boucle `while`, les éléments de `myarr_1` d'indices compris entre `ind_min_1` (inclus) et `!i1` (exclu) ont déjà été copiés dans le tableau `res` ; il en est de même pour les éléments de `myarr_2` d'indices compris entre `ind_min_2` (inclus) et `!i2` (exclu) ; le plus petit des deux éléments `myarr_1.(!i1)` et `myarr_2.(!i2)` est alors ajouté à son tour au tableau `res`. La sortie de la boucle se fait quand `myarr_1` a été parcouru jusqu'à son indice `ind_max_1` ou quand `myarr_2` a été parcouru jusqu'à son indice `ind_max_2`.

Les appels suivants de la fonction `copy` permettent d'ajouter au tableau `res` les éléments non encore parcourus [1].

Ces fonctions ne sont applicables que si certaines préconditions sont vérifiées, portant en particulier sur les valeurs respectives des indices et des longueurs des tableaux. Par exemple, la fonction `copy` n'est applicable que si `0 <= min_ind`, `max_ind <= arr_len(myarr) - 1`, `0 <= r_ind` et `r_ind + max_ind - min_ind <= arr_len(res) - 1`. On suppose que les fonctions ne sont appelées que si ces préconditions sont vérifiées.

```
let copy(myarr, min_ind, max_ind, res, r_ind :
                   int array * int * int * int array * int) : unit =
  for i = 0 to (max_ind - min_ind)
  do res.(r_ind + i) <- myarr.(min_ind + i)
  done
;;
let merge(myarr_1, min_ind_1, max_ind_1,
          myarr_2, min_ind_2, max_ind_2,
          res, r_ind : int array * int * int * int array * int
                                  * int * int array * int) : unit =
  let i1 : int ref = ref min_ind_1 and i2 : int ref = ref min_ind_2
    and ir : int ref = ref r_ind and the_end : bool ref = ref false
  in
    (
    while not(!the_end)
    do
      if myarr_1.(!i1) < myarr_2.(!i2)
      then
        (
        res.(!ir) <- myarr_1.(!i1) ;
        i1 := !i1 + 1 ;
        the_end := (!i1 > max_ind_1)
        )
```

1. Par la suite, les algorithmes ne sont pas toujours expliqués. Il est conseillé d'expérimenter manuellement les algorithmes sur un ou plusieurs exemples choisis, afin de bien comprendre leurs fonctionnements.

```
      else
        (
        res.(!ir) <- myarr_2.(!i2) ;
        i2 := !i2 + 1 ;
        the_end := (!i2 > max_ind_2)
        ) ;
      ir := !ir + 1
    done ;
    copy(myarr_1, !i1, max_ind_1, res, !ir) ;
    copy(myarr_2, !i2, max_ind_2, res, !ir) ;
    )
;;
```

Voici ci-dessous des exemples d'exécutions. On peut noter que la fonction permet de fusionner des parties ordonnées d'un même tableau, et ce dans le même tableau : simplement, toutes ces parties doivent être disjointes (il est bien sûr supposé que ces vérifications sont faites avant l'appel de la fonction `merge`).

```
# let t1 : int array = [|1; 5; 9|] ;;
val t1 : int array = [|1; 5; 9|]
# let t2 : int array  = [|2; 3; 6; 8|] ;;
val t2 : int array = [|2; 3; 6; 8|]
# let t : int array  = arr_make(7, 0) ;;
val t : int array = [|0; 0; 0; 0; 0; 0; 0|]
# merge(t1, 0, 2, t2, 0, 3, t, 0) ;;
- : unit = ()
# t ;;
- : int array = [|1; 2; 3; 5; 6; 8; 9|]
# let other_t : int array  =
                    [|1; 5; 9; 2; 3; 6; 8; 0; 0; 0; 0; 0; 0; 0|] ;;
val other_t : int array = [|1; 5; 9; 2; 3; 6; 8; 0; 0; 0; 0; 0; 0; 0|]
# merge(other_t, 0, 2, other_t, 3, 6, other_t, 7) ;;
- : unit = ()
# other_t ;;
- : int array = [|1; 5; 9; 2; 3; 6; 8; 1; 2; 3; 5; 6; 8; 9|]
```

1.3 Types abstraits

1.3.1 Motivations

Comme son nom l'indique, la notion de *type abstrait* permet de définir un type sans faire référence à une implantation particulière dans un langage donné, ni même à un langage de programmation particulier.

Par opposition, un type, tel que défini et manipulé dans les sections précédentes, est un *type concret*.

Par exemple, la notion de *liste* peut être définie de manière générique par un type abstrait : *cf.* chapitre 3 page 139. Cette notion peut être implantée de différentes manières, en utilisant différents types concrets, par exemple le type `list` ou le type `array` en `OCaml` : *cf.* chapitre 6 page 351.

Notons que :

— certains types sont prédéfinis dans certains langages, mais ne le sont pas dans d'autres langages : par exemple, le type des listes est prédéfini en `OCaml`, mais pas en langage `C` ;
— même si un type est prédéfini dans un langage, il peut être parfois intéressant d'implanter ce type de manière différente dans ce même langage, par exemple pour privilégier l'efficacité de certaines opérations de manipulation du type. Par exemple, l'opération `nth`, qui permet d'accéder au $i^{ème}$ élément d'une liste, n'a pas la même efficacité suivant que les listes sont implantées avec le type `list` ou le type `array` en `OCaml`.
— en général, même si un type n'est pas prédéfini dans un langage de programmation, celui-ci offre différents mécanismes permettant d'implanter ce type. Par exemple, il est possible d'implanter les listes en langage `C` en utilisant des tableaux, ou encore des enregistrements et des pointeurs.

La notion de type abstrait permet donc de définir des types indépendamment des outils particuliers d'un langage de programmation donné. Ceci permet :

— de raisonner à différents niveaux d'abstraction, ce qui est toujours très utile, voire indispensable, pour *concevoir* des types et des algorithmes non triviaux ;
— de décrire des types et des algorithmes de manière abstraite, *indépendamment d'une implantation particulière*. Il est alors possible de *réutiliser* ces structures et algorithmes dans différents contextes. Simplement, dans un contexte donné, il est nécessaire de choisir une implantation particulière, la mieux adaptée possible à ce contexte. Et si le contexte vient à être modifié, il « suffit » de changer l'implantation sans avoir besoin de ré-étudier le problème depuis le début. Dit autrement, cette notion permet de mieux séparer *l'algorithmique* de la *programmation*, qui ont chacune leurs propres enjeux.

1.3.2 Exemples et définition

1.3.2.1 Les booléens

Il existe différentes manières de définir un type abstrait. Nous choisissons ici de définir le *comportement* du type via l'ensemble des *fonctions* permettant de le manipuler [1]. En particulier, les *valeurs du type ne sont pas définies en tant que telles*, mais comme les résultats de fonctions, appelées des ***constructeurs***. Par exemple, la définition du type abstrait `t_booleen` ci-dessous comporte deux constructeurs `vrai` et `faux` : ces deux fonctions n'ont pas de paramètre, et leur résultat est une valeur de type `t_booleen`.

Les ***autres fonctions*** que nous avons choisi de définir sont `non`, `et`, `ou`, dont les paramètres sont respectivement une valeur ou deux valeurs de type `t_booleen`, et dont les résultats sont de type `t_booleen`.

On indique pour chaque fonction, constructeurs compris, la *signature* de la fonction.

Pour être complet, il reste à définir la manière dont les fonctions *se comportent* les unes par rapport aux autres. Ces comportements sont décrits ici par des *formules logiques* appelées ***axiomes***, dans lesquelles peuvent intervenir en particulier les opérateurs = (égalité) et $\Leftrightarrow$ (équivalence logique).

NB. Pour distinguer l'écriture d'un type abstrait de l'écriture d'un code `OCaml`, tous les noms choisis lors de la définition d'un type abstrait sont en français, et non en anglais.

Type abstrait `t_booleen`
Constructeurs
- `vrai` : $\emptyset \rightarrow$ `t_booleen`
- `faux` : $\emptyset \rightarrow$ `t_booleen`

Autres opérations
- `non` : `t_booleen` $\rightarrow$ `t_booleen`
- `et` : `t_booleen` $\times$ `t_booleen` $\rightarrow$ `t_booleen`
- `ou` : `t_booleen` $\times$ `t_booleen` $\rightarrow$ `t_booleen`

Axiomes
- `non(vrai()) = faux()`
- `non(faux()) = vrai()`
- `et(vrai(), vrai()) = vrai()`
- `et(vrai(), faux()) = faux()`

1. Voir aussi le chapitre 4 du livre [6] et l'ouvrage [5].

- `et(faux(), vrai()) = faux()`
- `et(faux(), faux()) = faux()`

- `ou(x, y) = non(et(non(x), non(y)))`

Le symbole $\emptyset$, utilisé pour désigner le domaine de définition des fonctions `vrai` et `faux`, signifie que ces fonctions n'ont pas de paramètre.

Pour le type `t_booleen` :

1. les deux premiers axiomes expriment les résultats de la fonction `non` par rapport aux constructeurs `vrai` et `faux` ;
2. les quatre axiomes suivants expriment les résultats de la fonction `et` par rapport aux deux constructeurs [1] ;
3. le dernier axiome exprime le résultat de la fonction `ou` par rapport aux fonctions `non` et `et`. On peut noter qu'on aurait pu choisir d'exprimer le résultat de la fonction `ou` par rapport aux constructeurs `vrai` et `faux`, ce qui aurait été équivalent d'un point de vue logique.

1.3.2.2 Les entiers naturels

Le type abstrait `t_entier` décrit ci-dessous est une représentation de l'ensemble des entiers naturels $\mathbb{N}$. Il comporte deux constructeurs `zero` et `plusun` :

— la valeur `zero()` correspond intuitivement à la valeur 0 des entiers naturels,

— le constructeur `plusun`, appliqué à une valeur de type `t_entier`, correspond intuitivement à l'addition d'une unité à l'entier naturel correspondant : `plusun(zero())` correspond donc intuitivement à l'entier 1, `plusun(plusun(zero()))` correspond intuitivement à l'entier 2, *etc.*

Les autres fonctions définies ici sont :

— `estnul`, qui correspond à la comparaison [2] avec 0,

— `egal`, qui correspond à un test d'égalité de deux entiers naturels [3],

1. On peut remarquer la similitude avec la *table de vérité* de la fonction `et`.

2. Généralement, dans la définition d'un type abstrait, les types abstraits utilisés dans cette définition sont explicitement indiqués : par exemple, le type `t_entier` utilise le type `t_booleen` défini précédemment. Pour alléger les notations, nous considérons dans ce livre que tout type abstrait préalablement défini peut être utilisé pour la définition d'un autre type abstrait.

3. Il faut avoir à l'esprit qu'*aucune fonction de manipulation d'un type abstrait n'est prédéfinie* : il est donc nécessaire de définir toutes les fonctions de manipulation que l'on souhaite utiliser, y compris le cas échéant les comparateurs usuels, comme par exemple la fonction `egal` pour le type `t_entier`.

— `plus`, qui correspond à l'addition de deux entiers naturels,
— `moinsun`, qui correspond à la soustraction d'une unité,
— `mult`, qui correspond à la multiplication de deux entiers naturels.

D'autres fonctions pourraient bien sûr être définies : toutes les fonctions de comparaison, les opérations de division entière, l'exponentiation, *etc.*

Type abstrait `t_entier`
Constructeurs
- `zero : ∅ → t_entier`
- `plusun : t_entier → t_entier`

Autres opérations
- `estnul : t_entier → t_booleen`
- `egal : t_entier × t_entier → t_booleen`
- `plus : t_entier × t_entier → t_entier`
- `moinsun : t_entier → t_entier`
- `mult : t_entier × t_entier → t_entier`

Préconditions
- `moinsun(x) défini ⇔ non(estnul(x))`

Axiomes
- `estnul(zero()) = vrai()`
- `estnul(plusun(x)) = faux()`

- `egal(zero(), zero()) = vrai()`
- `egal(zero(), plusun(x)) = faux()`
- `egal(plusun(x), zero()) = faux()`
- `egal(plusun(x), plusun(y)) = egal(x,y)`

- `plus(x, zero()) = x`
- `plus(x, plusun(y)) = plusun(plus(x,y))`

- `moinsun(plusun(x)) = x`

- ```
  mult(x, y) =
    si estnul(y)
    alors zero()
    sinon plus(mult(x, moinsun(y)), x)
  ```

On peut noter sur cet exemple de définition du type `t_entier` que plusieurs notions peuvent être utilisées afin de définir un type abstrait :
  ```

1. ***préconditions*** : le domaine de définition d'une fonction peut être restreint par une *précondition*. Par exemple, la fonction `moinsun` correspond intuitivement à la soustraction d'une unité à un entier ; or, l'entier $0 - 1$ n'est pas un entier naturel. Le type abstrait comporte donc une précondition, indiquant que la fonction `moinsun` n'est pas applicable à la valeur `zero()` ;

2. ***variables*** : des variables peuvent être utilisées, permettant de désigner toute valeur du type. Par exemple, le deuxième axiome s'écrit :
   ```
   estnul(plusun(x)) = faux()
   ```
   `x` est une variable, représentant n'importe quelle valeur de type `t_entier` ; ceci signifie que `estnul(plusun(x)) = faux()` pour *tout* élément `x` du type `t_entier`.
   De même, l'axiome :
   ```
   plus(x, plusun(y)) = plusun(plus(x,y))
   ```
   utilise deux variables `x` et `y` ; cet axiome doit se lire : *pour toutes valeurs* `x` et `y` de type `t_entier`, la propriété décrite est vérifiée ;

3. ***conditionnelle*** : comme cela a déjà été vu lors de la définition du type `t_booleen`, il est nécessaire de distinguer des cas (*cf.* par exemple les axiomes définissant le comportement de la fonction `et` par rapport aux constructeurs `vrai` et `faux`). À cette fin, nous utilisons la *conditionnelle* [1] comme facilité d'écriture, comme cela est fait pour la définition du comportement de la fonction `mult` ;

4. ***composition de fonctions, récursivité*** : comme cela a déjà été vu lors de la définition du type `t_booleen`, une fonction peut être définie par *composition de fonctions* définies antérieurement. De même, la *récursivité* peut être utilisée pour la définition d'un axiome. Par exemple, l'axiome `egal(plusun(x), plusun(y)) = egal(x,y)` est défini récursivement [2], de même que d'autres axiomes liés aux fonctions `plus` et `mult`.

---

1. On peut noter que la conditionnelle peut être définie comme une fonction qui a en paramètre une valeur de type `t_booleen` et deux valeurs de même type, et dont le résultat est l'une de ces deux valeurs suivant la valeur du booléen : la première valeur si le booléen est `vrai()`, la deuxième valeur si le booléen est `faux()`. Dans ce livre, la conditionnelle est écrite de manière similaire à celle d'une expression conditionnelle en `OCaml`. Par exemple, le comportement de la fonction `et` du type `t_booleen` aurait pu être défini ainsi :
```
et(x, y) =
    si x
    alors y
    sinon faux()
```
2. Intuitivement, il correspond à la propriété suivante pour les entiers naturels : $\forall x, y \in \mathbb{N}, (x + 1 = y + 1) \Leftrightarrow (x = y)$.

1.3.2.3 Les ensembles

Un type peut être paramétré par un autre type. Par exemple, on peut vouloir définir le type `t_ensemble` dont les éléments sont d'un type quelconque[1] `t_t`.

Type abstrait `t_ensemble(t_t)`
Constructeurs
- `e_vide : ∅ → t_ensemble(t_t)`
- `e_singleton : t_t → t_ensemble(t_t)`
- `e_union : t_ensemble(t_t) × t_ensemble(t_t) → t_ensemble(t_t)`

Autres opérations
- `e_estvide : t_ensemble(t_t) → t_booleen`
- `e_appartient : t_ensemble(t_t) × t_t → t_booleen`
- `e_choix : t_ensemble(t_t) → t_t`
- `e_suppression : t_ensemble(t_t) × t_t → t_ensemble(t_t)`
- `e_cardinal : t_ensemble(t_t) → t_entier`

Préconditions
- `e_choix(e) défini ⇔ non(e_estvide(e))`

Axiomes
- `e_estvide(e_vide()) = vrai()`
- `e_estvide(e_singleton(e)) = faux()`
- `e_estvide(e_union(e, f)) = et(e_estvide(e), e_estvide(f))`

- `e_appartient(e_vide(), x) = faux()`
- `e_appartient(e_singleton(x), y) = egal_t(x, y)`
- `e_appartient(e_union(e, f), x) =`
  `ou(e_appartient(e, x), e_appartient(f, x))`

- `e_appartient(e, e_choix(e)) = vrai()`

- `e_suppression(e_vide(), x) = e_vide()`
- `e_suppression(e_singleton(x), y) =`
  `si egal_t(x, y)`
  `alors e_vide()`
  `sinon e_singleton(x)`
- `e_suppression(e_union(e, f), x) =`
  `e_union(e_suppression(e, x), e_suppression(f, x))`

1. Dans cette définition, nous supposons que la fonction `egal_t : t_t × t_t → t_booleen` figure dans la définition du type abstrait `t_t`, et qu'elle teste l'égalité de deux valeurs de ce type.

```
• e_cardinal(e) =
      si est_vide(e)
      alors zero()
      sinon plusun(e_cardinal(e_suppression(e, e_choix(e))))
```

Les correspondances intuitives entre le type abstrait et les ensembles au sens mathématique sont les suivantes :

Type abstrait	**Ensembles**
`e_vide()`	$\emptyset$
`e_singleton(x)`	$\{x\}$
`e_union(e, f)`	$e \cup f$
`e_appartient(e, x)`	$x \in e$
`e_suppression(e, x)`	$e \setminus \{x\}$
`e_cardinal(e)`	$\lvert e \rvert$

La fonction `e_choix` appliquée à un ensemble correspond intuitivement au choix d'un élément quelconque de cet ensemble. On peut remarquer que l'axiome :

`e_appartient(e, e_choix(e)) = vrai()`

ne définit pas la valeur de `e_choix(e)` ; il indique simplement que cette valeur appartient à `e`.

Remarque 1 : il ne faut pas confondre le type abstrait avec la notion que l'on cherche à représenter, par exemple :

1. mathématiquement, les ensembles $\{x\}$ et $\{x\} \cup \{x\}$ sont égaux ; en revanche, rien n'indique, dans la définition du type abstrait `t_ensemble`, que les valeurs `e = e_singleton(x)` et `f = e_union(e_singleton(x), e_singleton(x))` sont égales [1]. Ceci dit, ces deux valeurs sont *indistinguables*, dans le sens où il est possible de prouver les propriétés suivantes à partir des axiomes des types `t_ensemble(t_t)`, `t_entier` et `t_booleen` :
   — les valeurs `e_estvide(e)` et `e_estvide(f)` sont égales, dans le sens où elles sont toutes deux égales soit à `vrai()`, soit à `faux()` ;
   — les valeurs `e_appartient(e, x)` et `e_appartient(f, x)` sont égales (dans le même sens), pour tout élément `x` de `t_t` ;
   — `egal(e_cardinal(e), e_cardinal(f)) = vrai()` ;

1. Pour obtenir cette propriété, et plus généralement pour représenter l'égalité des ensembles au sens mathématique, on peut :
— ajouter des axiomes, ce qui peut restreindre les possibilités d'implantation du type abstrait : *cf.* section suivante 1.3.3 ;
— ou, de manière similaire à ce qui a été fait pour le type `t_entier`, définir une fonction « `e_egal` » et les axiomes correspondants.

— les propriétés ci-dessus peuvent être prouvées aussi pour les valeurs `e_suppression(e, y)` et `e_suppression(f, y)`, pour tout élément `y` de `t_t`.

2. il est possible d'utiliser le type `t_ensemble` pour représenter d'autres notions que les ensembles mathématiques habituels : il suffit que les opérations que l'on a choisi de représenter vérifient les préconditions et les axiomes du type. En particulier, il peut arriver que deux valeurs indistinguables du type (par exemple les valeurs `e` et `f` définies ci-dessus) correspondent à deux valeurs différentes pour la notion représentée.

Remarque 2 : il est bien sûr possible d'imaginer plusieurs types abstraits pour représenter une même notion : voici par exemple une autre définition du type `t_ensemble` qui utilise des constructeurs différents :

Type abstrait `t_ensemble(t_t)`
Constructeurs
- `e_vide` : $\emptyset \rightarrow$ `t_ensemble(t_t)`
- `e_insertion` : `t_ensemble(t_t)` $\times$ `t_t` $\rightarrow$ `t_ensemble(t_t)`

Autres opérations
- `e_estvide` : `t_ensemble(t_t)` $\rightarrow$ `t_booleen`
- `e_appartient` : `t_ensemble(t_t)` $\times$ `t_t` $\rightarrow$ `t_booleen`
- `e_choix` : `t_ensemble(t_t)` $\rightarrow$ `t_t`
- `e_cardinal` : `t_ensemble(t_t)` $\rightarrow$ `t_entier`
- `e_suppression` : `t_ensemble(t_t)` $\times$ `t_t` $\rightarrow$ `t_ensemble(t_t)`
- `e_singleton` : `t_t` $\rightarrow$ `t_ensemble(t_t)`
- `e_union` : `t_ensemble(t_t)` $\times$ `t_ensemble(t_t)` $\rightarrow$ `t_ensemble(t_t)`

Préconditions
- `e_choix(e)` défini $\Leftrightarrow$ `non(e_estvide(e))`

Axiomes
- `e_estvide(e_vide()) = vrai()`
- `e_estvide(e_insertion(e, x)) = faux()`

- `e_appartient(e_vide(), x) = faux()`
- `e_appartient(e_insertion(e, x), y) =`

```
si egal_t(x, y)
alors vrai()
sinon e_appartient(e, y)
```

- `e_appartient(e, e_choix(e)) = vrai()`

- `e_cardinal(e_vide()) = zero()`

```
• e_cardinal(e_insertion(e, x)) =
        si e_appartient(e, x)
        alors e_cardinal(e)
        sinon plusun(e_cardinal(e))

• e_suppression(e_vide(), x) = e_vide()
• e_suppression(e_insertion(e, x), y) =
        si egal_t(x, y)
        alors e_suppression(e, y)
        sinon e_insertion(e_suppression(e, y), x)

• e_singleton(x) = e_insertion(e_vide(), x)

• e_union(e, e_vide()) = e
• e_union(e, e_insertion(f, x)) =
                        e_insertion(e_union(e, f), x)
```

Il est donc possible d'imaginer différents types abstraits pour une même notion, ce qui pose plusieurs questions dont l'étude dépasse le cadre de cet ouvrage :

1. un type abstrait défini pour représenter une notion « capte-t-il » toutes les propriétés de cette notion, et seulement ces propriétés ?
2. si deux types abstraits sont définis pour représenter la même notion, l'un est-il « meilleur » que l'autre à cette fin ?

1.3.2.4 Définition

Dans cet ouvrage, la définition d'un type abstrait de nom `nom_type` a la forme suivante :

Type abstrait *nom_type*
ou *nom_type(type_parametre)* si le type est paramétré par un autre type
Constructeurs
liste de signatures de fonctions
Autres opérations
liste de signatures de fonctions
Préconditions
liste de préconditions
où une précondition est exprimée par une expression logique
Axiomes
liste d'axiomes
où un axiome est exprimé par une expression logique

Peuvent intervenir dans la définition d'un type abstrait des variables, des expressions conditionnelles, des définitions composant des fonctions, en particulier des définitions récursives.

1.3.3 Implantation

Implanter un type abstrait signifie *choisir un langage de programmation*, *choisir un (ou plusieurs) type(s) concret(s) pour représenter les valeurs du type abstrait*, et *programmer les fonctions de manière à ce qu'elles vérifient les signatures des fonctions du type abstrait, les préconditions ainsi que les axiomes*. Les implantations discutées ci-dessous utilisent le langage `OCaml`.

1.3.3.1 Une implantation du type `t_booleen` utilisant le type `bool`

Le type abstrait peut être implanté ainsi :

```
type t_booleen = bool ;;
let vrai() : t_booleen = true ;;
let faux() : t_booleen = false ;;
let non(x : t_booleen) = not(x) ;;
let et(x, y : t_booleen * t_booleen) : t_booleen = x && y ;;
let ou(x, y : t_booleen * t_booleen) : t_booleen = x || y ;;
```

On peut alors vérifier que l'implantation vérifie les axiomes du type abstrait. Pour les booléens, il suffit de vérifier explicitement les propriétés sur toutes les (combinaisons de) valeurs possibles :

```
# non(vrai()) = faux() ;;
- : bool = true
# non(faux()) = vrai();;
- : bool = true
# et(vrai(), vrai()) = vrai();;
- : bool = true
# et(vrai(), faux()) = faux() ;;
- : bool = true
# et(faux(), vrai()) = faux() ;;
- : bool = true
# et(faux(), faux()) = faux() ;;
- : bool = true
# ou(vrai(), vrai()) = non(et(non(vrai()), non(vrai()))) ;;
- : bool = true
# ou(vrai(), faux()) = non(et(non(vrai()), non(faux()))) ;;
- : bool = true
# ou(faux(), vrai()) = non(et(non(faux()), non(vrai()))) ;;
- : bool = true
# ou(faux(), faux()) = non(et(non(faux()), non(faux()))) ;;
- : bool = true
```

1.3.3.2 Une implantation du type `t_booleen` utilisant les valeurs `0` et `1` du type `int`

Il est bien sûr possible d'implanter un type abstrait de différentes manières. Pour être valide, une implantation doit respecter la définition du type abstrait (signatures, préconditions, axiomes). Deux implantations valides peuvent parfois se distinguer par des différences de complexité des fonctions. Ce n'est pas le cas ici pour cette deuxième implantation du type `t_booleen` :

```
type t_booleen = int ;;
let vrai() : t_booleen = 1 ;;
let faux() : t_booleen = 0 ;;
let non(x : t_booleen) = (x + 1) mod 2 ;;
let ou(x, y : t_booleen * t_booleen) : t_booleen =
   if x + y > 0 then 1 else 0
;;
let et(x, y : t_booleen * t_booleen) : t_booleen =
  non(ou(non(x), non(y)))
;;
```

La vérification des axiomes se fait de manière similaire à celle présentée dans le cadre de l'implantation utillisant le type `bool`.

Une autre implantation des fonctions `et` et `ou` est :

```
let et(x, y : t_booleen * t_booleen) : t_booleen =
   x * y
;;
let ou(x, y : t_booleen * t_booleen) : t_booleen =
  non(et(non(x), non(y)))
;;
```

Dans ce cas, il est inutile de vérifier l'axiome lié à la fonction `ou`, puisque l'implantation de cette fonction correspond exactement à cet axiome.

On remarque ici que le type concret `int` contient d'autres valeurs que celles correspondant aux valeurs construites par les constructeurs, c'est-à-dire toutes les valeurs du type `int` sauf `0` et `1`. Ceci dit, ces valeurs sont *inatteignables*, dans le sens où aucune composition des opérations du type abstrait `t_booleen` ne s'évalue dans le type concret `int` par une valeur autre que les valeurs `0` et `1` associées aux valeurs des constructeurs [1].

Notons que les langages de programmation rendent généralement possible *l'encapsulation*, qui permet de « cacher » le(s) type(s) sous-jacent(s) à une

1. D'autres possibilités existent. Par exemple en langage `C`, les booléens sont implantés par des entiers, avec la convention suivante : la valeur `0` représente `false`, tout autre entier représente la valeur `true`.

implantation (par exemple ici le type `int`), ainsi que les opérations de manipulation (par exemple ici `=`, `mod`, *etc*) : cela permet en particulier de rendre les valeurs effectivement inatteignables (par exemple, il ne serait pas possible de déclarer un élément de type `t_booleen` avec une valeur non construite par un constructeur).

1.3.3.3 Validité d'une implantation

De même que la *conception* d'un type abstrait, l'*implantation* d'un type abstrait est délicate, car il est essentiel que l'implantation vérifie tous les axiomes de la définition, et ce pour toutes les valeurs possibles. Un contre-exemple simple est l'implantation « directe » du type `t_entier` par le type `int` de `OCaml`, de la manière suivante :

```
type t_entier = int ;;
let zero() : t_entier =
  0
;;
let plusun(x : t_entier) : t_entier =
  x + 1
;;
let estnul(x : t_entier) : t_booleen =
  if x = 0
  then vrai()
  else faux()
;;
```

En effet, le type `int` de `OCaml` comprend des entiers négatifs. Or, le deuxième axiome du type abstrait, `estnul(plusun(x)) = faux()`, n'est pas vérifié pour l'entier négatif `-1` :

```
# estnul(plusun(-1)) = faux() ;;
- : bool = false
```

Comme nous l'avons vu précédemment, ce n'est pas gênant si les entiers négatifs sont inatteignables, mais ce n'est pas le cas ici : les constructeurs `zero` et `plusun` engendrent *toutes* les valeurs du type `int`. En effet, la valeur qui suit le plus grand élément du type `int` (`max_int`) est le plus petit élément (`min_int`) :

```
# (max_int, min_int) ;;
- : int * int = (4611686018427387903, -4611686018427387904)
# plusun(max_int) = min_int ;;
- : bool = true
```

Dit autrement, tant que les valeurs entières que l'on manipule sont comprises dans l'intervalle $[0, max_int]$, les valeurs de type `int` se comportent

comme les valeurs correspondantes de type `t_entier`. Au contraire, les valeurs négatives (c.-à-d. comprises dans l'intervalle $[min_int, 0[$) ne respectent pas les axiomes, et elles sont atteignables. Le type `int` n'est donc pas une implantation valide du type abstrait `t_entier`.

1.3.3.4 Quelques remarques

On peut noter que *certains axiomes permettent d'implanter directement les fonctions correspondantes* : par exemple, les fonctions `non`, `et` et `ou` du type `t_booleen` pourraient être implantés directement à partir des axiomes associés (la définition de la fonction `et` est simplifiée ci-dessous par regroupement des cas) :

```
type t_booleen = int ;;
let vrai() : t_booleen = 1 ;;
let faux() : t_booleen = 0 ;;
let non(x : t_booleen) =
  if x = vrai() then faux() else vrai()
;;
let et(x, y : t_booleen * t_booleen) : t_booleen =
   if x = vrai() && y = vrai()
   then vrai()
   else faux()
;;
let ou(x, y : t_booleen * t_booleen) : t_booleen =
  non(et(non(x), non(y)))
;;
```

Ceci n'est pas toujours possible : par exemple, l'axiome lié à la fonction `e_choix` du type `t_ensemble` ne donne aucune indication sur une manière possible d'implanter cette fonction. En fait, la fonction `e_choix` est incomplètement décrite par son axiome `e_appartient(e, e_choix(e)) = vrai()`, qui indique simplement que le résultat de la fonction est un élément de l'ensemble paramètre ; mais aucun axiome ne précise *quel* élément doit être le résultat.

Par exemple, on peut implanter le premier type abstrait `t_ensemble` par des listes `OCaml`, de la manière suivante (le type `t_booleen` est remplacé par le type `bool` pour simplifier) :

```
type 'a t_ensemble = 'a list ;;
let e_vide() : 'a t_ensemble =
  empty()
;;
let e_singleton(x : 'a) : 'a t_ensemble =
  add_fst(empty(), x)
;;
let e_union(e, f : 'a t_ensemble * 'a t_ensemble) : 'a t_ensemble =
  concat(e, f)
;;
```