Algorithmes et structures de données. Une introduction à la programmation fonctionnelle et au langage Caml

Jocelyn Sérot

22 avril 2004

Préambule

Ce document constitue une introduction à la programmation fonctionnelle – et en particulier au langage Objective Caml – à travers la mise en œuvre de structures de données classiques en informatique (listes, arbres, graphes). Dans cette optique, les implantations et les algorithmes proposés restent délibérément simples. Il existe de nombreux documents traitant de manière beaucoup plus approfondie des aspects purement algorithmiques. Par ailleurs, bien que ce cours puisse être vu comme une initiation à Caml, il ne constitue en aucune façon une formation complète à ce langage et encore moins d'un manuel de référence. Le lecteur est donc invité, corrélativement, à se reporter aux ouvrages, documents et publications (pour la plupart disponibles en lignes) traitant du langage et de ses applications (voir la bibliographie en fin de document).

La forme du document privilégie une approche « en situation » des problèmes, les types de données et fonctions étant construits et validés « en temps réel » via un interpréteur *Caml*. Ces notes constituent une trace commentée de ces sessions de travail : les phrases entrées par le programmeur sont précédées du caractère d'invite # et suivies de la réponse de l'interpréteur.

Chapitre 1

Rudiments

1.1 Expressions, valeurs et types

La notion clé en Caml (et plus généralement dans tout langage *fonctionnel*) est celle d'*expression*. Toute expression possède une *valeur* et un *type*. Ecrire un programme consiste à définir une expression dont la valeur est la solution au problème posé. L'ordinateur ne fait alors qu'évaluer cette valeur. Dans les exemples qui suivent, cette évaluation est effectuée de manière interactive par le *toplevel*.

#1+2;;

- : *int* = 3

Parmi les types de base on trouve notamment : *int*, *float*, *bool* et *string*

1.1.1 Entiers et flottants

#2.1 *. 3.4;;

- : *float* = 7.14

#4.5 *. 2;;

Characters 7 − 8 :
    4.5 *. 2;;
          ^

This expression has type *int but is here used* with type *float*

Les types *int* et *float* sont distincts et requièrent des opérateurs spécifiques. Les coercions sont toujours explicites[1] via les fonctions *float_of_int* et *int_of_float* par exemple

#4.5 *. *float_of_int*(2);;

- : *float* = 9.

1.1.2 Chaines de caractères

Les chaines de caractères sont écrites entre ". L'opérateur de concaténation est ^

#`"Hello"`;;

- : *string* = `"Hello"`

[1] On verra pourquoi plus loin.

#"Hello, " ^ "world!";;

- : *string* = "Hello, world!"

1.1.3 Booléens

Il y deux valeurs booléennes : true et false. Les relations usuelles de comparaison (=, <, >=, ...) en particulier retournent une valeur booléenne. Elles opérent sur des valeurs n'importe quel type :

#1> 2;;

- : *bool* = false

#1.2 = 1.2;;

- : *bool* = true

#"hello" = "world";;

- : *bool* = false

La conditionnelle s'exprime avec if, then et else. **Attention**, c'est ici une *expression*, qui possède donc une *valeur* :

#if 5+4 > 10 then "Oui" else "Non";;

- : *string* = "Non"

1.1.4 N-uplets

Les *n-uplets* correspondent à un produit cartésien de valeurs : ils combinent des valeurs de types quelconques sous la forme de paires, triplets, *etc.* :

#(1, 2);;

- : *int* × *int* = (1, 2)

#("jean", 46, 2650.0);;

- : *string* × *int* × *float* = ("jean", 46, 2650.)

1.2 Définitions

Le mot-clé let permet de nommer une valeur (pour la réutiliser plus tard par exemple). La syntaxe est la suivante :

let *nom* = *valeur*

#let *pi* = 4.0 *. *atan* 1.0;;

val *pi* : *float* = 3.14159265359

#*pi* *. *pi*;;

- : *float* = 9.86960440109

Attention : Les définitions ne correspondent pas aux « variables » (du C notamment). En particulier, elles ne sont pas modifiable : le nom défini est définitivement lié à la valeur calculée lors de la définition. C'est un simple synonyme pour cette valeur. On ne peut donc pas utiliser un nom avant de l'avoir défini.

```
#let y = sin (pi /. 2.0);;
val y : float = 1.
#let u = y *. 2.0;;
val u : float = 2.
#let z = x + 1;;
Characters 8 - 9 :
    let z = x + 1;;
            ^
Unbound value x
```

1.2.1 Définitions locales

La forme

let *v* = *e1* in *e2*

signifie que x vaut *e1* pendant le calcul de *e2* (et seulement pendant ce calcul).

```
#let r =
    let x = 3 in
    x * x;;
val r : int = 9
#x;;
Characters 0 - 1 :
    x;;
    ^
Unbound value x
```

L'identificateur x n'est visible que dans l'expression qui suit le in du let correspondant.

Les définitions peuvent être imbriquées à une profondeur arbitraire. Dans ce cas, les identificateurs définis à un niveau d'imbrication sont visibles à tous les niveaux « inférieurs ».

```
#let u = 2;;
val u : int = 2
#let v = 3;;
val v : int = 3
#let w =
    let x = u + v in
      let y = x * u in
      x * y;;
val w : int = 50
```

1.3 Fonctions

En Caml, les fonctions sont des valeurs comme les autres. On les définit aussi avec le mot clé let :

$$\text{let } nom\_de\_la\_fonction = \text{function } nom\_du\_paramètre \rightarrow \ expression$$

#let *square* = function $x \rightarrow x \times x$;;

val *square* : $int \rightarrow int$ = <fun>

#*square*(2);;

- : $int = 4$

#let *cube* = function $x \rightarrow square(x) \times x$;;

val *cube* : $int \rightarrow int$ = <fun>

Noter le type de la fonction *square* :

$$int \rightarrow \ int$$

c.-à-d. « fonction prenant un argument de type *int* et retournant un résultat de type *int* ». Remarquer aussi que ce type a été *inféré* automatiquement par le système.

En Caml, les parenthèses autour des arguments des fonctions sont facultatives. On peut donc écrire plus simplement :

#*square* 3;;

- : $int = 9$

Les parenthèses restent utiles, bien sur, pour *grouper* des expressions :

#*square* $(2 + 3)$;;

- : $int = 25$

#*square* 2 + 3;;

- : $int = 7$

La définition de fonction étant une opération très fréquente, il existe une notation abrégée. A la place de

$$\text{let } nom\_de\_la\_fonction = \text{function } nom\_du\_paramètre \rightarrow \ expression$$

on peut écrire

$$\text{let } nom\_de\_la\_fonction \ nom\_du\_paramètre = expression$$

#let *square* $x = x \times x$;;

val *square* : $int \rightarrow int$ = <fun>

Remarque importante : la (re)définition de la fonction *square* « cache » désormais celle donnée au début de ce paragraphe. Lors du calcul de

#*square* 3;;

- : $int = 9$

c'est la bien la *seconde* définition qui est utilisée. Par contre, l'évaluation de la fonction *cube* continuera d'utiliser la *première* définition de *square*. C'est ce que l'on appelle la règle de *portée statique* des identificateurs en Caml : la valeur d'un identificateur est déterminée par l'environnement au sein duquel cet identificateur est *défini*, pas par celui au sein duquel il est *évalué*.

1.3.1 Fonctions à plusieurs arguments

Il y a deux manières de définir des fonctions à plusieurs arguments.

La première consiste à définir une fonction prenant un n-uplet.

#let $norme\ (x, y)\ =\ sqrt(x * .x + .y * .y);;$

val $norme\ :float \times float \rightarrow\ float =$ <fun>

La seconde consiste à passer les arguments « les uns après les autres »

#let $norme\ x\ y\ =\ sqrt(x * .x + .y * .y);;$

val $norme\ :float \rightarrow\ float \rightarrow\ float =$ <fun>

Noter ici le *type* de la fonction f :

$$float \rightarrow\ float \rightarrow\ float$$

La fonction f prend donc deux arguments de type *float* et renvoie un résultat de type *float*[2]. Cette forme permet par ailleurs de définir des fonctions par *application partielle* (note : ce qui suit peut être sauté en première lecture). Considérons par exemple la fonction *add* à deux arguments définie ci-dessous :

#let $add\ x\ y\ =\ x\ +\ y;;$

val $add\ :int \rightarrow\ int \rightarrow\ int =$ <fun>

Le type de cette fonction :

$$int \rightarrow\ int \rightarrow\ int$$

peut en fait se lire comme[3] :

$$int \rightarrow\ (int \rightarrow\ int)$$

Ceci signifie que l'on peut voir *add* comme une fonction prenant un entier et renvoyant une *fonction* prenant elle-même un entier et renvoyant un entier. L'intérêt de cette interprétation est que l'on peut alors définir des fonctions par *application partielle* de *add*. Par exemple :

#let $incr\ =\ add\ 1;;$

val $incr\ :int \rightarrow\ int =$ <fun>

$\#incr\ 4;;$

- : $int = 5$

Cette manière de définir les fonctions à n arguments – par « imbrication » de fonctions à un seul argument – est dite *curryfiée*[4]. Elle est très utile, car elle permet souvent de définir des fonctions particulières par *spécialisation* de fonctions plus générales.

1.3.2 Fonctions à plusieurs résultats

Les n-uplets servent aussi à définir des fonctions renvoyant plusieurs résultats

#let $div\_eucl\ x\ y\ =\ x/y,\ x$ mod $y;;$

val $div\_eucl\ :int \rightarrow\ int \rightarrow\ int \times int =$ <fun>

$\#div\_eucl\ 14\ 3;;$

- : $int \times int = (4,\ 2)$

[2] La définition d'une fonction à plusieurs arguments en notation non abrégée se fait avec le mot clé fun au lieu de function. On aurait ainsi pu écrire let $norme\ =$ fun $x\ y\ \rightarrow\ sqrt(x * .x + .y * .y)$.

[3] L'opérateur $\rightarrow$ étant associatif à gauche.

[4] Ce nom fait référence à H. Curry, un mathématicien qui a introduit cette technique dans le cadre du λ-calcul.

1.4 Un peu plus sur les fonctions

1.4.1 Fonctions récursives

Les fonctions récursives – *i.e.* dont la définition contient un appel à elle même – jouent un rôle clé en programmation fonctionnelle. Elles permettent notamment d'exprimer la répétition sans itération. Un exemple classique est celui de la fonction *factorielle*. Cette fonction est définie mathématiquement par

$$fact(n) = n \times (n-1) \times \ldots \times 1 = \begin{cases} 1 & \text{si } n = 0, \\ n \times fact(n-1) & \text{si } n > 1 \end{cases}$$

Elle s'écrit « naturellement » en Caml

```
#let rec fact n =
    if n = 0 then 1
    else n * fact (n-1);;
val fact : int -> int = <fun>
#fact 5;;
- : int = 120
```

Noter l'usage du mot-clé rec après le let. Sans cela, l'identificateur de la fonction n'est pas connu dans son corps.

Un mot sur l'évaluation

L'évaluation des fonctions – en fait de toute expression – procède par *réductions* successives. La séquence de réductions correspondant à l'évaluation de *fact* 3 est ainsi la suivante :

$$\begin{array}{ll} & fact\ 3 \\ \Rightarrow & \text{if } 3 = 0 \text{ then } 1 \text{ else } 3 \times fact\ (3-1) \\ \Rightarrow & 3 \times fact\ (3-1) \\ \Rightarrow & 3 \times fact\ 2 \\ \Rightarrow & 3 \times (2 \times fact\ (2-1)) \\ \Rightarrow & 3 \times (2 \times fact\ 1) \\ \Rightarrow & 3 \times (2 \times (1 \times fact\ (1-1))) \\ \Rightarrow & 3 \times (2 \times (1 \times fact\ 0)) \\ \Rightarrow & 3 \times (2 \times (1 \times 1)) \\ \Rightarrow & 3 \times (2 \times 1) \\ \Rightarrow & 3 \times 2 \\ \Rightarrow & 6 \end{array}$$

On peut « visualiser » ce mécanisme d'évaluation en utilisant la directive[5] `#trace` :

Note : le nombre de réductions (d'opérations « élémentaires ») nécessaire pour calculer $f(n)$ définit la *complexité* $\chi_f(n)$ de la fonction f. Cette complexité est généralement mesurée par *analyse asymptotique* : on dit que f « est en $O(g(n))$ » si il existe une constante C telle que, pour $n > N$, $\chi_f(n) < C \times g(n)$. La fonction *fact* définie plus haut est ainsi en $O(n)$.

Exercice. Ecrire une fonction calculant x^n pour deux entiers x et n en utilisant la récurrence

$$x^0 = 1, \quad x^{n+1} = x \times x^n$$

Idem en utilisant la récurrence

$$x^0 = 1, \quad x^1 = 1, \quad x^{2n} = (x^n)^2, \quad x^{2n+1} = x \times x^{2n}$$

[5] Les *directives*, commencant par le caractère `#`, sont des ordres permettant de contrôler le comportement de l'interpréteur.

1.4.2 Définition de fonction par filtrage

Caml dispose d'un mécanisme très puissant pour définir des fonctions par analyse de cas : le *filtrage* (*pattern matching*). La forme générale de ce mécanisme est la suivante :

```
  match expr with
    motif1  →  exp1
|   motif2  →  exp2
...
|   motifN  →  expN
```

Un motif (*pattern*) est une expression faisant intervenir des constantes et des variables et définissant une « forme » possible pour l'expression *expr*. Lors de l'évaluation l'expression *expr* est filtrée par (mise en correspondance avec) les motifs *motif1* ... *motifN*. Dès qu'un motif « colle » à l'expression (par exemple $motif_i$) la valeur correspondante (exp_i) est évaluée et retournée.

Par exemple : la fonction dite de Fibonacci est définie classiquement par

$$Fib(n) = \begin{cases} 1 & \text{si } n = 0, \\ 1 & \text{si } n = 1, \\ Fib(n-1) + Fib(n-2) & \text{si } n > 1 \end{cases}$$

Elle se code immédiatement en Caml :

```
#let rec fib n = match n with
    0 → 1
  | 1 → 1
  | n → fib (n-1) + fib (n-2);;

val fib : int → int = <fun>

#fib 5;;

- : int = 8
```

Noter que lorsque le troisième motif est sélectionné (lors de l'évaluation de *fib* 5, par exemple), l'identificateur n est automatiquement *lié* à la valeur filtrée (5) pour être utilisée dans l'expression associée au motif.

La fonction *fact* définie plus haut aurait aussi pu s'écrire :

```
#let rec fact n = match n with
    0 → 1
  | n → n × fact (n-1);;

val fact : int → int = <fun>
```

Le filtrage se généralise à des valeurs multiple :

```
#let et_logique x y = match x, y with
    true, true → true
  | true, false → false
  | false, true → false
  | false, false → false;;

val et_logique : bool → bool → bool = <fun>

#et_logique true false;;

- : bool = false

#et_logique true true;;
```

- : *bool* = true

Une grande force de Caml est que le compilateur est capable de vérifier l'*exhaustivité* du filtrage

```
#let ou_logique x y = match x, y with
        true, true -> true
    | true, false -> true
    | false, false -> false;;

.....................match x, y with
          true, true -> true
        | true, false -> true
        | false, false -> false...
val ou_logique : bool -> bool -> bool = <fun>

Warning : this pattern-matching is not exhaustive.
Here is an example of a value that is not matched :
(false, true)
```

Il faut toujours prêter attention aux avertissements du compilateur, sous peine d'erreurs à l'exécution ...

#ou_logique false true;;

Exception : *Match_failure* ("", 21, 109).

Ici, il suffit de rajouter le cas manquant ou, plus simplement, de redéfinir la fonction *ou_logique* en utilisant le motif universel (_), qui filtre n'importe quelle valeur :

```
#let ou_logique x y = match x, y with
        false, false -> false
    | _, _ -> true;;

val ou_logique : bool -> bool -> bool = <fun>

#ou_logique false true;;

- : bool = true
```

1.5 Définition de types

La facilité avec laquelle on peut définir et manipuler – via le mécanisme de filtrage – de nouveaux types de données en Caml contribue fortement à l'expressivité du langage. Cet aspect sera notamment développé dans les chapitres suivants. On présente ici deux types de données « extensibles » élémentaires : les types sommes et les enregistrements.

1.5.1 Enregistrements

Les enregistrements peuvent être vus comme des n-uplets dont les composantes sont nommées[6]. Pour pouvoir les utiliser, il faut d'abord définir le type correspondant. Cela se fait avec le mot clé `type` :

```
#type employe = {
          nom : string;
          age : int;
     salaire : float
#};;
```

[6] On peut aussi les voir comme l'analogue des `struct` du C ou des `record` du Pascal.

```
type employe = { nom : string; age : int; salaire : float; }
```

On peut alors définir des valeurs de type *employe* en spécifiant la valeur de chaque champ :

```
#let jean = { nom = "Jean"; age = 34; salaire = 2800.0 };;
val jean : employe = {nom = "Jean"; age = 34; salaire = 2800.}
```

et accéder aux différents champs via la notation pointée *valeur.champ* ou par filtrage :

```
#jean.age;;
- : int = 34
#let combien_gagne e = match e with
    { nom = n; age = a; salaire = s } -> s
#;;
val combien_gagne : employe -> float = <fun>
#combien_gagne jean;;
- : float = 2800.
```

Noter que le filtrage permet d'accéder simultanément à plusieurs champs d'un même enregistrement. Il n'est d'ailleurs pas nécessaire de faire figurer tous ces champs si seuls certains sont à observer[7]. La fonction *combien_gagne*, par exemple, qui n'utilise que le champ *salaire*, aurait pu s'écrire :

```
#let combien_gagne e = match e with
    { salaire = s } -> s;;
val combien_gagne : employe -> float = <fun>
```

Exercice. Définir un type enregistrement pour représenter les nombres complexes, formés d'une partie réelle et d'une partie imaginaire. Ecrire les fonctions d'addition et de multiplication complexe, qui prennent deux nombres complexes et renvoient respectivement la somme et le produit de ces nombres.

1.5.2 Types sommes

A la différence des n-uplets et des enregistrements, qui correspondent à un *produit cartésien* de types, un type somme correspond à une *union* au sens ensembliste de types. Un tel type regroupe sous un même nom un ensemble de types distincts, chacun étant étiqueté avec un *constructeur* spécifique. Ce constructeur sert à la fois à construire les valeurs du type somme et à discriminer (grâce au filtrage) les types membres. Comme pour les types enregistrements, L'utilisation d'un type somme passe par sa définition préalable :

```
#type piece = Pile | Face;;
type piece = Pile | Face
```

Le type *piece* définit deux constructeurs constants (sans arguments). Ces constructeurs[8] peuvent ensuite être utilisés comme n'importe quelle valeur du langage :

```
#let v = Pile;;
val v : piece = Pile
#let essai p = match p with
      Pile -> "Gagne !"
    | Face -> "Perdu !";;
```

[7] Il suffit en fait que le compilateur puisse deviner quel est le type enregistrement concerné.
[8] Dont le nom doit commencer par une majuscule.

```
val essai : piece → string = <fun>
```

```
#essai v;;
```

```
- : string = "Gagne !"
```

Les constructeurs peuvent aussi accepter des arguments. Cela permet de regrouper dans un type somme une collection de types distincts mais reliés entre eux. Exemple :

```
#type couleur =
      Trefle
    | Coeur
    | Carreau
    | Pique;;
```

```
type couleur = Trefle | Coeur | Carreau | Pique
```

```
#type carte =
      As of couleur
    | Roi of couleur
    | Dame of couleur
    | Valet of couleur
    | Petite_carte of int × couleur;;
```

```
type carte =
      As of couleur
    | Roi of couleur
    | Dame of couleur
    | Valet of couleur
    | Petite_carte of int × couleur
```

```
#let c1 = As Trefle;;
```

```
val c1 : carte = As Trefle
```

```
#let c2 = Petite_carte (9, Pique);;
```

```
val c2 : carte = Petite_carte (9, Pique)
```

```
#let valeur_d_une_carte atout carte = match carte with
      As _ → 11
    | Roi _ → 4
    | Dame _ → 3
    | Valet c → if c = atout then 20 else 2
    | Petite_carte (10, _) → 10
    | Petite_carte (9, c) → if c = atout then 14 else 0
    | _ → 0;;
```

```
val valeur_d_une_carte : couleur → carte → int = <fun>
```

```
#valeur_d_une_carte Trefle (Valet Coeur);;
```

```
- : int = 2
```

Chapitre 2

Listes

Les listes – séquences ordonnées d'éléments d'un même type – sont un type de données très souvent utilisé en Caml (et en programmation fonctionnelle en général). Nous nous en servirons notamment pour implémenter certains types de données plus complexes. Le type *list* est prédéfini en Caml.

Les listes peuvent être définies par extension :

#let *l1* = ["What "; "a "; "wonderful "; "World !"];;

val *l1* : *string list* = ["What "; "a "; "wonderful "; "World !"]

ou constructivement, à partir de la liste vide ([]) et du constructeur d'ajout en tête (: :)

#let *l2* = [3; 4];;

val *l2* : *int list* = [3; 4]

#let *l3* = 1 :: *l2*;;

val *l3* : *int list* = [1; 3; 4]

Remarquer la notation post-fixée pour le type liste : *t list* désigne le type d'une liste d'éléments de type *t*. Les fonctions *List.hd* et *List.tl* permettent d'accéder respectivement à la *tête* et à la *queue* d'une liste[1].

#*List.tl l1*;;

- : *string list* = ["a "; "wonderful "; "World !"]

#*List.hd l3*;;

- : *int* = 1

2.1 Quelques fonctions élémentaires sur les listes

La plupart de ces fonctions opèrent par filtrage sur la liste passée en argument.

Le prédicat *empty* renvoie true si son argument est la liste vide, false sinon :

```
#let empty l = match l with
      [] → true
   | x :: xs → false;;
```

[1] Ces fonctions sont définies dans le module `List` de la bibliothèque standard d'`Objective Caml`. La notation pointée *Module.nom* permet d'accéder à la valeur *nom* du module *Module*. On peut aussi « ouvrir » le module avec la directive `#open List`.

val *empty* : α *list* $\rightarrow$ *bool* = <fun>

Remarquer le type inféré par le compilateur pour la fonction *empty* :

$$\alpha \ list \rightarrow \ bool$$

Ici, le paramètre α du constructeur de type *list* doit être compris comme une *variable de type*[2], qui peut être remplacée par n'importe quel type lors de l'application de la fonction. En fait le type de *empty* peut se lire

$$\forall \alpha, \alpha \ list \rightarrow bool$$

La fonction *empty* est dite *polymorphe*. La raison en est ici que cette fonction ne s'occupe pas, en fait, du type des éléments de la liste, mais seulement de la *structure* de cette liste[3]. De fait elle peut opérer sur des listes d'éléments *de n'importe quel type* (*int*, *string*, *int list*, *etc.*) :

#*empty* ["here"; "there"];;

- : *bool* = false

#*empty* [[1; 2]; [3; 3; 3]; []];;

- : *bool* = false

La fonction *length* renvoie la longueur (nombre d'éléments) d'une liste.

```
#let rec length l = match l with
      [] -> 0
    | x :: reste -> 1 + length reste ;;
```

val *length* : α *list* $\rightarrow$ *int* = <fun>

#*length* ["hello"; "world"];;

- : *int* = 2

#*length* [];;

- : *int* = 0

La définition de la fonction *length* est *récursive* ; elle dit en substance :

- si la liste est vide (première clause), la longueur de la liste est 0
- sinon, elle est formée d'au moins un élement, suivi d'une autre liste ; sa longueur est alors 1 + la longueur de cette autre liste. On remarquera l'analogie de cette formulation avec celle d'un raisonnement mathématique par récurrence.

La fonction *rev* « renverse » une liste, c.-à-d. renvoie une liste dont les éléments sont ordonnés dans l'ordre inverse.

```
#let rec rev l = match l with
      [] -> []
    | x :: xs -> (rev xs) @ [x];;
```

val *rev* : α *list* $\rightarrow$ α *list* = <fun>

@ est l'opérateur de concaténation de liste

#[1; 2] @ [3; 4; 5];;

[2] Les variables de type sont dénotées par des lettres grecques (α, β, ...) dans ce document. Le compilateur et l'interpréteur les affichent sous la forme 'a, 'b, ...

[3] Le polymorphisme peut aussi provenir de l'usage d'opérateurs eux-mêmes polymorphiques, comme on le verra plus loin.

- : *int list* = [1; 2; 3; 4; 5]

Le prédicat *member* teste si un élément *e* appartient à une liste *l*

```
#let rec member e l = match l with
      [] -> false
    | x :: xs -> if e = x then true else member e xs;;
val member : 'a -> 'a list -> bool = <fun>
#member 2 [1;2;5];;
- : bool = true
#member "foo" ["toto"; "titi"];;
- : bool = false
```

Remarque : le polymorphisme provient ici du fait que l'opérateur = est lui même polymorphique :

```
#(=);;
- : 'a -> 'a -> bool = <fun>
```

La fonction *list_rem* retire un élément d'une liste.

```
#let rec list_rem e l = match l with
      [] -> []
    | x :: xs -> if e = x then xs else x :: list_rem e xs;;
val list_rem : 'a -> 'a list -> 'a list = <fun>
#list_rem "a" ["bc"; "a"; "def"];;
- : string list = ["bc"; "def"]
```

Il est essentiel de comprendre que la fonction *list_rem* ne modifie pas la liste passée en argument mais renvoie une nouvelle liste, formée des éléments de la liste d'entrée auxquels on a oté l'élément considéré. Cette approche est caractéristique d'un style de programmation *fonctionnel* (ou *applicatif*), dans lequel la notion de valeur *mutable* (« variable ») n'existe pas[4].

```
#let l1 = [1;2;3];;
val l1 : int list = [1; 2; 3]
#let l2 = list_rem 2 l1;;
val l2 : int list = [1; 3]
#l1;;
- : int list = [1; 2; 3]
```

Exercice : on a fait l'hypothèse ici que cet élément à retirer apparaissait au plus une fois dans la liste d'entrée. Quelle modification faut il apporter à la fonction *list_rem* pour qu'elle retire *toutes les occurences* d'un élément *e* dans une liste

[4]En fait, il est possible de définir de telles valeurs mutables en Caml, mais cet aspect n'est délibérément pas abordé ici.

2.2 Fonctionnelles

Considérons la fonction *list_sum*, qui calcule la somme des éléments d'une liste d'entiers :

```
#let add x y = x + y;;
val add : int -> int -> int = <fun>
#let rec list_sum l = match l with
      [] -> 0
    | x :: xs -> add x (list_sum xs);;
val list_sum : int list -> int = <fun>
#list_sum [1;2;3];;
- : int = 6
```

Le schéma de calcul de cette fonction peut être compris en observant les réductions successives lors de l'évaluation :

$$
\begin{array}{ll}
 & sum\_list\ [1;2;3] \\
\Rightarrow & add\ 1\ (sum\_list\ [2;3]) \\
\Rightarrow & add\ 1\ (add\ 2\ (sum\_list\ [3])) \\
\Rightarrow & add\ 1\ (add\ 2\ (add\ 3\ (sum\_list\ [\,]))) \\
\Rightarrow & add\ 1\ (add\ 2\ (add\ 3\ 0)) \\
\Rightarrow & add\ 1\ (add\ 2\ 3) \\
\Rightarrow & add\ 1\ 5 \\
\Rightarrow & 6
\end{array}
$$

Considérons maintenant la fonction qui *list_prod* calcule, elle, le *produit* des éléments d'une liste d'une entier :

```
#let mul x y = x * y;;
val mul : int -> int -> int = <fun>
#let rec list_prod l = match l with
      [] -> 1
    | x :: xs -> mul x (list_prod xs);;
val list_prod : int list -> int = <fun>
#list_prod [4;5;6];;
- : int = 120
```

Les fonctions *list_sum* et *list_prod* sont très similaires. Si les x_i désignent les éléments de la liste d'entrée, f un opérateur (*add* pour *list_sum* et *mul* pour *list_prod*) et z l'élément neutre pour cet opérateur, elles calculent toutes les deux un résultat de la forme

$f\ x_1\ (f\ x_2\ (\ldots (f\ x_n\ z)\ldots))$

soit encore

$x_1 \oplus x2 \oplus \ldots \oplus x_n \oplus z$

en notant $f\ x\ y = x \oplus y$. Ce « schéma de calcul générique » peut être formulé de la manière suivante

- si la liste est vide, renvoyer z,
- sinon, renvoyer $f\ x\ r$, où x désigne la tête de la liste et r le résultat de l'application du schéma de calcul à la queue xs de cette liste.

Il est « visualisé » sur la figure 2.1.

On peut le formuler directement en Caml :

```
#let rec list_fold f z l = match l with
      [] -> z
    | x :: xs -> f x (list_fold f z xs);;
```

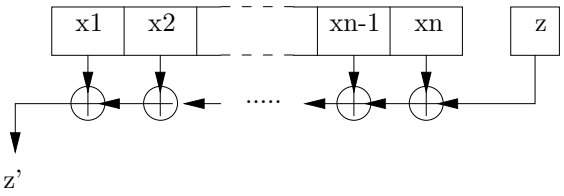

FIG. 2.1 – Un schéma de calcul opérant sur des listes

val *list_fold* : $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha\ list \rightarrow \beta$ = <fun>

La fonction *list_fold* prend trois arguments
- une fonction f, de type $\alpha \rightarrow \beta \rightarrow \beta$, c.-à-d. un opérateur prenant un argument de type α, un argument de type β et renvoyant un résultat de type β,
- un élement de type β,
- une liste d'éléments de type α.

Une telle fonction acceptant une autre fonction en argument est dite d'*ordre supérieur* (on emploie aussi parfois le terme *fonctionnelle*). La possibilité de définir et de manipuler librement des fonctions d'ordre supérieur polymorphiques contribue de manière significative à la puissance d'expression des langages fonctionnels comme Caml. Les fonctions *list_sum* et *list_prod* peuvent ainsi se définir simplement par spécialisation de la fonctionnelle *list_fold* :

```
#let list_sum l = list_fold add 0 l;;

val list_sum : int list -> int = <fun>

#list_sum [1;2;3];;

- : int = 6

#let list_prod l = list_fold mul 1 l;;

val list_prod : int list -> int = <fun>

#list_prod [4;5;6];;

- : int = 120
```

Mais on peut aussi appliquer *list_fold* à une fonction définie « à la volée », par exemple pour calculer $\sum_{i=1}^{i=n} x_i^2$

```
#list_fold (fun x y -> x * x + y) 0 [1; 2; 3; 4];;

- : int = 30
```

Exercice. réécrire la fonction *length* avec la fonctionnelle *list_fold*

Remarque. En `Objective Caml` il existe en fait deux variantes de la fonctionnelle *list_fold* :
- *List.fold_right*, de type $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \alpha\ list \rightarrow \beta \rightarrow \beta$ et telle que *List.fold_right f* [*a1*; ...; *an*] *b* = *f a1* (*f a2* (... (*f an b*) ...)), et
- *List.fold_left*, de type $(\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta\ list \rightarrow \alpha$ et telle que *List.fold_left f a* [*b1*; ...; *bn*] = *f* (... (*f* (*f a b1*) *b2*) ...) *bn*.

La fonctionnelle *list_fold* définie plus haut est équivalente à *List.fold_right*. La définition de *List.fold_left* est la suivante :

```
#let rec fold_left f z l =
    match l with
      [] -> z
    | x :: xs -> fold_left f (f z x) xs;;

val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Exercice : en s'inspirant de la figure 2.1, dessiner le schéma de calcul correspondant à la fonctionnelle *fold_left*.

2.2.1 Deux autres fonctionnelles utiles : map et filter

La fonctionnelle *map* est définie de la manière suivante :

```
#let rec map f l = match l with
      [] -> []
    | x::xs -> f x :: map f xs;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Cette fonctionnelle renvoie la liste obtenue en appliquant une fonction f à tous les éléments d'une liste l. Autrement dit

$$map\ f\ [x_1; \ldots; x_n] = [f(x_1); \ldots; f(x_n)]$$

```
#map (function x -> x * x) [1;2;3;4];;
- : int list = [1; 4; 9; 16]
```

La fonctionnelle *filter* extrait la sous-liste formée des éléments satisfaisant un prédicat p

```
#let rec filter p l = match l with
      [] -> []
    | x::xs -> if p x then x :: filter p xs else filter p xs;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
#filter (function x -> x > 4) [1;5;8;2;10];;
- : int list = [5; 8; 10]
```

Exercice. On appelle *prédicat* toute fonction de type $\alpha \rightarrow$ bool. Une valeur x telle que $p(x) = true$ est dite satisfaisant le prédicat p. Ecrire la fonctionnelle $exists$ qui, étant donné un prédicat p et une liste l renvoie $true$ ssi au moins un élement de l satisfait p. Ecrire la fonctionnelle $forall$ qui, étant donné un prédicat p et une liste l renvoie $true$ ssi tous les éléments de l satisfont p. Quel est le type des fonctionnelles $exists$ et $forall$?

Exercice. Un polynome $a_n x^n + \ldots + a_1 x + a_0$ peut être représenté par la liste $a0; a1; ...; an$ de ses coefficients. Ecrire une fonction d'évaluation de polynome pour cette représentation. Cette fonction prend une valeur flottante x et un polynome P (représenté par la liste de ses coefficients) et renvoie la valeur de $P(x)$. Ecrire une fonction d'addition de polynomes dans cette représentation. Cette fonction prend deux polynomes P_1 et P_2 représentés par la liste de leur coefficients et renvoie la représentation du polynome $P_1 + P_2$ (attention : les degrés de P_1 et P_2 ne sont pas forcément identiques).

2.3 Un mot sur le système de types de Caml

Caml est un langage fortement typé : toute expression possède exactement un type qui définit précisément quelles opérations on peut lui appliquer. Le typage est statique, c.-à-d. que la vérification de cohérence des types est effectuée à la compilation. C'est la garantie qu'un programme ne générera jamais d'erreur de type à l'exécution. D'autres langages (Lisp, Perl, ...) reposent eux sur un typage dynamique, c.-à-d. que la cohérence des types est vérifiée à l'exécution. C'est en ce sens que Caml est un langage *sûr*[5]. Une grande force de Caml est que dans ce cas cette sureté via le typage n'est obtenue

[5] Jamais de *seg fault*, ni de *core dump* !

- ni au prix d'une verbosité accrue du langage : les types n'ont pas à être spécifiés par le programmeur, ils sont *inférés* par le compilateur[6],
- ni au prix d'une limitation de l'expressivité du langage. Au contraire, la notion de type *polymorphe* permet de définir très simplement des fonctions *génériques* très puissantes[7] (comme *length*, *member*, ...)

La paragraphe suivant donne une idée, sur un exemple simple, de la manière dont le compilateur procède pour inférer le type d'une expression (ce paragraphe peut être sauté en première lecture).

Reprenons la définition de la fonction *list_sum*

```
#let rec list_sum l = match l with
      [] -> 0
    | x :: xs -> x + list_sum xs;;
val list_sum : int list -> int = <fun>
```

Pour déduire que *list_sum* a pour type *int list* → *int*, le compilateur procède en gros de la manière suivante[8] :

1. l'argument de *list_sum* peut être []. Il s'agit donc d'une liste. Son type est donc α *list*, où α est inconnu pour l'instant
2. dans la seconde clause, *x :: xs* a aussi pour type α *list*, donc *x* a pour type α et *xs* pour type α *list*
3. la valeur de retour de *list_sum* dans le premier cas est 0, de type *int*
4. dans le second cas, cette valeur est ajoutée à *x* pour produire un *int*. Donc *x* a pour type *int*
5. donc α = *int*
6. donc l'argument de *list_sum* a pour type *int list*
7. donc *list_sum* a pour type *int list* → *int*

Si on on écrit :

```
#let rec list_sum l = match l with
      [] -> 0.0
    | x :: xs -> x +. list_sum xs;;
val list_sum : float list -> float = <fun>
```

le type inféré est *float* → *float list*. Si on écrit :

```
#let rec list_sum l = match l with
      [] -> true
    | x :: xs -> x + list_sum xs;;
Characters 64 - 79 :
      | x :: xs -> x + list_sum xs;;
                   ^^^^^^^^^^^^^^^
This expression has type int but is here used with type bool
```

le système signale fort justement une erreur de typage.

Avec Caml, la démarche du programmeur est donc le plus souvent la suivante : dans un premier temps, on écrit le code de la fonction recherchée. On laisse le compilateur inférer son type puis on vérifie que ce type est conforme à son intuition. Le compilateur peut selon les cas trouver un type plus général que celui attendu[9] ou au contraire moins général[10]. La détection d'une erreur de type est toujours le signe d'un défaut majeur dans la fonction.

[6] A contrario, la nécessité de déclarer explicitement tous les types en C ou Pascal est souvent percue comme une lourdeur.

[7] Dans des langages monomorphes comme C ou C++ la généricité est obtenue au détriment de la sécurité (en « trichant » avec le système de types via le type `void *`) ou en augmentant considérablement la verbosité (via les *templates* du C++).

[8] Pour plus de détails sur l'algorithme d'inférence / vérification de types, se reporter à [6].

[9] Par exemple : α *list* alors qu'on attendait *float list*.

[10] Par exemple : *int list* alors qu'on attendait α *list*.

2.4 Listes d'association

Une liste d'association est une liste de couples (clé,information). Une telle liste permet d'implanter très simplement – sinon très efficacement – un *dictionnaire*, c.-à-d. une structure de données mémorisant un ensemble d'informations indexées par une clé. Les exemples sont nombreux : répertoire téléphonique (un nom → un numéro de téléphone), table de correspondance nom de machine → adresse IP, *etc.* Les trois opérations fondamentales que doit supporter un dictionnaire sont :

- la recherche d'une information par sa clé,
- l'insertion d'une nouvelle information (avec sa clé),
- le retrait d'une information.

Si le dictionnaire est implanté avec une liste d'association, les signatures (c.-à-d. les types) de ces fonctions sont respectivement :

- *lookup* : $\alpha \rightarrow (\alpha, \beta)\ list \rightarrow \beta$, pour la fonction de recherche,
- *insert* : $(\alpha \times \beta) \rightarrow (\alpha, \beta)\ list \rightarrow (\alpha, \beta)\ list$ pour la fonction d'insertion, et
- *remove* : $\alpha \rightarrow (\alpha, \beta)\ list \rightarrow (\alpha, \beta)\ list$ pour la fonction de retrait.

Remarque : on remarquera que les fonctions *insert* et *remove* ne modifient pas la liste passée en argument mais renvoie une nouvelle liste, qui est une copie modifiée de celle ci.

Les fonctions *lookup*, *insert* et *remove* s'écrivent simplement en s'inspirant des fonctionnelles élémentaires vues à la section 2.1. On utilisera en particulier une variante de la fonction *member* pour tester la présence d'une clé donnée dans la liste d'association :

```
#let rec key_present c d = match d with
      [] -> false
    | (k,i)::rest -> c=k || key_present c rest;;
val key_present : 'a -> ('a * 'b) list -> bool = <fun>
```

Le signalement des erreurs (clé absente du dictionnaire – pour les fonctions *remove* et *lookup* – ou déjà présente – pour la fonction *insert*) se fera en déclenchant deux exceptions, définies avec le mot clé *Exception*.

```
#exception Key_not_found;;
exception Key_not_found
#exception Key_exists;;
exception Key_exists
#let rec lookup c d = match d with
      [] -> raise Key_not_found
    | (k,i)::rest -> if k=c then i else lookup c rest;;
val lookup : 'a -> ('a * 'b) list -> 'b = <fun>
#let rec insert (c,i) d =
    if key_present c d then raise Key_exists
    else (c,i) :: d;;
val insert : 'a * 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>
#let rec remove c d = match d with
      [] -> raise Key_not_found
    | (k,i)::rest -> if c=k then rest else (k,i) :: remove c rest;;
val remove : 'a -> ('a * 'b) list -> ('a * 'b) list = <fun>
#let d1 = [("pierre",54); ("luc",18); ("jean", 23)];;
val d1 : (string * int) list = [("pierre", 54); ("luc", 18); ("jean", 23)]
```

```
#lookup "jean" d1;;
- : int = 23
#lookup "marc" d1;;
Exception : Key_not_found.
#let d2 = insert ("marc",60) d1;;
val d2 : (string × int) list =
    [("marc", 60); ("pierre", 54); ("luc", 18); ("jean", 23)]
#lookup "marc" d2;;
- : int = 60
#let d3 = remove "jean" d2;;
val d3 : (string × int) list = [("marc", 60); ("pierre", 54); ("luc", 18)]
#lookup "jean" d3;;
Exception : Key_not_found.
```

Chapitre 3

Arbres

Les listes constituent une structure de données *linéaire* : chaque élément n'est lié qu'à son successeur. Pour certains algorithmes, cette structure est inadéquate et une structure *arborescente* est plus adaptée. Un *arbre* est un ensemble fini d'éléments liés par une relation dite « de parenté » telle que

- un élément et un seul ne possède pas de père (cet élement est appelé *racine* de l'arbre),
- tout élément (à part la racine) possède exactement un père.

On représente en général les arbres par un dessin où chaque élément est lié à ses fils par un trait descendant[1], comme sur la figure 3.1.

Quelques définitions :

- les élements d'un arbre sont appelés nœuds. Les éléments sans fils sont appelés nœuds terminaux ou *feuilles* ;
- la *taille* d'un arbre est le nombre de nœuds dans cet arbre ;
- la *profondeur* d'un nœud est le nombre d'ascendants de ce nœud. La profondeur maximale des nœuds d'un arbre est la *hauteur* de cet arbre ;
- le *degré* d'un nœud est son nombre de fils ;
- l'ensemble des descendants d'un nœud n forme un *sous-arbre* de racine n ; une *forêt* est un ensemble fini d'arbres sans nœud commun ; l'ensemble des descendants stricts d'un nœud n contitue une forêt ; les arbres de cette forêt sont les *branches* issues de n ;
- un arbre est dit *ordonné* si l'ordre des branches d'un nœud n est significatif ;
- un arbre *n-aire* est un arbre ordonné pour lequel tout nœud non-terminal possède exactement n branches ;
- un arbre est *étiqueté* si ses nœuds portent une information

[1] Les arbres informatiques poussent vers le bas !

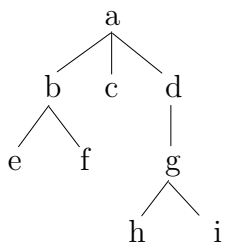

FIG. 3.1 – Exemple d'arbre

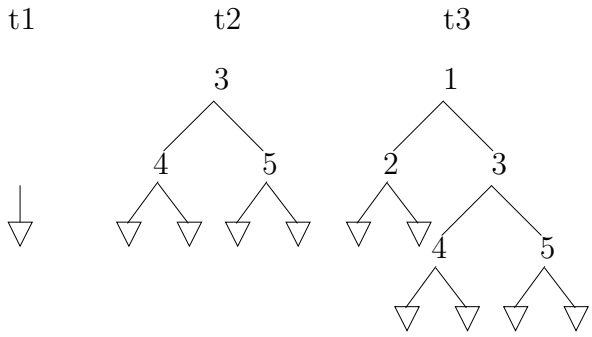

FIG. 3.2 – Exemples d'arbres

3.1 Arbres binaires

On s'intéresse ici aux arbres *binaires*, c.-à-d. aux arbres pour lesquels chaque nœud possède exactement deux branches. Ces branches sont classiquement appelées *branche gauche* et *branche droite*. Pour les feuilles, les branches gauche et droite sont vides.

Contrairement aux listes, il n'existe pas de type arbre prédéfini en Caml. Mais il est très facile de créer soi-même un tel type en utilisant une *définition de type*. Voici par exemple la définition d'un type *int_btree* pour des arbres binaires étiqueté par des entiers

```
#type int_btree =
      Empty
    | Node of int_btree × int × int_btree;;

type int_btree = Empty | Node of int_btree × int × int_btree
```

Cette définition ne crée pas de valeur. Elle introduit un nouveau type (le type *int_btree*). Elle peut se lire de la manière suivante : un arbre binaire d'entiers est soit vide (*Empty*), soit formé d'un nœud racine (*Node*) auquel sont attachés un entier et deux sous-arbres d'entiers (la branche gauche et la branche droite). Dans cette définition, *Empty* et *Node* sont des noms arbitraires, choisis par le programmeur[2]. On les appelle *constructeurs de types*. Le constructeur *Empty* n'attend pas d'argument. Le constructeur *Node* attend — c'est la signification du mot-clé of — trois arguments, de type *int*, *int_btree* et *int_btree* respectivement. Ces deux constructeurs sont utilisés pour fabriquer des objets de type *int_btree*

```
#let t1 = Empty;;

val t1 : int_btree = Empty

#let t2 = Node (Node (Empty, 4, Empty), 3, Node (Empty, 5, Empty));;

val t2 : int_btree = Node (Node (Empty, 4, Empty), 3, Node (Empty, 5, Empty))

#let t3 = Node (Node (Empty, 2, Empty), 1, t2);;

val t3 : int_btree =
    Node (Node (Empty, 2, Empty), 1,
     Node (Node (Empty, 4, Empty), 3, Node (Empty, 5, Empty)))
```

Les constructeurs vont aussi servir à accéder, via le mécanisme de *filtrage* (*pattern-matching*), aux éléments des arbres ainsi construits.

```
#let root_label t = match t with
```

[2]On aurait pu tout aussi bien les appeler *Vide* et *Noeud* (ou *A* et *B*), la seule contrainte étant que ces noms commencent par une majuscule.

```
      Empty → failwith "root_label : empty tree"
    | Node (l, x, r) → x;;
val root_label : int_btree → int = <fun>
#let left_subtree t = match t with
      Empty → failwith "left_subtree : empty tree"
    | Node (l, x, r) → l;;
val left_subtree : int_btree → int_btree = <fun>
#let right_subtree t = match t with
      Empty → failwith "right_subtree : empty tree"
    | Node (l, x, r) → r;;
val right_subtree : int_btree → int_btree = <fun>
#root_label t2;;
- : int = 3
#left_subtree t2;;
- : int_btree = Node (Empty, 4, Empty)
#right_subtree t3;;
- : int_btree = Node (Node (Empty, 4, Empty), 3, Node (Empty, 5, Empty))
```

On remarquera la similitude avec les définitions utilisées pour les listes, les constructeurs *Empty* et *Node* jouant le rôle des constructeurs [] et : : pour les listes. Un arbre binaire peut être vu comme une liste à deux queues (la branche droite et la branche gauche). Réciproquement, une liste peut être vue comme un arbre unaire.

Le type *int_btree* est monomorphe : les arbres de ce type ne peuvent être étiquetés que par des entiers. Il est possible de rendre ce type polymorphe, et donc de définir des arbres binaires génériques, en introduisant un *paramètre de type* dans la définition de type

```
#type α btree =
      Empty
    | Node of α btree × α × α btree;;
type α btree = Empty | Node of α btree × α × α btree
```

Le type *int_btree* apparait alors comme un pas particulier du type α *btree* :

```
#let t2 = Node ( Node (Empty, 4, Empty), 3, Node (Empty, 5, Empty));;
val t2 : int btree = Node (Node (Empty, 4, Empty), 3, Node (Empty, 5, Empty))
```

Mais on peut alors définir des arbres binaires étiquetés avec des valeurs de type quelconque :

- chaine de caractères :

```
#let t4 = Node ( Node(Empty, "its", Empty), "hello", Node(Empty, "me", Empty));;
val t4 : string btree =
    Node (Node (Empty, "its", Empty), "hello", Node (Empty, "me", Empty))
```

- listes :

```
#let t5 = Node ( Node (Empty, [5;6;7], Empty), [3;4], Node (Empty, [8;9], Empty));;
val t5 : int list btree =
    Node (Node (Empty, [5; 6; 7], Empty), [3; 4], Node (Empty, [8; 9], Empty))
```

– ou même fonctions :

```
#let t6 = Node (
    Node(Empty, (function x -> x * 2), Empty),
    (function x -> x),
    Node(Empty, (function x -> x * 4), Empty));;

val t6 : (int -> int) btree =
    Node (Node (Empty, <fun>, Empty), <fun>, Node (Empty, <fun>, Empty))
```

3.1.1 Quelques opérations sur les arbres binaires

La fonction *btree_size* renvoie la taille d'un arbre binaire, c.-à-d. le nombre d'éléments qu'il contient :

```
#let rec btree_size t = match t with
      Empty -> 0
    | Node (l, _, r) -> 1 + btree_size l + btree_size r;;

val btree_size : α btree -> int = <fun>

#btree_size t5;;

- : int = 3
```

La fonction *btree_depth* renvoie la profondeur d'un arbre binaire

```
#let rec btree_depth t = match t with
      Empty -> 0
    | Node (l, _, r) -> 1 + max (btree_depth l) (btree_depth r);;

val btree_depth : α btree -> int = <fun>

#btree_depth t5;;

- : int = 2
```

La fonction *btree_mem* permet de tester si un élement *e* appartient à un arbre :

```
#let rec btree_mem e t = match t with
      Empty -> false
    | Node (l, x, r) -> x = e ∨ btree_mem e l ∨ btree_mem e r;;

val btree_mem : α -> α btree -> bool = <fun>

#btree_mem "me" t4;;

- : bool = true

#btree_mem "you" t4;;

- : bool = false
```

La fonction *btree_mirror* renvoie la version « miroir » d'un arbre binaire, c.-à-d. un arbre pour lequel toutes les branches gauche et droite ont été interchangées :

```
#let rec btree_mirror t = match t with
      Empty -> Empty
    | Node(l, x, r) -> Node(btree_mirror r, x, btree_mirror l);;

val btree_mirror : α btree -> α btree = <fun>

#btree_mirror t4;;
```

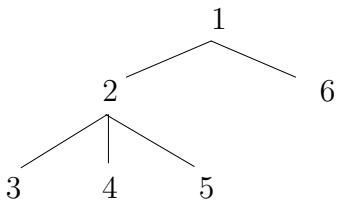

FIG. 3.3 – Exemple d'arbre à arité variable

```
- : string btree =
Node (Node (Empty, "me", Empty), "hello", Node (Empty, "its", Empty))
```

Comme pour les listes, il est utile de définir des *fonctionnelles* génériques opérant sur les arbres binaires. La première, *map_btree*, renvoie l'arbre obtenu en appliquant une fonction f à toutes les étiquettes d'un arbre :

```
#let rec map_btree f t = match t with
      Empty -> Empty
    | Node(l,x,r) -> Node(map_btree f l, f x, map_btree f r);;
val map_btree : ('a -> 'b) -> 'a btree -> 'b btree = <fun>
#map_btree String.length t4;;
- : int btree = Node (Node (Empty, 3, Empty), 5, Node (Empty, 2, Empty))
```

La seconde, *fold_btree*, est analogue à la fonctionnelle *list_fold* vue au chapitre précédent. Elle permet de définir un *homomorphisme* $\mathcal{H}$ sur les arbres binaires, c.-à-d. une opération f – ternaire, cette fois – et une valeur z telle que :

- $\mathcal{H}(Empty) = z$
- $\mathcal{H}(Node(l,x,r)) = f(\mathcal{H}(l), x, \mathcal{H}(r))$

```
#let rec fold_btree f z t = match t with
      Empty -> z
    | Node(l,x,r) -> f (fold_btree f z l) x (fold_btree f z r);;
val fold_btree : ('a -> 'b -> 'a -> 'a) -> 'a -> 'b btree -> 'a = <fun>
#fold_btree (fun l x r -> l+x+r) 0 t2;;
- : int = 12
```

Exercice : redéfinir les fonctions *btree_size*, *btree_depth*, *btree_mirror* et *btree_mem* avec la fonctionnelle *btree_fold*.

Remarque : la version définie de *fold_btree* ici suppose que l'opérateur f est associatif et commutatif. Le cas contraire, l'ordre de parcours de l'arbre devient significatif et il faut définir des versions spécifiques des *fold_btree* (à la manière des fonctionnelles *fold_right* et *fold_left* pour les listes).

Exercice : l'implantation des arbres n-aires – et plus généralement des arbres à arité non fixe[3] peut se faire avec le type suivant :

```
type 'a vtree = Node of 'a * ('a vtree) list
```

L'ensemble des fils d'un nœud est ici représenté par une liste et pour les nœuds terminaux cette liste est vide. Donner la représentation, sous la forme d'une valeur de type *int vtree* de l'arbre de la figure 3.3.

Ecrire et donner le type de la fonction *vtree_size*, qui renvoie le nombre de nœuds d'un arbre à arité variable.

Ecrire et donner le type de la fonction *vtree_depth*, qui renvoie la profondeur d'un arbre à arité variable.

[3] C.à.d. pour lesquels les nœuds peuvent avoir un nombre variable de fils.

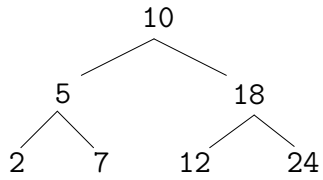

FIG. 3.4 – Un arbre binaire de recherche

3.2 Arbres binaires de recherche

On a vu au chapitre précédent comment implanter un dictionnaire sous la forme d'une liste d'association. Mais, même avec des listes ordonnées, la complexité des opérations de recherche, d'insertion et de retrait restait en $O(n)$ (avec une liste ordonnée, ces opérations requièrent en moyenne $n/2$ accès à la liste). On peut se ramener à un cout en $O(log(n))$ en utilisant des *arbres binaires de recherche*.

Un arbre binaire de recherche (ABR, ou BST – pour *Binary Search Tree* – en anglais) est un arbre binaire possédant la propriété suivante : *l'information portée par chaque nœud (l'étiquette) est supérieure – au sens d'une relation d'ordre – à celle portée par tous les nœuds de la branche gauche issue de ce nœud et inférieure – au sens de cette même relation – à celle portée par tous les nœuds de la branche droite (P1)*. La figure 3.4 donne un exemple d'ABR d'entiers.

La recherche d'un élément dans un ABR requiert au maximum N comparaisons, où N est la profondeur maximale de l'arbre. Pour un arbre binaire complet de taille T (*i.e.* pour lequel chaque nœud possède exactement 0 ou 2 fils), on a $N = log_2 T$ (au lieu de $N = T$ pour un arbre quelconque ou une liste). En effet, la stratégie de recherche peut se formuler de la manière suivante :

- si l'élément recherché est celui porté par la racine de l'arbre, la recherche est terminée,
- sinon, si cet élément est inférieur à celui porté par la racine, il faut le recherche dans la branche gauche issue de la racine, sinon dans la branche droite.

A chaque étape, la profondeur du nœud examiné augmente au moins de un. Le nombre d'étapes est donc borné par la profondeur maximale de l'arbre.

L'implantation en Caml d'un dictionnaire avec un ABR utilise la représentation des arbres binaires introduite au chapitre précédent. L'information portée par chaque nœud est ici un couple (clé,information) :

```
#type (α,β) bst =
      Empty
    | Node of (α,β) bst × (α × β) × (α,β) bst;;

type (α, β) bst = Empty | Node of (α, β) bst × (α × β) × (α, β) bst
```

La fonctions *key_present* est une traduction directe de la stratégie de recherche énoncée plus haut :

```
#let rec key_present c d = match d with
      Empty → false
    | Node(left,(k,i),right) →
          if c = k then true
          else if c < k then key_present c left
          else key_present c right;;

val key_present : α → (α, β) bst → bool = <fun>
```

La fonction *bst_insert* d'insertion dans un ABR doit produire un arbre dans lequel le couple (clé,information) a été inséré « à la bonne place », c.-à-d. telle que les clés de l'arbre résultat vérifient la propriété (P1). Cette condition est facile à satisfaire dans le cas où l'arbre d'entrée est vide : il suffit de renvoyer un arbre formé d'un seul nœud portant le couple à insérer. Le cas contraire, l'insertion se fera dans le sous-arbre gauche ou droit selon que la valeur présente à la racine est supérieure ou inférieure à la valeur à insérer :

```
#exception Key_exists;;
```

```
exception Key_exists

#let rec insert (c,i) d = match d with
      Empty -> Node(Empty, (c,i), Empty)
    | Node(left,(k,j),right) ->
        if c = k then raise Key_exists
        else if c < k then Node(insert (c,i) left, (k,j), right)
        else Node(left, (k,j), insert (c,i) right);;

val insert : α * β -> (α, β) bst -> (α, β) bst = <fun>
```

La fonction *make* utilise la fonctionnelle *fold_right* vue au Chap. 2 et *insert* pour construire un ABR à partir d'une liste de couples (clé,information) :

```
#let make l = List.fold_right insert l Empty;;

val make : (α * β) list -> (α, β) bst = <fun>

#let d1 = make [ ("xavier",6); ("marc",35); ("pierre",56); ("jean", 23); ("luc",17) ];;

val d1 : (string, int) bst =
  Node (Node (Empty, ("jean", 23), Empty), ("luc", 17),
    Node (Node (Empty, ("marc", 35), Empty), ("pierre", 56),
      Node (Empty, ("xavier", 6), Empty)))
```

La fonction *remove*, assurant le retrait d'un élément d'un ABR est un peu plus subtile. On peut la décomposer de la manière suivante :

- si l'arbre d'entrée est vide, renvoyer un arbre vide ;
- si la valeur à retirer se trouve à la racine et qu'une des deux branches issues de cette racine est vide, renvoyer l'autre branche ;
- si la valeur à retirer se trouve à la racine et qu'aucune des deux branches issues de cette racine n'est vide, alors il faut remplacer la valeur présente à la racine par une autre valeur ; cette autre valeur peut être le plus grand élément de la branche gauche ou le plus petit élément de la branche droite ; on choisira ici la deuxième solution ;
- si la valeur à retirer ne se trouve pas à la racine de l'arbre d'entrée, on cherche à la retirer de la branche gauche ou droite selon qu'elle est inférieure ou supérieure à cette valeur.

La formulation précédente fait apparaître la nécessité d'une fonction *min_key*, renvoyant le couple portant la plus petite clé (au sens de la relation d'ordre) d'un ABR, supposé non-vide. Ce couple est, par construction, celui porté par le nœud situé « le plus à gauche » de l'arbre. La fonction *key_min* peut donc s'écrire :

```
#let rec key_min t = match t with
      Empty -> failwith "this should not happen"
    | Node(Empty, x, r) -> x
    | Node(left, _, right) -> key_min left;;

val key_min : (α, β) bst -> α * β = <fun>

#key_min d1;;

- : string * int = ("jean", 23)
```

Exercice : écrire la fonction duale *key_max*, qui renvoie le plus grand élément d'un ABR supposé non-vide.

On peut désormais écrire la fonction *remove* de retrait d'un élément d'un dictionnaire représenté par un ABR :

```
#let rec remove c d = match d with
```

```
      Empty → Empty
    | Node(left,(k,i),right) →
        if k = c then begin match (left,right) with
              (Empty, r) → r
            | (l, Empty) → l
            | (l, r) →
                let (cm,im) = key_min r in
                Node(l, (cm,im), remove cm r)
        end
        else if c < k then Node(remove c left, (k,i), right)
        else Node(left, (k,i), remove c right);;

val remove : α → (α, β) bst → (α, β) bst = <fun>

#let d2 = remove "xavier" d1;;

val d2 : (string, int) bst =
    Node (Node (Empty, ("jean", 23), Empty), ("luc", 17),
     Node (Node (Empty, ("marc", 35), Empty), ("pierre", 56), Empty))

#let d3 = remove "marc" d2;;

val d3 : (string, int) bst =
    Node (Node (Empty, ("jean", 23), Empty), ("luc", 17),
     Node (Empty, ("pierre", 56), Empty))
```

Remarque : la formulation donnée ci-dessus pour la fonction *remove* n'est pas optimale. En effet, dans le cas où $c = k$ et où ni la branche gauche ni la branche droite ne sont vides, la branche droite est visitée deux fois : une première fois par la fonction *key_min* pour obtenir le plus petit élément, et une seconde fois (récursivement) par la fonction *remove* pour retirer cet élément. Il est plus judicieux d'effectuer ces deux opérations en une fois. C'est ce que fait la fonction *rem_key_min* donnée ci-dessous. Etant donné un ABR t, supposé non-vide, cette fonction renvoie un couple dont le premier élément est la valeur minimum de t et le second l'arbre t auquel on a retiré cet élément :

```
#let rec rem_key_min t = match t with
      Empty → failwith "this should not happen"
    | Node(Empty,x,r) → (x,r)
    | Node(l,x,r) →
        let (m,l') = rem_key_min l in
        (m, Node(l',x,r));;

val rem_key_min : (α, β) bst → (α × β) × (α, β) bst = <fun>
```

On peut alors réécrire la fonction *remove* :

```
#let rec remove c d = match d with
      Empty → Empty
#| Node(left,(k,i),right) →
        if c = k then begin match (left,right) with
              (Empty, r) → r
            | (l, Empty) → l
            | (l, r) →
                let x',r' = rem_key_min r in
                Node(l, x', r')
        end
        else if c < k then Node(remove c left, (k,i), right)
        else Node(left, (k,i), remove c right);;
```

val *remove* : $\alpha \rightarrow (\alpha, \beta)\ bst \rightarrow (\alpha, \beta)\ bst$ = <fun>

Chapitre 4

Graphes

Les graphes sont des structures de données très fréquemment rencontrées en traitement de l'information. Ils peuvent par exemple servir à modéliser

- le maillage d'une région de l'espace pour la résolution discrète d'équations différentielles,
- le résultat d'une segmentation d'image au sens région,
- un réseau d'hypothèses pour la reconnaissance de formes,
- ...

Ce chapitre n'a pas prétention à faire une revue complète des algorithmes opérant sur les graphes, qui sont nombreux, et pour certains complexes. Il s'agit avant tout de montrer comment les principes de programmation abordés dans les chapitres précédents permettent de fournir une implémentation extrêmement concise d'une telle structure de données et des principales opérations élémentaires qui y sont associées. La fin de ce chapitre introduit par ailleurs la notion de *type de données abstrait* (TDA) à travers la notion de *signature* en Caml.

4.1 Définitions

Formellement, un graphe est défini par deux ensemble : un ensemble S de *sommets* et un ensemble E d'*arcs*, un arc reliant deux sommets. Un graphe est dit *orienté* si chaque arc possède une direction. Le sommet de départ d'un arc est alors appelé *source* et celui d'arrivée *destination*. Une suite d'arcs permettant de « passer » d'un sommet à un autre est appelée *chemin*. Le nombre d'arcs composant ce chemin est sa longueur. Un sommet s_2 est dit *atteignable* à partir d'un sommet s_1 s'il existe un chemin de s_1 à s_2. Un chemin est dit simple s'il ne passe pas deux fois par le même sommet. Un *cycle* est un chemin d'un sommet vers lui même. Les graphes peuvent être *valués* en associant un valeur à chaque arc. On peut alors définir le *coût* d'un chemin comme la somme des valeurs arcs qui le composent.

Dans ce chapitre on s'intéressera uniquement aux graphes orientés finis (c.-à-d. dont le nombre de sommets est fini) non valués.

4.2 Représentation

Il y a de multiples manières de représenter concrêtement un graphe :

- avec une *matrice d'adjacence*, c.-à-d. une matrice M carrée de taille $N \times N$, où N est le nombre de sommets, et où $M_{i,j}$ vaut 1 ssi il existe un arc du sommet s_i au sommet s_j[1] ;
- via une fonction de transition, qui associe à chaque sommet la liste de ses successeurs ;

[1] Dans le cas d'un graphe valué, on donne à $M_{i,j}$ la valeur de l'arc correspondant ou une valeur par défaut si cet arc n'existe pas.

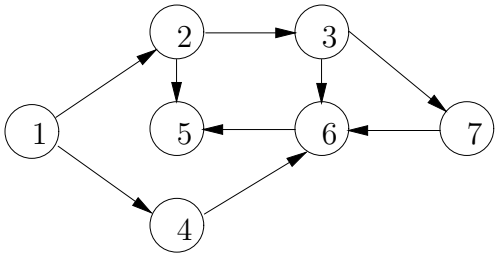

FIG. 4.1 – Exemple de graphe

- comme une collection de sommets, chaque sommet étant associé à la liste de ses successeurs, c.-à-d. l'ensemble des sommets directement atteignables depuis ce sommet ;
- comme une collection d'arcs.

Chaque approche a ses avantages et ses inconvénients, qui dépendent essentiellement du type d'algorithmes que l'on veut appliquer au graphe. L'approche matricielle, par exemple, donne lieu à des implémentations très efficaces (lorsqu'il s'agit de détecter des cycles par exemple) mais s'accomode mal des graphes dont le nombre de sommets varie. L'approche par fonction de transition, en revanche ne requiert pas la construction explicite du graphe et est donc bien adaptée aux algorithmes de recherche dans des graphes de taille potentiellement infinie. Nous décrivons ici la technique consistant à représenter un graphe comme l'ensemble de ses arcs. Cette technique a le mérite d'être simple et de conduire à des algorithmes de recherche raisonnablement efficaces pour des graphes de taille pas trop importante. Le graphe de la figure 4.1, par exemple, sera représenté par la liste suivante :

#let *g1* = [(1,2); (2,3); (2,5); (3,6); (3,7); (6,5); (7,6); (1,4); (4,6)];;

val *g1* : (*int* × *int*) *list* =
[(1, 2); (2, 3); (2, 5); (3, 6); (3, 7); (6, 5); (7, 6); (1, 4); (4, 6)]

Un graphe est donc vu ici comme une simple liste de couples, chaque couple correspondant à un arc. La première composante de chaque couple identifie le sommet source, la seconde le sommet destination.

La fonctions *is_edge* teste si un arc appartient à un graphe :

#let *is_edge g* (*v1*, *v2*) = *List.mem* (*v1*, *v2*) *g*;;

val *is_edge* : (α × β) *list* → α × β → *bool* = <fun>

La fonction *add_edge* permet d'ajouter un arc à un graphe, en vérifiant au préalable que cet arc n'existe pas déjà :

#exception *Edge_exists*;;

exception *Edge_exists*

#let *add_edge g* (*v1*, *v2*) =
  if *is_edge g* (*v1*, *v2*) then *raise Edge_exists*
  else (*v1*, *v2*) :: *g*;;

val *add_edge* : (α × β) *list* → α × β → (α × β) *list* = <fun>

La fonction *is_vertex* teste si un sommet appartient à un graphe :

#let *is_vertex g v* = *List.exists* (function (*v1*, *v2*) → *v* = *v1* ∨ *v* = *v2*) *g*;;

val *is_vertex* : (α × α) *list* → α → *bool* = <fun>

La fonction *rem_edge* permet de retirer un arc d'un graphe. Cette fonction déclenche l'exception *Edge_not_found* si l'arc spécifié n'appartient pas au graphe.

```
#exception Edge_not_found;;

exception Edge_not_found

#let rec rem_edge g (v1,v2) = match g with
    [] -> raise Edge_not_found
#| (v1',v2')::rest -> if v1 = v1' & v2 = v2' then rest else rem_edge rest (v1,v2);;

val rem_edge : ('a * 'b) list -> 'a * 'b -> ('a * 'b) list = <fun>
```

Les fonctions *succ* et *pred* renvoient respectivement les successeurs et prédécesseurs immédiats d'un sommet donné, c.-à-d. l'ensemble des sommets atteignables par un chemin de longueur 1 depuis ce sommet et, respectivement, l'ensemble des sommets depuis lesquels ce sommet est atteignable par un chemin de longueur 1

```
#let rec succ g v = match g with
    [] -> []
#| (v1,v2)::rest -> if v1 = v then v2 :: succ rest v else succ rest v;;

val succ : ('a * 'b) list -> 'a -> 'b list = <fun>

#let rec pred g v = match g with
    [] -> []
#| (v1,v2)::rest -> if v2 = v then v1 :: pred rest v else pred rest v;;

val pred : ('a * 'b) list -> 'b -> 'a list = <fun>

#succ g1 3;;

- : int list = [6; 7]

#pred g1 6;;

- : int list = [3; 7; 4]
```

La fonction *vertices* renvoie la liste des sommets d'un graphe. Elle utilise la fonctionnelle *fold_left* pour itérer sur la liste des arcs et accumuler les sommets non déjà rencontrés dans une liste.

```
#let vertices g =
    let update acc (v1,v2) =
      let acc' = if List.mem v1 acc then acc else v1::acc in
      if List.mem v2 acc' then acc' else v2::acc' in
    List.fold_left update [] g;;

val vertices : ('a * 'a) list -> 'a list = <fun>

#vertices g1;;

- : int list = [4; 7; 6; 5; 3; 2; 1]
```

Les fonctions *inits* et *terms* donnent respectivement la liste des sommets initiaux (sans prédecesseur) et terminaux (sans successeur) d'un graphe :

```
#let inits g = List.filter (function v -> pred g v = []) (vertices g);;

val inits : ('a * 'a) list -> 'a list = <fun>

#let terms g = List.filter (function v -> succ g v = []) (vertices g);;

val terms : ('a * 'a) list -> 'a list = <fun>

#inits g1;;

- : int list = [1]
```

```
#terms g1;;
- : int list = [5]
```

Exercice. Une autre représentation possible pour les graphes consiste à lister les sommets et, pour chaque sommet, la liste de ces successeurs (le graphe est alors vu comme une liste d'association entre les sommets et leurs successeurs). Le graphe *g1* défini plus haut, par exemple, peut se décrire de la manière suivante :
let *g1* = [(1, [2; 4]); (2, [3; 5]); (3, [6; 7]); (4, [6]); (5, [ ]); (6, [5]); (7, [6])]
Avec cette représentation, donner les définitions des fonctions *is_vertex*, *is_edge*, *rem_edge*, *succ* et *pred*, *vertices*, *inits* et *terms* introduites plus haut

4.3 Quelques algorithmes

4.3.1 Algorithmes de parcours

Le but de ces algorithmes est de produire la liste des sommets atteignables depuis un sommet donné. Chacun de ces sommets doit apparaître exactement une fois. L'ordre d'apparition des sommets dans la liste dépend du type de parcours :

- avec un parcours **en profondeur d'abord** (*depth-first traversal*), on suit récursivement chaque chemin issu du sommet initial. Lorsqu'on atteint un sommet sans successeur – ou dont tous les successeurs ont déjà été vus – on revient au dernier sommet rencontré pour lequel il restait des successeurs à explorer.
- avec un parcours **en largeur d'abord** (*breadth-first traversal*), on commence par lister tous les sommets successeurs au sommet de départ, puis tous les successeurs de ces successeurs, *etc.*.

La fonction *dft* ci-dessous implémente un parcours en profondeur d'abord. Le parcours lui même est réalisé par la fonction récursive *trav*, qui accepte sur deux listes de sommets :

- la première (*visited*) garde trace des sommets déjà visités,
- la seconde (*vertices*) est la liste des sommets encore à traiter.

A chaque appel de *trav*, on commence par regarder si le prochain sommet à traiter (au début de la liste *vertices*) a déjà été visité. Si oui, on poursuit le parcours. Si non, on ajoute ce sommet à la liste des sommets visités et on ajoute la liste de ses successeurs en tête de la liste des sommets à traiter. Le parcours se termine lorsque cette liste est vide. La liste des sommets visités décrit alors le parcours recherché (dans l'ordre inverse).

```
#let dft g v =
    let rec trav visited vertices = match vertices with
      [] → List.rev visited
    | v :: vs →
        if List.mem v visited then trav visited vs
        else trav (v :: visited) ((succ g v)@vs) in
    trav [] [v];;
val dft : (α × α) list → α → α list = <fun>
#dft g1 1;;
- : int list = [1; 2; 3; 6; 5; 7; 4]
```

La fonction *bft* implémente un parcours en largeur d'abord. Elle opère de manière similaire à *dft* sauf que la liste de successeurs d'un sommet qui vient d'être visité est cette fois ajoutée *en queue* de la liste des sommets à traiter. De cette manière, on assure que les tous les successeurs d'un sommet seront visités avant que leurs propres successeurs le soient.

```
#let bft g v =
    let rec trav visited vertices = match vertices with
      [] → List.rev visited
    | v :: vs →
```

```
        if List.mem v visited then trav visited vs
        else trav (v :: visited) (vs@(succ g v)) in
    trav [] [v];;
val bft : ('a * 'a) list -> 'a -> 'a list = <fun>
#bft g1 1;;
- : int list = [1; 2; 4; 3; 5; 6; 7]
```

4.3.2 Recherche de cycle

Un cycle est chemin d'un sommet vers lui-même de longueur non-nulle. La fonction de parcours en profondeur d'abord définie plus haut peut être modifiée afin de détecter de tels cycles. Il suffit pour cela, lorqu'on « descend » le long d'un chemin issu d'un sommet s de garder trace de tous les sommets visités lors de cette descente (on appelle *orbite* de s la séquence de ces sommets). Lorsqu'on s'apprête à appeler récursivement la fonction de descente sur les successeurs directs d'un sommet, on teste leur appartenance à cette orbite, ce qui, le cas échéant traduit l'existence d'un cycle.

Les deux fonctions suivantes implémentent cet algorithme. La première est une variante de la fonction *trav* définie plus haut pour *dft*. Elle prend une liste de sommets déjà visités, une orbite (c.-à-d. la liste des sommets visités pour la descente récursive en cours) et une liste de successeurs à explorer. Elle renvoie une liste de sommets visités ou déclence l'exception *Cycle* si un cycle est détecté pendant la descente.

```
#exception Cycle;;
exception Cycle
#let rec dft_c g visited orbit vertices = match vertices with
      [] -> visited
    | v :: vs ->
        let visited' =
          if List.mem v orbit then raise Cycle
          else if List.mem v visited then visited
          else dft_c g (v :: visited) (v :: orbit) (succ g v) in
        dft_c g visited' orbit vs;;
val dft_c : ('a * 'a) list -> 'a list -> 'a list -> 'a list -> 'a list =
    <fun>
```

La fonction *dft_c* est appelée par la fonction *has_cycle* pour chaque sommet initial du graphe :

```
#let has_cycle g =
    let rec trav visited vertices = match vertices with
      [] -> false
    | v :: vs ->
        if List.mem v visited then trav visited vs
        else
          try
            let visited' = dft_c g visited [v] (succ g v) in
            trav visited' vs
          with
            Cycle -> true in
    trav [] (inits g);;
val has_cycle : ('a * 'a) list -> bool = <fun>
#has_cycle g1;;
```

- : *bool* = false

#has_cycle [(1, 2); (2, 4); (4, 3); (3, 2); (1, 3)];;

- : *bool* = true

Exercice. Modifier la fonction *has_cycle* (et *dft_c*) de telle sorte que celle-ci renvoie, si un cycle est détecté, l'index du sommet au niveau duquel ce cycle a été détecté.

4.3.3 Tri topologique

Un tri topologique consiste à assigner un ordre linéaire aux sommets d'un graphe de telle sorte que s'il existe un arc du sommet i au sommet j, i apparaisse avant j dans cet ordre. Par exemple, [1 ;4 ;2 ;3 ;7 ;6 ;5] est un tri topogique du graphe de la figure 4.1. Il facile d'obtenir un tri topologique à partir d'un parcours en profondeur d'abord : il suffit de noter le sommet courant dès que l'ensemble de ses successeurs ont été explorés. On utilise pour cela une liste supplémentaire (*sorted*). La fonction *tsort* ci-dessous implémente cet algorithme. A chaque appel à *trav*, si le sommet courant (v) n'a pas déjà été visité, on explore l'ensemble de ses successeurs (en mettant à jour la liste des sommets visités et celle des sommets triés), puis on ajoute v à la liste des sommets triés et on continue. Lorsque la liste des sommets à explorer est vide, la liste triée constitue le résultat. Le tri topologique n'est pas défini dès que le graphe contient un cycle. La fonction *tsort* vérifie donc cette condition et déclenche une exception le cas échéant.

```
#let tsort g s =
    let rec trav visited sorted vertices = match vertices with
      [] → visited, sorted
    | v :: vs →
        if List.mem v visited then visited, sorted
        else
          let visited', sorted' = trav (v :: visited) sorted (succ g v) in
          trav visited' (v :: sorted') vs in
    if has_cycle g then failwith "tsort : cyclic graph"
    else
      let _, sorted = trav [] [] [s] in
      sorted;;

val tsort : (α × α) list → α → α list = <fun>

#tsort g1 1;;

- : int list = [1; 4; 2; 3; 7; 6; 5]
```

4.4 Types de données abstraits. Modules

Le types de données *graphe* défini dans la section précédente est dit *concret* : sa représentation – sous forme d'une liste ici – reste visible. Cette approche va à l'encontre d'un principe général de programmation qui vise à séparer l'*interface* d'un type de données (quelles opérations peut-on effectuer sur des données de ce type) de son *implémentation* (comment les données de ce type sont elles représentées en mémoire et comment sont réalisées les opérations qui les manipulent. La notion de *type de données abstrait* découle de ce principe. L'implémentation d'un type de données abstrait (TDA) n'est pas accessible aux programmes utilisant ce TDA et ces programmes ne peuvent manipuler ce TDA qu'au travers des opérations listées dans son interface (notion d'*encapsulation*). L'utilisation des TDA dans les programmes favorise notamment la modularité, dans la mesure où ceux-ci peuvent être vus comme des « boîtes noires » : deux implémentations satisfaisont la même interface sont interchangeables.

Le système de *modules* de Caml permet d'implanter aisément et efficacement cette notion de TDA. On va illustrer ici ce système à travers sa manifestation la plus élémentaire, fondée sur le découpage des programmes en fichiers compilés séparément[2].

La définition d'un TDA commence par l'écriture d'un fichier dit d'interface. Ce fichier (suffixé `.mli`) va donner la *signature* de ce type, c.-à-d. déclarer ce type et donner le type de toutes les fonctions le manipulant. Voici par exemple le contenu d'un fichier `graph.mli`, correspondant à l'interface d'un TDA graphe tel qu'étudié dans cette section (les `(* *)` délimitent les commentaires) :

[2]Cette vision des choses est essentiellement celle offerte par `Caml Light`. Avec `Objective Caml`, la notion de module est en fait beaucoup plus riche que celle présentée ici – grâce au concept de *foncteur* notamment –, mais sa présentation sort du cadre de ce cours.

Fichier 1 `graph.mli`

```
type 'a t
  (* The type of graphs with vertex indexed with values of type 'a *)

val make : ('a*'a) list -> 'a t
  (* Makes an empty graph from a list of edges *)

val is_edge : 'a t -> 'a * 'a -> bool
  (* [is_edge g (v1,v2)] checks whether graph [g] contains an edge from vertex [v1]
     to vertex [v2] *)

val is_vertex : 'a t -> 'a -> bool
  (* [is_vertex g v] checks whether graph [g] contains vertex [v] *)

exception Edge_exists
exception Edge_not_found

val add_edge : 'a t -> 'a * 'a -> 'a t
  (* [add_edge g (v1,v2)] adds an edge connecting vertices [v1] and [v2] to the graph [g]
     and returns the resulting graph.
     [add_edge] raises Edge_exists if the edge already exists. *)

val rem_edge : 'a t -> 'a * 'a  -> 'a t
  (* [rem_edge g (v1,v2)] removes the edge connecting vertices [v1] and [v2] from the graph [g]
     and returns the resulting graph.
     [rem_edge] raises Edge_exists if the edge does not exist. *)

val succ : 'a t -> 'a -> 'a list
  (* [succ g v] returns the list of all vertices of [g] which are directly reachable from [v] *)

val pred : 'a t -> 'a -> 'a list
  (* [pred g v] returns the list of all vertices of [g] from which [v] is directly reachable *)

val vertices : 'a t -> 'a list
  (* [vertices g] returns the list of all vertices from graph [g] *)

val inits : 'a t -> 'a list
  (* [inits g] returns the list of all vertices with no predecessors *)

val terms : 'a t -> 'a list
  (* [inits g] returns the list of all vertices with no successors *)
```

On remarquera la déclaration du type abstrait $\alpha\ t$, qui correspond aux graphes dont les sommets sont indexés par des valeurs de type α. Les fonctions opérant sur ces graphes prennent et renvoient des valeurs de type $\alpha\ t$ sans faire mention de leur représentation concrète. Ceci explique notamment la présence de la fonction *make*, servant à « fabriquer » une valeur (abstraite) de type graphe à partir d'une liste (concrète) de listes d'arcs.

Ce fichier est compilé avec la commande

```
ocamlc -c graph.mli
```

et produit un fichier `graph.cmi`.

La deuxième étape consiste à écrire l'*implémentation* du type *Graph*. Cette implémentation est traditionnellement donnée dans un fichier `graph.ml`. Voici par exemple un fichier `graph.ml` reprenant la représentation par liste d'arcs décrite à la section précédente :

Fichier 2 `graph.ml`

```
type 'a t = ('a * 'a) list

let is_edge g (v1,v2) = List.mem (v1,v2) g

let is_vertex g v = List.exists (function (v1,v2) -> v=v1 or v=v2) g

exception Edge_exists
exception Edge_not_found

let add_edge g (v1,v2) =
  if is_edge g (v1,v2) then raise Edge_exists
  else (v1,v2)::g

let make es = List.fold_left add_edge [] es

let rec rem_edge g (v1,v2) = match g with
  [] -> raise Edge_not_found
| (v1',v2')::rest -> if v1=v1' && v2=v2' then rest else rem_edge rest (v1,v2)

let rec succ g v = match g with
  [] -> []
| (v1,v2)::rest -> if v1=v then v2 :: succ rest v else succ rest v

let rec pred g v = match g with
  [] -> []
| (v1,v2)::rest -> if v2=v then v1 :: pred rest v else pred rest v

let vertices g =
  let update vs (v1,v2) =
    let vs' = if List.mem v1 vs then vs else v1::vs in
    if List.mem v2 vs' then vs' else v2::vs' in
  List.fold_left update [] g

let inits g = List.filter (function v -> pred g v = []) (vertices g)

let terms g = List.filter (function v -> succ g v = []) (vertices g)
```

La compilation de ce fichier s'effectue avec la commande

```
ocamlc -c graph.ml
```

et produit un fichier `graph.cmo`. A ce niveau, le compilateur vérifie que l'implémentation `graph.ml` est bien conforme à l'interface `graph.cmi`, *i.e.* que chaque type, exception ou fonction déclarée dans l'interface est bien défini dans l'implémentation et que les types *inférés* sont compatibles avec les types *déclarés*.

Une fois compilé, le module *Graph* peut être utilisé comme n'importe quel autre type prédéfini du langage. On peut ainsi l'utiliser depuis l'interpréteur, en le chargeant explicitement :
#*load* `"graph.cmo";;`
#let *g2* = *Graph.make* [(`"A","B"`); (`"B","C"`); (`"C","D"`)];;
val *g2* : *string Graph.t* = < *abstr* >
#*Graph.succ* `"B"` *g2*;
- : *string list* = [`"C"`]
Le type *Graph.t* est maintenant abstrait (*abstr*) : sa représentation est cachée et la seule manière d'accéder à un graphe est d'utiliser les fonctions listées dans `graph.cmi`.

On peut aussi écrire des programmes *stand-alone* utilisant le type *Graph* directement dans des fichiers compilables. Voici par exemple un petit programme interactif qui lit la description d'un graphe, sous la forme d'une liste d'arcs dans un fichiera – dont le nom est passé sur la ligne de commande – et affiche ensuite, pour chaque identificateur de sommet entré au clavier, le résultat du parcours en profondeur d'abord de ce graphe[3] :

Fichier 3 `example.ml`

```
open Scanf

let read_graph ch =
  let rec loop acc =
    try
      let n1,n2 = fscanf ch "%d %d" (fun n1 n2 -> (n1,n2)) in
      loop ((n1,n2)::acc)
    with End_of_file ->
      acc
  in Graph.make (loop []);;

let user_interact f g =
  let prompt msg = print_string msg; read_int () in
  let print_vertex n = print_int n; print_string " " in
  while true do
    let n = prompt "Enter vertex id: " in
    if Graph.is_vertex g n then
      let r = f g n in
      begin print_string "> "; List.iter print_vertex r; print_newline () end
    else
      begin print_string "Unknown vertex : "; print_int n; print_newline() end
  done;;

let main () =
  if Array.length Sys.argv < 2 then begin prerr_endline "usage: example file"; exit 1 end
  else
    let filename = Sys.argv.(1) in
    let g = read_graph (open_in filename) in
    user_interact Graph_algos.dft g;;

main();;
```

[3]Ce programme fait appel à certaines fonctionnalités du langage Caml non décrites dans ce document. Son seul but est d'illustrer ici le fonctionnement du compilateur et la notion de programme *stand-alone*.

Ce programme se compile avec les commandes

```
ocamlc -c example.ml
ocamlc -o example graph.cmo graph_algos.cmo example.cmo
```

et se lance de la manière suivante

```
./example g1.dat
```

où `g1.dat` est le fichier contenant la description du graphe (un couple d'entiers, séparé par un espace par ligne) et `graph_algos.cmo` le module contenant le code les fonctions *dft*, *bft*, ... vues à la section 4.3.

Par la suite, on peut envisager de changer d'implantation pour le module *Graph* : remplacer la représentation par liste d'arcs par une représentation par matrice d'adjacence par ex.. Il suffit alors de réécrire le fichier `graph.ml`. Tant que cette implémentation est conforme à la signature à l'interface donnée par `graph.mli`, ce changement est transparent pour tous les programmes utilisant le module *Graph* (les fichiers `graph_algos.ml` et `example.ml` n'ont pas à être recompilé en particulier).

Bibliographie

[1] X. Leroy. The Objective Caml system documentation ana user's manual. Disponible en ligne à `http ://caml.inria.fr/ocaml/htmlman/index.html`

[2] E. Chailloux, P. Manoury, B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, Paris, Avril 2000. Disponible en ligne à `http ://www.pps.jussieu.fr/Livres/ora/DA-OCAML/index.htmlx`

[3] J. Hickey. *An introduction to OCaml programming*. Available on-line at `http ://www.cs.caltech.edu/cs134/cs134b/book.pdf`

[4] D. Remy. *Using, Understanding, and Unraveling The OCaml Language*. Notes de cours. Disponible en ligne à `http ://pauillac.inria.fr/ remy/isia/`

[5] P. Weis. *Le langage Caml*. Transparents d'une formation à Caml Light destinée au professeurs de Math Spé. Disponible en ligne à `http ://caml.inria.fr/tutorials/ups.ps.gz`

[6] M. Mauny. *Functional Programming using Caml Light*. Available on-line at `http ://caml.inria.fr/tutorial/index.html`

[7] P. Weis, X. Leroy. *Le langage Caml*, Dunod, Paris, 1999

[8] X. Leroy, P. Weis. *Manuel de Référence du langage Caml*, InterEditions, Paris 1993

[9] T. Accart Hardin, V. Donzeau-Gouge Viguié. *Concepts et outils de programmation – le style fonctionnel, le style impératif avec CAML et Ada*, InterEditions, Paris 1991

[10] G. Cousineau, M. Mauny. *Approche fonctionnelle de la programmation*, Ediscience (Collection Informatique), Paris 1995

[11] J. Rouablé. *Programmation en Caml – Cours et atelier*, Eyrolles, Paris 1997

Table des matières

1 Rudiments **2**
- 1.1 Expressions, valeurs et types . . . 2
  - 1.1.1 Entiers et flottants . . . 2
  - 1.1.2 Chaines de caractères . . . 2
  - 1.1.3 Booléens . . . 3
  - 1.1.4 N-uplets . . . 3
- 1.2 Définitions . . . 3
  - 1.2.1 Définitions locales . . . 4
- 1.3 Fonctions . . . 5
  - 1.3.1 Fonctions à plusieurs arguments . . . 6
  - 1.3.2 Fonctions à plusieurs résultats . . . 6
- 1.4 Un peu plus sur les fonctions . . . 7
  - 1.4.1 Fonctions récursives . . . 7
  - 1.4.2 Définition de fonction par filtrage . . . 8
- 1.5 Définition de types . . . 9
  - 1.5.1 Enregistrements . . . 9
  - 1.5.2 Types sommes . . . 10

2 Listes **12**
- 2.1 Quelques fonctions élémentaires sur les listes . . . 12
- 2.2 Fonctionnelles . . . 15
  - 2.2.1 Deux autres fonctionnelles utiles : map et filter . . . 17
- 2.3 Un mot sur le système de types de Caml . . . 17
- 2.4 Listes d'association . . . 19

3 Arbres **21**
- 3.1 Arbres binaires . . . 22
  - 3.1.1 Quelques opérations sur les arbres binaires . . . 24
- 3.2 Arbres binaires de recherche . . . 26

4 Graphes **30**
- 4.1 Définitions . . . 30
- 4.2 Représentation . . . 30
- 4.3 Quelques algorithmes . . . 33
  - 4.3.1 Algorithmes de parcours . . . 33
  - 4.3.2 Recherche de cycle . . . 34
  - 4.3.3 Tri topologique . . . 35
- 4.4 Types de données abstraits. Modules . . . 35