Initiation à la programmation en `OCaml` et à l'algorithmique des graphes

Vincent Gripon

8 novembre 2010

Table des matières

1 Introduction

1.1 Introduction

1.1.1 Le langage

`OCaml` se classe dans les langages dits fonctionnels. Pour autant, il peut être utilisé comme un langage impératif. Les langages fonctionnels sont des langages qui privilégient une programmation par applications de fonctions, ce qui leur donne une proximité avec les mathématiques très intéressante. Par opposition, les langages impératifs se veulent proche de l'architecture de von Neumann. Ils exécutent un set d'instructions modifiant l'état des variables du système.

Ces deux paradigmes ne sont en réalité pas incompatibles et beaucoup de langages permettent d'utiliser les deux. Il faut simplement retenir que certains domaines sont plus adaptés à l'un qu'à l'autre.

`OCaml` vient de `O`, `Cam` et `ML`. Le premier spécifie qu'il s'agit d'un langage orienté objet. Cette partie est rarement utile sur de petits projets, et il y a peu de chances que nous en ayons besoin durant la formation. Pour ma part, je n'ai commencé à utiliser les objets dans `OCaml` que très récemment sans que c'eût été un choix nécessaire. Les deux autres parties vont bien plus nous intéresser. `Cam` vient de *Categorical Abstract Machine*. En effet, `OCaml` utilise des résultats de la théorie des catégories dans sa syntaxe. On y retrouve en particulier les projections. Enfin, la dernière partie vient de `ML` qui signifie *Meta-Langage*. Les *Meta-langages* offrent plusieurs choses :

- un *garbage collector* automatique,
- un typage fort,
- un système d'inférence de type,
- un filtrage par motif (*pattern matching*),
- une gestion des exceptions...

Par rapport aux autres langages, `OCaml` se place très bien tant en termes de performances qu'en termes de longueur des codes sources :

http ://blog.gmarceau.qc.ca/2009/05/speed-size-and-dependability-of.html

1.1.2 Avant de commencer

Let types de base dans `OCaml` sont les suivants :

- `int`, pour les entiers,
- `float`, pour les nombres à virgule flottante,
- `char`, pour les caractères,
- `string`, pour les chaînes de caractères,
- `unit`, le type inexistant.

Et bien sûr toute combinaison de ces types à l'aide de tableaux, listes, mélange des deux etc

Les commentaires sont écrits entre `(*` et `*)`. Pour la génération automatique de documentation, on utilisera parfois `(**` et `*)`.

Les outils que j'utilise pour programmer en `OCaml` se limitent à trois fenêtres. Un navigateur internet sur le manuel en ligne de `OCaml`, un émulateur de termi-

nal pour la compilation, et `emacs` pour l'édition des sources. Notez que `emacs` nécessite un mode externe pour la syntaxe `OCaml` qui s'appelle `tuareg`.

Pour commencer à utiliser `emacs` avec l'environnement dédié à `OCaml`, il suffit de créer un nouveau fichier avec l'extension `.ml`.

1.1.3 Mes premiers `Hello World!`

`OCaml` a une autre grande force qui est qu'il peut être utilisé à trois niveaux. Le langage peut être interprété dans un `toplevel`, c'est en particulier ce que fait `tuareg`. C'est le mode le moins rapide mais le plus pratique pour le développement. Le fichier source peut ensuite être compilé soit en bytecode `OCaml` soit en natif. Le bytecode `OCaml` est similaire à ce que l'on peut trouver en Java par exemple, il est interprété par la machine `ocamlrun`. Le gros avantage du bytecode est sa portabilité, mais ses performances peuvent être bien inférieures à celles du code natif. Pour générer du bytecode `OCaml`, il faudra passer par une ligne de commande : `ocamlc monfichier.ml -o nomprogramme`. La compilation native produit quand à elle du code binaire spécifique à la machine où il est produit, c'est le programme le plus rapide et celui qui devrait être utilisé pour l'algorithmique. Pour l'obtenir, il faut rentrer la commande : `ocamlopt monfichier.ml -o nomprogramme`.

Faisons une première comparaison :

```
for i = 1 to 100000000 do
  ()
done;
print_string "Hello World!\n";
print_float (Sys.time());;
```

Enregistrez ce code dans un fichier `.ml` et comparez les vitesses d'exécution pour les différentes méthodes.

2 Le langage

2.1 Syntaxe de base

2.1.1 Déclarations

En `OCaml`, les déclarations sont faites en utilisant le mot clé `let`. Contrairement aux langages non `ML`, le type des variables n'est pas précisé.

```
let zero = 0;;
val zero : int = 0
```

L'usage du `;;` signale la fin d'un bloc d'instructions. Un simple `;` signale quant à lui la fin d'une instruction. Si par exemple on demande à tuareg d'évaluer un bloc d'instructions par le biais du raccourci `ctrl-c e`, les instructions seront évaluées jusqu'à rencontrer le prochain `;;`.

```
print_string "instruction 1\n";
print_string "instruction 2 et fin de bloc\n";;
print_string "instruction 3";;
  instruction 1
instruction 2 et fin de bloc
- : unit = ()
```

2.1.2 Références

L'affectation n'est pas réalisée par l'usage du `=`.

```
1=2;;
- : bool = false
```

Pour pouvoir changer la valeur d'une variable, il faut la définir comme référence. Ceci se fait par l'intermédiaire du mot clé `ref`. L'affectation est alors permise avec le symbole `:=`. Afin d'accéder à la valeur d'une référence, on utilise le symbole `!` ajouté avant le nom de la référence.

```
let zero = ref 0;;
zero := !zero + 1;
!zero;;
 - : int = 1
```

Notez qu'il n'est pas possible d'enlever le premier `;;`. Ceci vient du fait qu'il y a deux types de variables en `OCaml` : locales et globales. L'usage de `;;` définit une variable globale. Par opposition, il faudra utiliser le mot clé `in` qui précisera la localité d'une variable.

```
let zerolocal = ref 0 in
zerolocal := !zerolocal + 1;
!zerolocal;;
 - : int = 1
```

Si à la suite de cet exemple je redemande la valeur de la variable `zerolocal`, le programme me dira qu'il n'existe pas de telle variable.

```
!zerolocal;;
Error: Unbound value zerolocal
```

2.1.3 Fonctions

En `OCaml`, les fonctions sont des éléments de premier ordre, au même titre que les variables. Dans cette idée, la déclaration d'une fonction est très proche de celle d'une variable.

```
let foisdeux a =
  a * 2;;
val foisdeux : int -> int = <fun>
```

On notera déjà la façon originale de noter la signature de la fonction.

L'appel de cette fonction se réalisera alors aussi simplement que :

```
foisdeux 12;;
- : int = 24
```

Puisque les fonctions sont des objets de premier ordre, il est également possible de les définir localement à l'aide du mot clé `in`.

```
let foissix a =
  let foistrois a =
    a * 3
  in
  foisdeux (foistrois a);;
val foissix : int -> int = <fun>
```

De cette manière, je peux demander à l'interprète des informations sur `foissix` ou `foistrois`.

```
foissix;;
- : int -> int = <fun>
```

```
foistrois;;
Error: Unbound value foistrois
```

2.1.4 Fonctions récursives

Lorsque le langage interprète une fonction, il ne crée que les liens locaux dont il a besoin. En particulier, il ne lui est pas nécessaire de connaître un lien vers la fonction elle même dans le cas général... sauf si celle ci est récursive.

```
let estpair a =
  a=0 || not(estpair (a-1))
Error: Unbound value estpair
```

Le mot clé à utiliser est alors `rec`.

```
let rec estpair a =
  a=0 || not(estpair (a-1));;
val estpair : int -> bool = <fun>
```

Il est parfois utile d'avoir des fonctions doublement récursives. Il convient alors de définir les deux fonctions dans un même environnement à l'aide du mot clé `and`.

```
let rec estpair a =
  a=0 || estimpair (a-1)
and estimpair a =
  a=1 || not(estpair a);;
val estpair : int -> bool = <fun>
val estimpair : int -> bool = <fun>
```

2.1.5 Conditionnelles et boucles

Les condition-elles en `OCaml` s'expriment avec les mots clés `if`, `then` et `else`.

```
let estpair a =
  if a mod 2 = 0
  then true
  else false;;
val estpair : int -> bool = <fun>
```

Contrairement à beaucoup de langages impératifs, la conditionnelle de `OCaml` peut être utilisée dans une expression. Il est important de noter que le typage du langage force les différentes branches de la conditionnelle à avoir le même type.

Les boucles `while` et `for` sont également présentes.

```
for i = 0 to 3 do
  print_int i;
done;;
0123- : unit = ()
```

```
let i = ref 0 in
while !i < 4 do
  print_int !i;
  i := !i + 1;
done;;
0123- : unit = ()
```

2.1.6 Curryfication

On a déjà remarqué la signature étrange des fonctions en `OCaml`. Elle devient plus claire lorsque le nombre d'arguments augmente.

```
let multiplie a b=
  a * b;;
val multiplie : int -> int -> int = <fun>
```

La bonne façon de concevoir les signatures en `OCaml` est d'imaginer que tout est parenthésé à droite. Ainsi la signature `int -> int -> int` se lira `int -> (int -> int)`. La fonction `multiplie` est donc une fonction qui prend un entier et qui rend une fonction qui prend un entier à son tour et qui rend un entier. Il est possible de récupérer la seconde fonction très simplement de la manière qui suit :

```
let multipliepardeux = multiplie 2;;
val multipliepardeux : int -> int = <fun>

multipliepardeux 3;;
- : int = 6
```

Autrement dit, une fonction de deux variables est en réalité une succession de deux fonctions, chacune avec son argument. L'ordre a donc une importance particulière lorsque l'on souhaite utiliser cette propriété, et le programmeur habitué le choisira judicieusement.

2.1.7 Types

`OCaml` est doté d'un système d'inférence de types. Ce procédé est à double tranchant. S'il est très pratique par certains aspects, il provoque en contre partie des lourdeurs.

```
1+2.;;
Error: This expression has type float but an expression was expected of type int
```

Il n'y a aucune ambiguïté sur ce que devrait faire cette expression. Mais le système de typage du langage l'empêche de permettre des *casts* implicites. Ainsi, toutes les opérations sont fortement typées.

```
(+);;
- : int -> int -> int = <fun>
```

```
(+.);;
- : float -> float -> float = <fun>
```

En contrepartie, le langage peut inférer des types définis implicitement.

```
let somme a b =
  a +. b;;
val somme : float -> float -> float = <fun>
```

Certaines fonctions peuvent utiliser un type générique, soit parce qu'il n'a pas encore été instancié, soit par commodité du langage.

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

On remarquera que malgré tout la fonction < force ses deux premiers arguments à être du même type, appelé `'a`.

```
let affiche fonctiondaffichage element=
  fonctiondaffichage element;
  print_string "\n";;
val affiche : ('a -> 'b) -> 'a -> unit = <fun>
```

Ici l'inféreur de type a trouvé que le premier argument, appelé `fonctiondaffichage`, est une fonction qui prend en argument un élément de type `'a` et qui rend un élément de type `'b`. Elle va par la suite l'appliquer au second argument nommé `element`, qui est donc forcément de type `'a`. Et elle rendra le résultat de cette opération qui est de type `'b`.

```
let afficheentier = affiche print_int;;
val afficheentier : int -> unit = <fun>
```

2.1.8 Quelques exercices

À ce stade, il est déjà possible de pratiquer un peu le langage. Voici quelques exercices pour se mettre en jambes.

1. Programmer une fonction récursive qui calcule factorielle d'un entier n,
2. Programmer une fonction récursive qui calcule fibonacci d'un entier n,
3. Programmer une fonction qui compte le nombre d'entiers premiers entre 2 et n,
4. Programmer une fonction qui réalise la composition de deux fonctions.

2.2 Listes et tableaux

2.2.1 Tableaux

Les tableaux sont créés en utilisant le module `Array`.

```
let tableau = Array.make 10 0;;
val tableau : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

Ils peuvent être définis explicitement avec la notation `[||]`.

```
let tableau = [|0;1;3;7|];;
val tableau : int array = [|0; 1; 3; 7|]
```

L'accès à un élément se fait avec `.(indice)`. La taille est connue par la structure, et s'accède quant à elle par l'usage de `Array.length`.

```
for i = 0 to Array.length tableau - 1 do
  print_int tableau.(i);
  print_newline ();
done;;
0
1
3
7
- : unit = ()
```

Pour modifier une case du tableau, on utilisera l'opérateur `<-`.

```
for i = 0 to Array.length tableau - 1 do
  tableau.(i) <- tableau.(i) + i;
done;;
```

Des fonctions prédéfinies permettent d'itérer directement sur les éléments d'un tableau ou d'opérer des transformations sur ses éléments.

```
Array.iter (fun x -> print_int x; print_newline()) tableau;;
0
2
5
10
- : unit = ()

Array.map (fun x -> x+1) tableau;;
 : int array = [|1; 3; 6; 11|]
```

2.2.2 Listes

Les listes sont gérées de manière plutôt élégante par le langage. Une liste peut s'écrire sous forme explicite de la façon suivante :

```
let liste = [1;2;3;5];;
val liste : int list = [1; 2; 3; 5]
```

Une liste ne peut contenir que des éléments du même type. Il est possible de rajouter un élément en tête de liste avec `::`.

```
let autreliste = 0::liste;;
val autreliste : int list = [0; 1; 2; 3; 5]
```

Réciproquement, des méthodes `List.hd` et `List.tl` permettent de récupérer le premier élément et la suite de la liste. On préférera néanmoins utiliser le *pattern matching* introduit plus tard pour manipuler les listes.

```
List.tl autreliste;;
- : int list = [1; 2; 3; 5]
```

La liste vide est dénotée `[]`. Demander le premier élément ou la suite de la liste vide renverra une exception.

Il est possible de décomposer une liste en utilisant le *pattern matching*, qui est une des plus grandes forces du langage.

```
let rec ajoute1 liste =
  match liste with
    |[] -> []
    |head::tale -> (head+1)::(ajoute1 tale);;
val ajoute1 : int list -> int list = <fun>
```

Cette fonction est particulièrement intéressante. Elle illustre à la fois la possibilité d'utiliser le *pattern matching* pour décomposer un objet et la souplesse du langage pour le reconstruire à la volée.

```
let rec derivepolynome polynome =
  match polynome with
    |[a1]|[]->[]
    |an::restepoly->(n*.an)::(derivepolynome restepoly);;
```

Encore une fois on notera la nécessité de compatibilité sur les types des différents *holds* du *pattern matching*.

2.2.3 Exercices

1. Programmer une fonction qui calcule la somme des valeurs d'un tableau d'entiers,
2. Programmer une fonction qui calcule fibonacci d'un entier n avec une complexité linéaire,
3. Programmer une fonction qui teste si un élément est dans une liste,
4. Programmer une fonction qui inverse une liste (on pourra utiliser l'opérateur `@`).

2.3 Types, exceptions

2.3.1 Types

Le *pattern matching*, bien utile pour manipuler des listes, peut être utilisé sur n'importe quel type algébrique défini dans le langage.

Un type peut être une union disjointe de différents éléments.

```
type 'a arbrebinaire = ArbreVide | Noeud of 'a * 'a arbrebinaire * 'a arbrebinaire;;
```

Je peux alors créer un arbre très simplement de la façon suivante :

```
let arbreexemple = Noeud(3,Noeud(4,ArbreVide,ArbreVide),Noeud(2,ArbreVide,ArbreVide));;
val arbreexemple : int arbrebinaire = Noeud (3, Noeud (4, ArbreVide, ArbreVide),
                                                 Noeud (2, ArbreVide, ArbreVide))
```

L'inférence de type a identifié qu'il s'agissait d'un arbre d'entiers. Je peux ensuite utiliser le *pattern matching* de manière à parcourir mon arbre :

```
let rec contient element arbre =
  match arbre with
    |ArbreVide -> false
    |Noeud(valeurnoeud,branchegauche, branchedroite) ->
       if valeurnoeud = element
       then true
       else contient element branchegauche || contient element branchedroite;;
val contient : 'a -> 'a arbrebinaire -> bool = <fun>

contient 2 arbreexemple;;
- : bool = true
```

Des types peuvent également être définis comme intersection de plusieurs caractéristiques.

```
type infos = {nom: string; prenom: string; adresse: string; age: int};;
```

On pourra alors consulter un champ d'un objet de ce type grâce à l'opérateur `..`

```
let majeur personne =
  personne.age >= 18;;
val majeur : donnee -> bool = <fun>

majeur {nom="toto";prenom="tata";adresse="";age=37};;
- : bool = true
```

De la même manière que pour les références et leurs valeurs, il faudra préciser les champs qui peuvent être modifiés.

```
let anniversaire personne =
  personne.age <- personne.age + 1;;
Error: The record field label age is not mutable
```

On précisera cette propriété avec le mot clé `mutable`.

```
type infos = {nom: string; prenom: string; adresse: string; mutable age: int};;
let anniversaire personne =
  personne.age <- personne.age + 1;;
val anniversaire : infos -> unit = <fun>
```

2.3.2 Exceptions

En tant que Méta-Langage, `OCaml` gère les exceptions. Les exceptions permettent d'interrompre le cours normal de l'exécution d'un bloc d'instruction éventuellement en passant des valeurs et de le récupérer (*handle*) plus loin.

```
let tableaucontient element tableau =
  Array.iter (fun x ->
    if x = element
    then failwith "oui"
    )
    tableau;;
val tableaucontient : 'a -> 'a array -> unit = <fun>

tableaucontient 2 [|3;4;2|];;
Exception: Failure "oui".
```

Pour récupérer cette exception, on utilise un bloc `try/with`.

```
let tableaucontient element tableau =
  try
    Array.iter (fun x ->
      if x = element
      then failwith "oui"
```

```
      )
      tableau;
    false
  with
    |Failure "oui" -> true;;
val tableaucontient : 'a -> 'a array -> bool = <fun>
```

Encore une fois, on prendra garde à la compatibilité des types. Afin de passer en plus des valeurs, il convient de définir un type spécial.

```
exception Trouve of int;;

let plusgrandquetrois tableau =
  try
    Array.iter (fun x ->
      if x > 3
      then raise (Trouve x)
      )
      tableau;
    failwith "non"
  with
    |Trouve x -> x;;
val plusgrandquetrois : int array -> int = <fun>

plusgrandquetrois [|1;2;1;2;1;5;2;1;6|];;
- : int = 5
```

2.3.3 Exercices

1. Créer un type arbre pour des arbres non binaires (on utilisera des listes),
2. Programmer une fonction qui cherche le plus petit élément dans un tel arbre,
3. Programmer une fonction qui vérifie si les éléments d'un arbre satisfont une propriété donnée comme argument à cette fonction. Cet argument sera de type `'a -> bool` et dans le cas où la propriété n'est pas vérifiée, la fonction renverra une exception,
4. Programmer une fonction qui parcoure un arbre d'entiers. Cette fonction renverra l'arbre vide dès qu'elle trouvera un entier négatif, ou le sous-arbre si l'entier est nul, et enfin l'arbre entier si aucun de ces événements ne se produit. Le tout sera traité à l'aide d'exceptions.

2.4 Modules

Les modules correspondent à des fichiers `ml` en `OCaml`. Ils implémentent un certain nombre de fonctions dont certaines (toutes par défaut) sont accessibles depuis un autre module. Lorsque l'utilisateur veut utiliser un module, il peut choisir soit d'en utiliser une fonction spécifique par le biais de

`NomDuModule.fonction` soit d'importer le module entier par le biais de la commande `open NomDuModule`.

Sauf rares exceptions (`Sys`, `Array`, `Printf`...), il est nécessaire de préciser les modules que l'on souhaite utiliser. Dans le cas du `toplevel`, cela revient à rajouter une ligne spéciale `#load "nomdumodule.cma"` dans l'interprète. Dans le cas de la compilation en bytecode, il faudra rajouter un fichier lors de la compilation `ocamlc nomdumodule.cma monfichier.ml`. Enfin, dans le cas de la compilation native, il faudra utiliser un autre fichier : `ocamlopt nomdumodule.cmxa monfichier.ml`.

Pour préciser les fonctions accessibles depuis l'extérieur, il faudra créer ou modifier un fichier `.mli` du même nom que le fichier `.ml`. Un tel fichier est généré automatiquement lors de la première compilation et peut être modifié à la main.

3 Application à la théorie des graphes

3.1 Modélisation et parcours de graphes

3.1.1 Modélisation

On va se restreindre dans cette étude à des graphes non valués. Pour autant, on gardera un esprit le plus générique possible de façon à pouvoir facilement adapter les algorithmes à des graphes plus riches.

On se propose d'utiliser deux modélisations différentes pour les graphes. La première est la plus simple : un graphe est représenté par une matrice carrée M donc les coefficients sont définis de la façon suivante : M_{ij} est le poids de l'arrête reliant le noeud i au noeud j (dans notre cas un `bool` suffirait). Ainsi, un graphe non orienté correspondra à une matrice symétrique. La seconde représentation est plus adaptée aux matrices creuses : Le graphe est représenté par un tableau de listes. La case i du tableau contient la liste de tous les noeuds qu'il est possible d'atteindre depuis le noeud i (dans le cas général, on pourrait prendre des couples formés des noeuds successeurs et des poids requis pour les atteindre).

- Programmer une fonction qui teste si un graphe sous forme matricielle est symétrique ou non,
- Programmer une fonction qui converti un graphe sous la première forme en graphe sous la seconde,
- Programmer une fonction qui fait le calcul réciproque.

3.1.2 Parcours de graphes

Un parcours de graphe revient à explorer tous les chemins élémentaires qui partent d'un noeud donné. Il existe deux familles de parcours de graphes largement utilisées : le parcours en largeur d'abord et le parcours en profondeur d'abord. Le parcours en profondeur privilégie toujours le fait d'aller plus loin dans un chemin que l'on explore avant d'aller en visiter d'autres. Au contraire, le parcours en largeur n'ira pas explorer un chemin plus loin que la i-ème étape

si un autre chemin n'a pas été exploré aussi loin. Il est possible d'unifier ces deux algorithmes sous la forme d'un seul en utilisant les structures de données `LIFO` et `FIFO`. L'algorithme générique devient alors le suivant : au départ je place le couple (noeud d'origine,liste qui ne contient que le noeud d'origine) dans ma structure de donnée. À chaque itération, je retire un élément (noeud, liste) de ma structure de donnée, je regarde l'ensemble des noeuds accessibles depuis le noeud que je viens de retirer qui ne sont pas dans la liste et je les ajoute tous dans ma structure accompagnés de la même liste agrémentée du noeud en question. Je m'arrête quand ma liste est vide.

- Programmer des fonctions `push`, `pop`, et `estvide` pour les deux structures de données,
- Les placer dans deux modules distincts,
- Programmer la version générique de l'algorithme de parcours de graphe à partir d'un noeud,
- Le tester sur des exemples à la main

3.2 Graphes aléatoires

Avant de continuer à manipuler des graphes, on va se donner les moyens d'en générer aléatoirement. Pour se faire, on utilisera le module `Random`.

Avant toute chose, il convient de l'initialiser :

```
Random.self_init ();
```

On pourra par la suite récupérer des valeurs générées aléatoirement, par exemple des `float` entre 0 et 1 avec :

```
Random.float (1.);;
```

Programmer une fonction qui prend comme argument une taille de graphe et une densité ciblée et qui génère un graphe aléatoire non valué avec ces paramètres (on ne cherchera pas spécialement à réduire l'écart-type à la densité). On fera une fonction pour générer des graphes sous forme matricielle et une autre pour les graphes sous forme de tableaux de listes.

3.3 Roy-Warshall et Dijkstra

3.3.1 Roy-Warshall

L'algorithme de Roy-Warshall permet de calculer l'ensemble des longueurs des plus courts chemins d'un noeud à l'autre. Il se présente sous la forme suivante pour une graphe de taille n :

```
matrice resultat de taille n par n = copie de matrice du graphe;
pour k noeud de 1 a n
  pour i noeud de 1 a n
    pour j noeud de 1 a n
```

```
      si resultat[i,j] > resultat[i,k] + resultat[k,j]
      alors resultat[i,j] = resultat[i,k] + resultat[k,j]
    fin pour
  fin pour
fin pour
```

Programmer l'algorithme de Roy-Warshall. Le tester sur des matrices de grande taille (graphe de taille 1000) et conclure.

3.3.2 Dijkstra

L'algorithme de Dijkstra calcule le plus court chemin d'un noeud à tous les autres dans un graphe. On peut le voir de la manière suivante : au noeud de départ on laisse couler de l'eau. On va visiter les noeuds du graphe dans l'ordre où ils reçoivent de l'eau. On peut noter que dans le cas d'un graphe non valué, l'algorithme de Dijkstra est équivalent à un parcours en largeur d'abord. Plus formellement, l'algorithme de Dijkstra repose sur une structure de donnée. Cette structure retient en permanence l'ensemble des noeuds non visités et leurs distances telles que connues à un certain instant au noeud d'orgine. Deux opérations sont réalisées sur cette structure : récupérer le noeud le plus proche (ou prévenir si elle est vide) et mettre à jour les distances de certains noeuds.

L'algorithme de Dijkstra itère ensuite sur cette structure, qui contient initialement seulement le noeud d'origine à une distance 0. À chaque itération, l'algorithme récupère le noeud n_0 le plus proche dans la structure. Il regarde ensuite tous les noeuds n_i accessibles depuis n_0, et si ces noeuds n'ont pas déjà été visités il compare la distance qu'il connaît déjà pour s'y rendre (s'il y en a une) à la distance qu'il faudrait en passant par n_0 pour se rendre à n_i (en utilisant l'arrête qui relie n_0 à n_i). Si la dernière est plus petite, il met à jour cette distance dans la structure. L'algorithme s'arrête quand il n'y a plus de noeuds à visiter à distance finie.

Programmer l'algorithme de Dijkstra. Regarder jusqu'à quels paramètres sur le graphe il reste raisonnable.

3.4 Recherche des cycles de longueur minimale dans un graphe

Afin de rechercher le cycle le plus court dans un graphe non valué, on peut utiliser le parcours en largeur. L'idée est alors assez simple : On réalise en même temps un parcours en largeur depuis tous les noeuds du graphes, et le premier d'entre eux qui revient sur le noeud de son départ définira le cycle le plus court.

En modifiant le parcours générique programmé plus tôt, proposer un algorithme qui calcule la longueur du cycle le plus court dans un graphe.

4 Interface graphique, illustration avec les flocons de von Koch

4.1 Module graphique

`OCaml` implémente quelques opérations de base pour interagir graphiquement avec l'utilisateur.

Le module à utiliser est `Graphics`.

Lors de l'utilisation d'un module, il est parfois plus lisible de l'ouvrir en entier. De fait, les fonctions qu'il contient pourront être utilisées sans préciser le nom du module.

```
open Graphics;;
```

Pour ouvrir la fenêtre graphique, l'utilisateur pourra utiliser la commande `open_graph`.

```
open_graph " 800x600";;
```

Il existe de nombreuses primitives permettant de tracer des figures et de changer les couleurs. Le plus simple est de consulter l'aide en ligne.

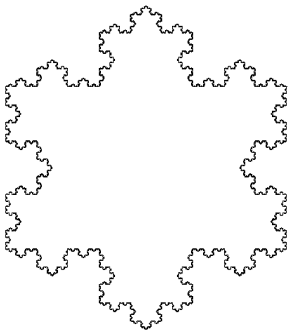
4.2 Flocons de von Koch

Les flocons de von Koch sont des fractales qui ont certaines propriétés amusantes : la courbe est fermée, bornée et continue mais pourtant de longueur infinie.

La fractale de von Koch est définie comme étant la limite du procédé itératif qui suit :

1. Au départ se trouve un ou plusieurs vecteurs,
2. Chaque itération effectue une transformation sur les vecteurs selon le schéma qui suit :

Après 6 itérations, et en partant d'un triangle équilatéral, le rendu d'un flocon est le suivant :



Programmer une fonction qui affiche un flocon de von Koch à une itération donnée.

4.3 Débogage, génération de documentation

4.3.1 Débogage

Il est rare de faire des erreurs dans le langage grâce à la rigidité imposée par le typage fort. Pour les erreurs restantes, l'utilisateur gagnera à utiliser des `assert` régulièrement. Le `assert` en `OCaml` lancera une exception spéciale en cas de résultat faux qui permettra d'identifier la ligne à problème dans le code. Prenons par exemple le fichier `testassert.ml` qui contient le code suivant :

```
(** f x renvoit l'inverse de x si x est différent de 0 *)
let f x =
  assert (x <> 0.);
  1./.x;;
f 0.;;
```

Son exécution lancera une exception de type `Assert_failure` :

```
Exception: Assert_failure ("testassert.ml", 5, 2).
```

Plus généralement, certaines erreurs classiques sont dues à des débordements de tableaux, et l'erreur par défaut de `OCaml` est peu instructive. Afin de récupérer les lignes d'erreurs, il faut conjointement compiler les programmes avec des paramètres particuliers `-g` et de lancer le bytecode produit avec `ocamlrun` et une option `-b`. Par exemple pour le fichier `depassementdetableau.ml` qui suit :

```
let f x =
  x.(1);;
f [||];;
```

```
Fatal error: exception Invalid_argument("index out of bounds")
Raised by primitive operation at unknown location
Called from file "depassementdetableau.ml", line 3, characters 0-6
```

4.3.2 Génération de documentation

Certains commentaires peuvent être utilisés automatiquement pour produire de la documentation. Pour ce faire, il faut utiliser les commentaires `(**, *)`. Il convient ensuite d'utiliser `ocamldoc` avec certaines options. Testez par exemple `ocamldoc -html nomfichier.ml`.

Pour le fichier `testassert.ml`, et en sortie `-latex`, cela donne :

Module `Testassert` : test du assert

`val f : float -> float`

f x renvoit l'inverse de x si x est différent de 0

4.3.3 Autres

`OCaml` est livré avec un profiler qui permet d'identifier les passages coûteux de certains algorithmes afin de les optimiser. Notez enfin qu'il est possible de créer des makefile très pratiques.