cours2

November 3, 2020

1 Programmation fonctionnelle

1.1 Semaine 2 : Programmer avec des listes

Etienne Lozes - Université Nice Sophia Antipolis - 2019

---

[1]:
```
let () = ()
```

```
Findlib has been successfully loaded. Additional directives:
  #require "package";;      to load a package
  #list;;                   to list the available packages
  #camlp4o;;                to load camlp4 (standard syntax)
  #camlp4r;;                to load camlp4 (revised syntax)
  #predicates "p,q,…";;   to set these predicates
  Topfind.reset();;         to force that packages will be reloaded
  #thread;;                 to enable threads
```

1.2 Les listes: des données structurées

Au premier cours nous avons beaucoup travaillé sur des données **atomiques** comme les entiers, les flottants, les booléens, ou encore les caractères. Nous avons aussi vu quelques données **structurées** comme les tuples et les chaînes de caractères.

Ce cours-ci, nous allons nous focaliser sur les **listes**. Une liste est une suite finie de valeurs, et ces valeurs sont *toutes du même type*.

1.3 Les listes par énumération

On peut créer une liste par **énumération**, en séparant chacune de ses valeurs par un `;` à l'intérieur de deux crochets.

[2]:
```
let liste_entiers = [0;1;2;3;4;5;6;7;8;9]
```

[2]:
```
val liste_entiers : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

[3]:
```
let liste_booleens = [true]  (* une liste à un seul élément! *)
```

[3]:
```
val liste_booleens : bool list = [true]
```

Si toutes les valeurs de la liste sont du type `t`, le type de la liste est `t list`.

OCaml détermine tout seul le type de la liste.

Il nous empêche aussi de créer des listes mal typées; une liste est mal typée si elle contient au moins deux valeurs de types différents.

[4]:
```
let liste_mal_typee = [0; 1; true]
```

```
File "[4]", line 1, characters 29-33:
1 | let liste_mal_typee = [0; 1; true]
                                 ^^^^
Error: This expression has type bool but an expression was expected of type
         int
```

OCaml détermine le type de la liste en regardant la première valeur qui apparait dans l'énumération.

Et pour la liste vide?

[5]:
```
let liste_vide = []
```

[5]:
```
val liste_vide : 'a list = []
```

Le type de la liste vide fait intervenir une *variable de type*, que nous avions déjà vues la semaine dernière dans le type de certaines fonctions.

On dit que c'est une liste **polymorphe**: OCaml pourra la voir comme une liste d'entiers ou comme une liste de booléens, selon les besoins de l'inférence de type.

1.4 Le module `List`

La longueur d'une liste est donnée par la fonction `List.length`

[6]:
```
List.length liste_entiers
```

[6]:
```
- : int = 10
```

On avait déjà rencontré `String.length` la semaine dernière, pour calculer la longueur d'une chaîne de caractères. Ce deux fonctions portent deux noms différents, et en effet, ce sont bien deux fonctions différentes.

D'ailleurs toutes les fonctions sur les listes sont rangées dans un **module** qui s'appelle `List`.

```
[7]: List.rev [1;2;3]  (* l'image mirroir de la liste *)
```

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

```
[8]: List.append [1;2;3] [4;5;6]   (* la concaténation de deux listes *)
```

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

```
[9]: List.hd [1;2;3] (* l'élément de tête (head) de la liste *)
```

```
[9]: - : int = 1
```

et ainsi de suite…

Notez bien que **les listes ne sont pas mutables**.

Toutes les fonctions sur les listes renvoient une **nouvelle** liste.

Elles ne peuvent pas modifier leur argument.

```
[10]: let l = [1;2;3]
```

```
[10]: val l : int list = [1; 2; 3]
```

```
[11]: let l2 = List.rev l
```

```
[11]: val l2 : int list = [3; 2; 1]
```

```
[12]: l
```

```
[12]: - : int list = [1; 2; 3]
```

1.5 À propos de la notation `List.`

Au début on peut trouver cela pénible de devoir écrire `List.` à chaque fois.

En fait je pourrais demander à OCaml de rendre implicite le `List.`. Pourtant, dans ce cours, je ne le ferai pas! ceci pour éviter, entre autre, les confusions entre les fonctions `length` de modules différents, comme `String.length`.

En TP, vous pourrez toujours créer votre propre alias, par exemple `let len = List.length` et l'utiliser ensuite…

Et puis il y a d'autres solutions, qui permettent d'écrire `length` au lieu de `List.length` *juste là où il faut*, patience, on aura un cours tout entier sur les modules!

Notez par ailleurs que vous n'aurez pas à écrire `List.append`, OCaml offre une notation alternative pour cette fonction

[13]:
```
[1;2;3] @ [4;5;6] (* souvenez vous : @ = a, comme dans *a*ppend *)
```

[13]:
```
- : int list = [1; 2; 3; 4; 5; 6]
```

1.6 La liste du débutant

[14]:
```
let ma_liste = ['h','e','l','l','o'] in List.length ma_liste
```

[14]:
```
- : int = 1
```

OCaml dit que la liste ci-dessus est de longueur 1, et il a parfaitement raison.

Voyez-vous pourquoi?

Le type de ma_liste nous renseigne sur l'erreur que nous avons commise

[15]:
```
let ma_liste = ['h','e','l','l','o']
```

[15]:
```
val ma_liste : (char * char * char * char * char) list =
  [('h', 'e', 'l', 'l', 'o')]
```

Et oui, c'est une liste de quintuplets de caractères; elle contient un seul élément, qui est un quintuplet!

Pour obtenir une liste de cinq caractères, il aurait fallu écrire

[16]:
```
let ma_liste = ['h';'e';'l';'l';'o']
```

[16]:
```
val ma_liste : char list = ['h'; 'e'; 'l'; 'l'; 'o']
```

[17]:
```
List.length ma_liste
```

[17]:
```
- : int = 5
```

Méfiez-vous! Vous n'êtes plus en Python…

1.7 Notion de liste chaînée

La différence entre les listes d'OCaml et celles de Python dépasse très largement les histoires de syntaxe.

- Les listes de Python sont ce qu'on appelle des **tableaux extensibles**. Toutes les valeurs sont rangées côte à côte en mémoire.
- Les listes d'OCaml, en revanche, sont des **listes chaînées**. Chaque valeur est dans une **cellule mémoire** qui contient aussi l'adresse de la cellule suivante. Ces cellules sont rangées n'importe où en mémoire.

Les tableaux extensibles ont les mêmes soucis que les familles nombreuses: quand il n'y a plus de place, il faut déménager tout le monde, et plus il y a de monde, plus c'est couteux.

Les listes chaînées ont des soucis plus administratifs: quand on cherche le bureau 719, il faut d'abord frapper à la porte de 718 autres bureaux.

On peut dire que les listes chaînées sont plus "bas niveau" que les tableaux extensibles: elles peuvent s'avérer plus efficaces, encore faut-il les utiliser à bon escient. Et cela demande un peu d'expertise…

1.8 Ajouter une valeur en début de liste

Pour ajouter une valeur en début de liste, on utilise l'opérateur `::`

[18]:
```
let liste1 = [1; 2; 3]
```

[18]:
```
val liste1 : int list = [1; 2; 3]
```

[19]:
```
let liste2 = 0 :: liste1
```

[19]:
```
val liste2 : int list = [0; 1; 2; 3]
```

[20]:
```
let liste3 = 9 :: 8 :: 7 :: liste1  (* = 9::(8::(7::liste1))) *)
```

[20]:
```
val liste3 : int list = [9; 8; 7; 1; 2; 3]
```

Notez bien que `liste3` ne contient pas le `0` qu'on a ajouté quand on a défini `liste2`: `liste1` n'a pas été modifiée pendant la définition de `liste2`. N'oubliez pas, **les listes ne sont pas mutables**.

Ceci permet d'économiser de la mémoire et du temps de calcul en recopies. Par exemple, `liste2` et `liste3` peuvent partager sans risque `liste1`.

En comparaison, le partage avec une structure de données mutable est toujours risqué: si l'on changeait une valeur dans liste1, on changerait par la même occasion le contenu de liste2 et liste3, et ce n'est pas toujours ce qu'on veut faire…

1.9 Parcourir une liste chaînée

Pour parcourir une liste chaînée, il faut savoir suivre les flèches. Pour cela, on va avoir besoin de la fonction `List.tl`, qui renvoie la queue (*tail*) de la liste.

[21]:
```
let l = [1;2;3]
```

[21]:
```
val l : int list = [1; 2; 3]
```

[22]:
```
let l' = List.tl l
```

[22]:
```
val l' : int list = [2; 3]
```

[23]:
```
l (* n'a pas été modifié par List.tl... *)
```

[23]:
```
- : int list = [1; 2; 3]
```

[24]:
```
List.tl []
```

```
Exception: Failure "tl".
Raised at file "stdlib.ml", line 29, characters 22-33
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

Supposons par exemple que l'on n'ait pas la fonction `List.length` et que l'on veuille la reprogrammer.

On va pouvoir faire un calcul **par récurrence**, avec **un cas de base** et un principe de récurrence permettant de se ramener à **une liste plus petite**

[25]:
```
let rec len li =
  if li = []                (* si la liste vide *)
  then 0                    (* alors sa longueur vaut 0 *)
  else 1 + len (List.tl li) (* sinon la longueur de sa queue + 1 *)
```

[25]:
```
val len : 'a list -> int = <fun>
```

Définissons de même la fonction `sum` qui calcule la somme d'une liste d'entiers.

[26]:
```
let rec sum li =
  if li = []
```

```
  then 0
  else List.hd li + sum (List.tl li)
```

[26]:
```
val sum : int list -> int = <fun>
```

[27]:
```
sum [1;2;3]
```

[27]:
```
- : int = 6
```

1.10 Les définitions par cas

Laissons les listes un instant, et supposons que nous voulions choisir au hasard l'une des chaînes de caractères `"pierre"`, `"feuille"`, ou `"ciseaux"` avec probabilité 1/3 chacune. On commence naturellement par tirer un nombre au hasard entre 0 et 2

[28]:
```
let n = Random.int 3
```

[28]:
```
val n : int = 0
```

Il reste ensuite à examiner la valeur de `n` pour renvoyer la chaîne de caractères qu'il code.

On peut utiliser une **définition par cas** de la forme `match ... with ...`.

[29]:
```
let tirage = match n with
| 0 -> "pierre"
| 1 -> "feuille"
| 2 -> "ciseaux"
| _ -> "erreur"
```

[29]:
```
val tirage : string = "pierre"
```

Chaque `|` démarre un nouveau cas (le premier `|`, qui suit le `with`, est optionnel). Le dernier cas, avec `_`, est toujours vérifié: il faut le mettre en dernier, car OCaml examine les cas dans l'ordre.

En fait on utilise en général les définitions par cas pour des fonctions qui portent sur des données structurées, car on peut en même temps **déstructurer** la donnée.

La fonction suivante renvoie `true` si le point `p` qu'on lui passe en paramètre est situé sur l'axe des abscisses ou sur celui des ordonnées

[30]:
```
let est_sur_un_axe p = match p with
  (0., y) -> true
| (x, 0.) -> true
| _       -> false
```

[30]:
```
val est_sur_un_axe : float * float -> bool = <fun>
```

Chaque cas est identifié par un **motif** auquel on associe un **traitement**. Le motif peut contenir des constantes explicites (`0`,`0.`,`"zero"`, etc) ou des **variables de motif**, comme `x` et `y` ci-dessus. Ces variables de motif sont locales: elles n'"existent" que durant le traitement associé au motif.

Avec des listes, on peut écrire des motifs qui utilisent `[...]` et `::`.

[31]:
```
let rec sum li = match li with
| [] -> 0 (* le "neutre" de +  *)
| tete::queue ->  tete + sum queue
```

[31]:
```
val sum : int list -> int = <fun>
```

[32]:
```
let rec maximum li = match li with
| [] -> -. infinity (* le "neutre" de max *)
| [x] -> x (* cas de base, on pourrait l'omettre car   max x (-. infinity) = x␣
 ↪*)
| head::tl -> max head (maximum tl) (* <- liste de longueur >=2 *)
```

[32]:
```
val maximum : float list -> float = <fun>
```

Notez que le cas `_` n'est pas obligatoire! Il aurait été inutile au-dessus, car toute liste est soit de la forme `[]`, soit de la forme `head::tail`.

Dans une définition par cas, l'essentiel est de bien couvrir tous les cas. L'une des grandes forces d'OCaml est qu'il vous aide à ne pas en oublier. Par exemple, si on avait oublié le cas de base dans la définition de `maximum` précédente, on aurait eu droit à un avertissement.

[33]:
```
let rec maximum li = match li with
| [x] -> x
| head::tl -> max head (maximum tl)
```

```
File "[33]", line 1, characters 21-81:
1 | …match li with
2 | | [x] -> x
3 | | head::tl -> max head (maximum tl)
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
```

[33]:
```
val maximum : 'a list -> 'a = <fun>
```

1.11 Listes et complexité

Nous allons maintenant re-programmer les fonctions les plus usuelles sur les listes. Il faudra bien comprendre à la fois leur **fonctionnement** et leur **complexité** en temps pour savoir les utiliser à bon escient!

Fonction	Complexité	Description
`List.append l1 l2`	$\mathcal{O}(n_1)$	concaténation
`List.length l`	$\mathcal{O}(n)$	longueur
`List.mem x l`	$\mathcal{O}(n)$	présence (*membership*)
`List.rev l`	$\mathcal{O}(n)$	miroir

1.12 Longueur d'une liste (`List.length`)

[34]:
```
let rec length l = match l with
| [] -> 0
| hd::tl -> 1 + length tl
```

[34]:
```
val length : 'a list -> int = <fun>
```

que l'on peut aussi écrire, sous une forme plus stylisée

[35]:
```
let rec length = function
| [] -> 0
| _ :: tl -> 1 + length tl
```

[35]:
```
val length : 'a list -> int = <fun>
```

On peut toujours remplacer `let f x = match x with ...` par la construction `let f = function ...`.

Le mot-clé `function` permet de définir des **fonctions anonymes**, comme `lam x: ...` en Python.

Nous aurons l'occasion d'en reparler…

Étudions maintenant la **complexité** de notre fonction `length`. Si la liste `l` est de longueur n, on fera n appels récursifs; chaque appel récursif prend un temps constant (le temps de sommer deux entiers).

On a donc une complexité en temps en $\mathcal{O}(n)$, autrement dit une complexité en temps linéaire.

Il ne s'agit pas ici de rentrer dans la définition exacte de cette notion, vous reverrez cela un peu plus sérieusement dans un cours d'algorithmique et très sérieusement dans un cours de calculabilité.

1.13 Présence dans une liste (`List.mem`)

Voyons maintenant comment définir par cas la fonction `mem x l` qui renvoie `true` si `x` est l'une des valeurs de `l`.

[36]:
```
let rec mem x = function
| [] -> false         (* CAS DE BASE : la liste vide ne peut pas contenir x *)
| head :: tail ->     (* SI LA LISTE N'EST PAS VIDE *)
  if x = head         (* soit x apparait en tête de liste *)
  then true
  else mem x tail      (* soit x apparait dans la queue de la liste *)
```

[36]:
```
val mem : 'a -> 'a list -> bool = <fun>
```

Complexité $\mathcal{O}(n)$, puisque *dans le pire cas* on fait n appels récursifs.

[37]:
```
let rec mem x = function (* équivalent, mais écrit plus compact *)
| [] -> false
| hd::tl -> x=hd || mem x tl
```

[37]:
```
val mem : 'a -> 'a list -> bool = <fun>
```

Le deuxième cas, `hd::tl`, contient deux sous-cas. Faisons-les remonter pour avoir une définition par cas avec 3 cas.

Nous allons utiliser le mot-clé `when`

[38]:
```
let rec mem x = function
| [] -> false
| head::_ when head = x -> true  (* <- notez le *when* *)
| _::tail -> mem x tail
```

[38]:
```
val mem : 'a -> 'a list -> bool = <fun>
```

Vous pourriez être tentés de remplacer le deuxième cas par `x::_`, mais ça ne marche pas.

[39]:
```
let rec mem x = function
| [] -> false
| x::_ -> true
| _::tail -> mem x tail
```

```
File "[39]", line 4, characters 2-9:
4 | | _::tail -> mem x tail
      ^^^^^^^
Warning 11: this match case is unused.
```

[39]:
```
val mem : 'a -> 'b list -> bool = <fun>
```

Le motif `x::_` signifie “appelons désormais `x` le premier élément de la liste”. On est en train de (re)définir localement la variable `x` en lui donnant pour valeur celle de la tête de `li`.

Du coup, OCaml nous avertit que le dernier cas `_::tail` est inutile. Et en effet, il est entièrement couvert par le précédent, `x::_`, qui englobe toute liste de longueur non nulle.

1.14 L’accès à la $n$ième valeur de la liste (`List.nth`)

[1]:
```
let rec nth l n = match (l, n) with
| (x::_, 0) -> x
| (_::tl, i) -> nth tl (i-1)
| ([], _) -> invalid_arg "erreur dans nth n l : n<0 ou n>=length l."
```

[1]:
```
val nth : 'a list -> int -> 'a = <fun>
```

La fonction `invalid_arg` permet de terminer la fonction en faisant une erreur avec un message d’explication. Si l’on ne veut pas avoir un avertissement d’OCaml, il faut bien traiter ce troisième cas d’une façon ou d’une autre!

[41]:
```
nth [1;2;3] 2
```

[41]:
```
- : int = 3
```

[2]:
```
nth [1;2;3] 3  (* en fait, invalid_arg "lève une exception" *)
```

```
Exception: Invalid_argument "erreur dans nth n l : n<0 ou n>=length l.".
Raised at file "stdlib.ml", line 30, characters 25-45
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

1.15 Concaténation de listes (`List.append`)

[43]:
```
let rec append l1 l2 = match l1 with
| [] -> l2
| x::tl -> x::append tl l2
```

[43]:
```
val append : 'a list -> 'a list -> 'a list = <fun>
```

COMPLEXITÉ : $\mathcal{O}(n)$, et oui… on verra comment faire un temps constant avec des listes *mutables*, mais avec des listes non mutables, on ne peut pas espérer mieux.

1.16 Image miroir d'une liste (`List.rev`)

[44]:
```
let rec rev = function
| [] -> []
| hd::tl -> (rev tl) @ [hd]
```

[44]:
```
val rev : 'a list -> 'a list = <fun>
```

Regardons de près la complexité de `rev`.

On doit faire n appel récursifs, et au cours d'un appel récursif il faut faire un append, qui prend un temps $\mathcal{O}(|tl|)$.

Le temps total est donc, à un facteur prêt

$$n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2}$$

soit une **complexité en $\mathcal{O}(n^2)$.**

On peut faire mieux!

D'ailleurs, la fonction `List.rev` de la librairie standard est de complexité $\mathcal{O}(n)$.

Pour cela, il va falloir s'interdire d'utiliser `@` ou `List.append`… Une idée?

L'astuce consiste à généraliser le problème: nous allons nous intéresser à la fonction `rev_append l1 l2`, qui concatène `l2` à l'image miroir de `l1`

[45]:
```
List.rev_append [1;2;3] [4;5;6]
```

[45]:
```
- : int list = [3; 2; 1; 4; 5; 6]
```

Re-programmons maintenant cette fonction `rev_append`.

[46]:
```
let rec rev_append l1 l2 = match l1 with
| [] -> l2
| hd::tl -> rev_append tl (hd::l2)
```

[46]:
```
val rev_append : 'a list -> 'a list -> 'a list = <fun>
```

À partir de `rev_append`, on peut définir la fonction `rev` assez facilement. Et cette fois-ci, on obtient bien une complexité en $\mathcal{O}(n)$.

[47]:
```
let rev l1 = rev_append l1 []
```

[47]:
```
val rev : 'a list -> 'a list = <fun>
```

1.17 Comment utiliser `List.sort`?

La fonction `List.sort` permet de trier une liste. Elle attend deux arguments:

- une fonction pour savoir sur quel critère trier la liste
- la liste à trier

Si l'on veut trier avec le critère "standard", on peut utiliser la fonction `compare` de la librairie standard

[48]:
```
(* ordre croissant sur les entiers *)
List.sort compare [3;2;1;4]
```

[48]:
```
- : int list = [1; 2; 3; 4]
```

[49]:
```
(* ordre alphabétique croissant *)
List.sort compare ["un"; "zèbre"; "pourchasse"; "le"; "lyon"]
```

[49]:
```
- : string list = ["le"; "lyon"; "pourchasse"; "un"; "zèbre"]
```

[50]:
```
(* ordre lexicographique croissant *)
List.sort compare [(1,1); (2,1); (1,3); (2,2)]
```

[50]:
```
- : (int * int) list = [(1, 1); (1, 3); (2, 1); (2, 2)]
```

[51]:
```
compare (* le type de la fonction compare de la librairie standard est : *)
```

[51]:
```
- : 'a -> 'a -> int = <fun>
```

Pour changer le critère de tri, il faut fournir la bonne fonction de comparaison.

Une fonction de comparaison `cmp` pour trier une `'a list` doit avoir le type

`cmp:'a -> 'a -> int`.

`cmp x y` renverra - 1 si x est strictement plus grand que y (au sens de cmp) - -1 si y est strictement plus grand que x - 0 sinon (x et y sont "équivalents").

Quelques exemples de fonctions de comparaison

[52]:
```
let decroissant x y =  (* decroissant x y = compare y x *)
   if x < y then 1 else if x > y then -1 else 0
in List.sort decroissant [1;2;3;4]
```

[52]:
```
- : int list = [4; 3; 2; 1]
```

[53]:
```
let norme2 (x,y) = x**2. +. y**2.
```

[53]:
```
val norme2 : float * float -> float = <fun>
```

[54]:
```
(* le tri par rapport à la norme *)
let cmp v1 v2 = compare (norme2 v1) (norme2 v2) in
List.sort cmp [(-. 1., -. 1.); (1., 1.); (0., 0.); (0.5, 2.)]
```

[54]:
```
- : (float * float) list = [(0., 0.); (-1., -1.); (1., 1.); (0.5, 2.)]
```

On préfèrera parfois la fonction `List.stable_sort` qui laisse deux éléments "équivalents" dans l'ordre dans lequel ils se trouvent dans la liste de départ.

Avec `sort`, il est possible d'intervertir deux éléments qui sont équivalents (par exemple (1,1) et (-1,-1) dans l'exemple précédent).

Il y aussi une fonction `List.fast_sort` qui est plus rapide.

1.18 Application : un algorithme de recommandation

Nous allons maintenant écrire un algorithme de recommandation pour une compagnie qui vend des bonbons en ligne.

Cette compagnie propose deux produits, des bonbons bleus et des bonbons rouges. Notre but est de recommander à un nouveau client un bonbon de la bonne couleur en devinant celle qu'il préfère à partir de son profil.

Pour cela, on cherche d'anciens clients ayant un profil similaire et on se base sur leur préférence pour deviner.

Pour faire simple, un profil est un couple de flottants `(age, revenu)`. On a donc un nuage de points du plan rouges ou bleus qui correspondent aux profils des anciens clients.

[55]:
```
let nuage = [(13., 50., "bleu"); (80., 2000., "rouge"); (19., 400., "bleu") (*␣
↪etc *)]
```

[55]:
```
val nuage : (float * float * string) list =
  [(13., 50., "bleu"); (80., 2000., "rouge"); (19., 400., "bleu")]
```

Notre algorithme de recommandation sera très simple: pour calculer la couleur au point `p`, on va déterminer la couleur **majoritaire** parmi **les trois points les plus proches** de `p` dans le nuage.

[56]:
```
let recommande nuage (age, revenu) =
   let dist (age', revenu', couleur) = norme2 (age' -. age, revenu' -. revenu)
 ↪in
   let cmp p1 p2 = compare (dist p1) (dist p2) in
   match List.sort cmp nuage with
   | _::(_,_,c2)::(_,_,c3)::_ when c2=c3 -> c2
   | (_,_,c1)::_ -> c1
   | _ -> invalid_arg "le nuage doit contenir au moins 1 point"
```

[56]:
```
val recommande : (float * float * 'a) list -> float * float -> 'a = <fun>
```

Et voilà!

Allez, pour finir, nous allons représenter le nuage de points dans une fenêtre graphique et utiliser la souris pour sélectionner le profil à deviner.

1.19 Illustration : introduction à la librairie Graphics

Commençons par un petit dessin

[57]:
```
#load "graphics.cma";;  (* <- au toplevel uniquement: pour "charger" le module
 ↪*)

open Graphics;;         (* from Graphics import *, si on était en Python *)

let () = begin
   open_graph " 600x400";   (* crée une nouvelle fenêtre 600 sur 400 *)
                            (* NOTE : faites attention à l'espace devant 600 et
 ↪au x minuscule *)
                            (* Graphics risque de bouder si vous les oubliez! *)
   set_color 0xcc33ff;      (* code RGB de la couleur en hexadecimal, ici violet
 ↪*)
   fill_circle 0 0 10;      (* dessine un cercle plein en bas à gauche de rayon
 ↪10 *)
   fill_circle 300 300 10;
   fill_circle 150 50 10;
   set_color red;
   moveto 0 0;              (* déplace la "tortue" en (0,0) sans tracer *)
   lineto 300 300;          (* déplace la "tortue" en (300,300) en traçant *)
   lineto 150 50;
   lineto 0 0               (* <- pas de ; après la dernière instruction *)
end
```

```
Exception: Graphics.Graphic_failure "Cannot open display :0".
Raised by primitive operation at unknown location
Called from unknown location
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

1.20 Comment tracer le graphe d'une fonction?

On va **discrétiser** la fonction $f : \mathbb{R} \to \mathbb{R}$ à tracer.

Les coordonnées dans le canevas d'OCaml sont **entières**, et on veut tracer une fonction **flottante**. Il faut donc la transformer en une fonction $g : \mathbb{N} \to \mathbb{N}$

[58]:
```
let discretise f x = int_of_float (f (float_of_int x))
```

[58]:
```
val discretise : (float -> float) -> int -> int = <fun>
```

Pour le tracé proprement dit, on va se placer en (0,f(0)), puis on avance en (1,f(1)), puis (2,f(2)), etc.

[59]:
```
let trace f =
  let f = discretise f in
  let rec trace_depuis x = begin
    lineto (x+1) (f (x+1));
    if x = size_x() then () (* si on est arrivé au bord droit, on s'arrête  *)
    else trace_depuis (x+1) (* sinon on continue le tracé en partant de x+1 *)
  end
  in begin
    moveto 0 (f 0);
    trace_depuis 0
  end
```

[59]:
```
val trace : (float -> float) -> unit = <fun>
```

[60]:
```
let f x = 400. /. (1. +. (x /. 60. -. 5.) ** 2.)
```

[60]:
```
val f : float -> float = <fun>
```

[61]:
```
let () = trace f
```

```
Exception: Graphics.Graphic_failure "graphic screen not opened".
Raised by primitive operation at file "[59]", line 4, characters 4-26
Called from unknown location
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

1.21 Comment générer un nuage de points?

On va simplement tirer 100 points au hasard: s'ils sont au-dessus de la courbe, ils sont bleus, sinon ils sont rouges.

[62]:
```
let couleur_point (x,y) =
  if f x > y then red else blue
```

[62]:
```
val couleur_point : float * float -> Graphics.color = <fun>
```

[63]:
```
let tire_point () =
   let (x, y) = (Random.float 600., Random.float 400.)
   in (x, y, couleur_point(x, y))
```

[63]:
```
val tire_point : unit -> float * float * Graphics.color = <fun>
```

[64]:
```
let rec tire_nuage_de_points = function
|  0 -> []
|  n -> tire_point() :: tire_nuage_de_points (n-1)
```

[64]:
```
val tire_nuage_de_points : int -> (float * float * Graphics.color) list =
  <fun>
```

[65]:
```
let nuage = tire_nuage_de_points 100
```

[65]:
```
val nuage : (float * float * Graphics.color) list =
  [(29.7452414129209259, 211.550216603539809, 255);
   (464.470170800856522, 191.848245586970592, 255);
   (295.214399322990516, 309.023029391436921, 16711680);
   (84.4472774314643857, 231.381283515723226, 255);
   (386.184755147091209, 125.852099024178131, 16711680);
   (235.26413174639751, 112.966757718757705, 16711680);
   (434.302587950570398, 274.422116011127457, 255);
   (287.365625118958405, 43.2861139467556626, 16711680);
   (506.258539783803656, 226.39104103169737, 255);
```

```
(551.322558348285838, 17.8396763547505, 16711680);
(82.7543266768089296, 299.01439582299065, 255);
(78.0055787073448101, 68.0882024596773903, 255);
(147.947903033208121, 318.199409740674298, 255);
(225.688133214431275, 121.161752079398283, 16711680);
(63.7715153342926868, 17.0889017870772584, 16711680);
(237.980650248506549, 329.914072498763062, 255);
(517.377865867396622, 137.752697694173236, 255);
(223.08902664452043, 133.545577857197486, 16711680);
(444.683167644707567, 42.2502304857296309, 16711680);
(517.465013784013422, 363.143238064615218, 255);
(490.194362604381297, 174.63870273583, 255);
(32.1499450042766526, 213.147197635300046, 255);
(405.660056272429131, 396.88089926213604, 255);
(390.14587667126915, 95.1738973144892, 16711680);
(225.165515131871643, 330.985780531387718, 255);
(445.934951999690895, 338.792040651539139, 255);
(369.657680075572159, 36.0362966860596643, 16711680);
(180.668443060806879, 63.6216916824679544, 16711680);
(188.627347610641891, 178.132301079265375, 255);
(484.391600686855213, 272.916229670187704, 255);
(168.520365736478567, 152.076396758565721, 255);
(137.737010842950838, 16.7291840637006537, 16711680);
(159.268437515405935, 310.131445723661955, 255);
(375.963318863291192, 349.144478378535723, 255);
(228.718427552120204, 238.431594657959153, 255);
(506.000090081186272, 96.4993462200926473, 255);
(377.380347911536433, 132.4097985562004, 16711680);
(505.350536934978379, 82.9191475543386076, 255);
(360.327028569818197, 195.70451982897535, 16711680);
(369.537221225746123, 136.721285053287602, 16711680);
(431.234895132322833, 295.463274766858603, 255);
(485.350826594508533, 376.599426661228563, 255);
(302.352314590393291, 141.757783764713025, 16711680);
(546.433030181966, 162.351090994501703, 255);
(526.706553003804743, 247.185520833001021, 255);
(285.280597089109619, 73.8683858080189708, 16711680);
(172.187111706815756, 315.618535389622934, 255);
(268.013115503100153, 277.098054114633442, 16711680);
(479.873096569196036, 5.50793542676871084, 16711680);
(521.504206418355693, 262.818697540797928, 255);
(381.581228844408656, 370.021666522402199, 255);
(402.662430567768467, 265.652287496980307, 255);
(320.869525987830627, 170.589622826218459, 16711680);
(86.8110179052320206, 115.642907689877546, 255);
(591.955930035988104, 367.671978186773458, 255);
(305.991079417598826, 237.763062233857397, 16711680);
```

```
(177.329447631798615, 135.698870972734738, 255);
(518.768885721841457, 307.173935160373958, 255);
(14.1884146915652121, 235.02688614771813, 255);
(183.458628693541584, 397.795085496454192, 255);
(1.41218903378545657, 309.790331206024689, 255);
(590.230350637869606, 214.134127085197576, 255);
(221.407267320139312, 95.9139841745278119, 16711680);
(504.548827935764962, 171.586735574556343, 255);
(449.368336875404111, 96.256944581075, 255);
(270.650420334689613, 128.984943716520775, 16711680);
(298.77645878765253, 59.1055188059368817, 16711680);
(518.630141298657918, 21.458612566375578, 16711680);
(423.050944406842859, 396.040391446398, 255);
(237.032612927103514, 351.220809377906676, 255);
(499.71158888107675, 336.747918183631725, 255);
(118.767154276580143, 95.3727260928084917, 255);
(371.81316122936596, 162.185702986226204, 16711680);
(283.311939888907489, 257.74004739095642, 16711680);
(5.91864974457724458, 90.680652953830986, …); …]
```

1.22 Comment dessiner un nuage de points?

Par récurrence! On commence par dessiner un point

[66]:
```
let dessine_point(x,y,c) = begin
  set_color c;
  fill_circle (int_of_float x) (int_of_float y) 5
end
```

[66]:
```
val dessine_point : float * float * Graphics.color -> unit = <fun>
```

Puis par récurrence on parcours la liste des points et on les dessine un à un.

[67]:
```
let rec dessine_nuage = function
| [] -> () (* rien à dessiner! *)
| p::tl -> begin dessine_point p; dessine_nuage tl end
```

[67]:
```
val dessine_nuage : (float * float * Graphics.color) list -> unit = <fun>
```

[68]:
```
let () = dessine_nuage nuage
```

```
Exception: Graphics.Graphic_failure "graphic screen not opened".
Raised by primitive operation at file "[66]", line 2, characters 2-13
Called from file "[67]", line 3, characters 17-32
Called from unknown location
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

OCaml propose aussi une fonction d'itération `List.iter` (c'est une **fonction d'ordre supérieur**, cf prochains cours)

`List.iter f [a1;a2;...;an]` est équivalent à

`begin f a1;f a2;...;f an end`.

On peut donc écrire `dessine_nuage` plus simplement.

[69]:
```
let dessine_nuage nuage = List.iter dessine_point nuage
(* ou plus simplement *)
let dessine_nuage = List.iter dessine_point
```

[69]:
```
val dessine_nuage : (float * float * Graphics.color) list -> unit = <fun>
```

[69]:
```
val dessine_nuage : (float * float * Graphics.color) list -> unit = <fun>
```

1.23 Comment tester l'algorithme de recommandation?

On va récupérer la position de la souris et dessiner un point à l'endroit où elle se trouve, en se basant sur notre algorithme de recommandation pour déterminer sa couleur.

Il faudra tout redessiner chaque fois que la souris bouge.

[70]:
```
let redessine _ =  begin
  clear_graph ();                    (* effacer tous les dessins *)
  trace f;                           (* tracer la courbe en cloche, la␣
 ↪"frontière" à deviner *)
  dessine_nuage nuage;               (* dessiner les points connus *)
  let (x,y) = mouse_pos () in        (* recuperer la position de la souris *)
  let (x,y) = ((float_of_int x),(float_of_int y)) in
  let c = recommande nuage (x,y) in  (* calculer la couleur recommandée en ce␣
 ↪point *)
  dessine_point(x,y,c)               (* dessiner ce nouveau point*)
end
```

[70]:
```
val redessine : 'a -> unit = <fun>
```

Il ne reste plus qu'à installer la fonction de dessin dans la boucle infinie de Graphics, et demander à l'appeler à chaque fois que la souris est déplacée.

[71]:
```
let () = begin
  open_graph " 600x400";
  loop_at_exit [Mouse_motion] redessine  (* similaire à quelque chose comme
 ↪*)
                              (* while true do begin wait_mouse(); redessine()
 ↪end;*)
end
```

```
Exception: Graphics.Graphic_failure "Cannot open display :0".
Raised by primitive operation at unknown location
Called from unknown location
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```