cours6

November 21, 2023

1 Programmation fonctionnelle

1.1 Semaine 6 : Compilation et modules

Etienne Lozes - Université Nice Sophia Antipolis - 2019

---

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

1.2 La compilation

La compilation peut se définir comme l'activité de traduire un "langage de programmation" dans un autre.

Il y a beaucoup de sortes de "langages de programmation":

- les **langages assembleurs**: ce sont des suites d'instructions élémentaires exécutées par un micro-processeur. Les premiers programmeurs ne programmaient qu'en assembleur! Aujourd'hui, les humains programment peu en assembleur, et plutôt sur des portions de code très restreintes.
- les **langages "haut-niveau"** : ce sont ceux que vous apprenez à programmer à l'université (Python, C, Java, Javascript, OCaml,...)
- des **"représentations intermédiaires"** : ce sont des sortes d'esperanto de la compilation, personne ne les utilise pour programmer, aucun micro-processeur ne sait les lire directement... mais ils sont très utiles!

1.3 Les compilateurs OCaml

Les deux compilateurs "officiels" sont `ocamlc` et `ocamlopt`, qui produisent respectivement du **bytecode** et du **code natif**.

Il en existe aussi un troisième "non officiel" mais très intéressant: `js_of_ocaml`, qui produit du **javascript**.

1.4 `ocamlc` : le compilateur "bytecode"

Le compilateur `ocamlc`, dans son usage le plus simple, transforme un fichier texte `.ml` en un fichier binaire "bytecode". Sous linux/mac, le bytecode s'appele `a.out` par défaut, et sous windows `camlprog.exe`. On peut spécifier le nom du fichier bytecode avec l'option `-o`. Pour exécuter le bytecode, il faut le passer à l'**interpréteur** `ocamlrun`

```
(* fichier toto.ml *)
let () = print_int (1+1)
```

Dans le shell:

```
$ ocamlc toto.ml -o toto.byte
$ ocamlrun toto.byte
2
$
```

Contrairement aux apparences, le fichier `toto.byte` n'est pas un "vrai" exécutable: il a besoin du programme `ocamlrun` pour s'exécuter. On peut cependant en général lancer directement `toto.byte`: sous linux, le shell regarde si la première ligne du fichier indique un **interpréteur**: si c'est le cas, l'interpréteur est appelé sur le fichier.

```
$ cat toto.byte
#!/Users/lozes/.opam/default/bin/ocamlrun
T^@^@^@<DF>^B^@^@^@^@^@^@W^@^@^@^A^@^O^@^P^@^@^@^S^@^@^@^\^@^@^@%^@^@^@.^@^@^@7
^@^@^@@^@^@^@I^@^@^@R^@^@^@[^@^@^@g^@^@^@t^@^@^@}^@^@^@<86>^@^@^@<8F>^@^@^@<98>
[...]
$ ./toto.byte
2
$
```

Supposons que je suis sous ubuntu 19.10 sur une machine Intel iCore9 (par exemple!).

Je compile `toto.ml`, et j'envoie `toto.byte` à ma grand mère. Elle aussi est sous ubuntu 19.10 et sur une machine Intel iCore9.

- Si elle a installé `ocamlrun` dans le même répertoire `#!/Users/lozes/.opam/default/bin/ocamlrum` que moi, il y a une chance, pour un programme aussi simple, qu'elle arrive à l'exécuter.
- Mais si elle n'a pas `ocamlrun`, ou simplement s'il n'est pas dans le même répertoire sur sa machine (probable!), la commande `./toto.byte` ne marchera pas.

Ce que contient le fichier `toto.byte`, au-delà de la première ligne, c'est une "représentation intermédiaire" propre à OCaml.

Le fichier `toto.byte` est en fait un programme dans un langage que Xavier Leroy a appelé **Zinc** (pour Zinc Is Not Caml).

`toto.byte` contient le programme Zinc sous un format binaire compressé non lisible à l'oeil nu. On peut demander au compilateur `ocamlc` de nous donner une idée du code Zinc avec l'option `-dinstr`.

```
$ ocamlc -dinstr toto.ml
    const 1
```

```
offsetint 1
push
getglobal Stdlib!
getfield 43
apply 1
push
makeblock 0, 0
pop 1
setglobal Print_int!
```

Le langage Zinc est un "langage orienté pile": certaines instructions (notamment `push`, `pop` et `apply`) manipulent une pile. D'autres font référence à des variables globales prédéfinies (~registres) qui contiennent des fonctions de la libraire standard comme `(+)` ou `print_int`.

Il n'est pas utile de rentrer plus dans le détail sur Zinc, mais ce qu'il faut retenir sur Zinc, c'est

- ce n'est pas de l'assembleur
- c'est un langage "bas niveau"
- c'est un langage indépendant du micro-processeur et de l'OS : il est **portable**
- pour pouvoir exécuter le bytecode, j'ai besoin de l'intepréteur Zinc `ocamlrun`, souvent appelé **machine virtuelle**

Autrement dit, si j'envoie mon linux/intel `toto.byte` à ma grand-mère unix/MIPS, mais si elle a installé ocamlrun chez elle, elle pourra l'exécuter (en théorie) en tapant `ocamlrun toto.byte`.

1.5 L'option `-custom`

Imaginons que: - je veux envoyer un `toto.byte` à ma grand-mère linux/intel (comme moi) et je veux qu'elle puisse l'exécuter en tapant `./toto.byte`, mais elle n'a pas installé `ocaml`. - je veux simplement pouvoir continuer à exécuter `toto.byte` après avoir désinstallé `ocaml` de ma machine

Comment faire?

Je peux compiler avec l'option `-custom`, qui a pour effet d'ajouter `ocamlrun` dans le fichier `toto.byte`.

Le fichier est plus gros, et ne commence plus par la ligne `#!/.../ocamlrun`.

```
$ ocamlc -o toto.byte toto.ml ; du -h toto.byte
24K toto.byte
$ ocamlc -custom -o toto.byte toto.ml ; du -h toto.byte
368K    toto.byte
$ cat toto.byte
<CF><FA><ED><FE>^G^@^@^A^C^@^@<80>^B^@^@^@^P^@^@^@H^F^@^@<85>
[...]
```

Question Si j'envoie `toto.byte` à ma grand-mère unix/MIPS est-ce qu'elle pourra l'exécuter?

Réponse: NON! Le bytecode est portable, mais la machine virtuelle, elle, n'est pas portable. Chaque config a besoin de sa propre machine virtuelle Zinc, autrement dit son propre `ocamlrun`.

Cette technique de `bytecode` et de `machine virtuelle` est aujourd'hui très courante: `Java` et `Python`, entre autres, sont eux aussi compilés vers du bytecode qui est interprété par une machine virtuelle.

1.6 Le compilateur `ocamlopt`

On l'utilise à peu près comme le compilateur `ocamlc`

```
$ ocamlopt -o toto.opt toto.ml
$ ./toto.opt
2
```

mais cette fois-ci, l'exécutable généré est un programme en assembleur.

On peut lire le code assembleur directement dans le fichier binaire `toto.opt` avec certains utilitaires (ex: `objdump -d toto.opt`) ou demander à `ocamlopt` de générer un fichier texte `toto.s` qui contient le code assembleur (option `-S`).

Sur mon mac OS X 10.15 avec un processeur Intel Core i7

```
$ ocamlopt -S toto.ml
$ cat toto.s
```

[...]

```
L102:
        movq    $5, %rax
        call    _camlStdlib__string_of_int_161
L100:
        movq    %rax, %rbx
        movq    _camlStdlib@GOTPCREL(%rip), %rax
        movq    304(%rax), %rax
        call    _camlStdlib__output_string_224
L101:
        movq    $1, %rax
        addq    $8, %rsp
```

[...]

Attention ce code natif n'est pas portable, c'est du code assembleur qui est compris par mon microprocesseur avec des appels systèmes spécifiques à mon OS. Sur une autre configuration, l'addition de deux entiers est une autre instruction d'assembleur, l'appel système pour faire un affichage n'a pas le même numéro, etc.

Quel intérêt? Certes, le code n'est pas portable, mais il est *optimisé* pour ma machine. Un programme compilé avec `ocamlopt` est quasiment toujours plus rapide que le même programme compilé avec `ocamlc`. `ocamlopt` propose de nombreuses options pour paramétrer les optimisations qui sont faites en fonction du code à compiler.

1.7 Le compilateur `js_of_ocaml`

Vous n'aurez pas à utiliser ce compilateur cette année! Mais `js_of_ocaml` est trop intéressant pour ne pas en dire deux mots rapides.

De quoi s'agit-il? D'un compilateur dont le langage cible est **javascript**.

Javascript? c'est *le* langage du web interactif. Tous les navigateurs webs savent l'exécuter, soit en l'interprétant, soit en le compilant à la volée (*just in time*). Un programme javascript est donc extrêmement portable: quel que soit l'OS et le microprocesseur, le code est toujours le même.

Et `js_of_ocaml`? Il permet de compiler du code OCaml en javascript, et donc de programmer un site web interactif en OCaml. C'est comme cela qu'a été programmé le site `tryOCaml`, par exemple... De plus `js_of_ocaml` travaille à partir du bytecode Zinc, qui n'a pratiquement pas changé depuis le début de Caml. Il est donc très stable par rapport aux évolutions à venir d'OCaml.

Comment fonctionne-t-il? `js_of_ocaml` pourrait n'être qu'une simple machine virtuelle capable d'exécuter du bytecode Zinc, un `ocamlrun` programmé en javascript, en quelque sorte. En fait, `js_of_ocaml` est un "décompilateur": il lit du bytecode Zinc, "identifie" des fonctions dans ce code bas-niveau, et reconstruit des fonctions javascript. C'est beaucoup moins trivial à faire qu'une machine virtuelle! mais cela donne des performances étonnantes. Le code javascript peut parfois être plus rapide que le bytecode!

1.8 Comment debugger mon programme OCaml?

Il y a essentiellement trois écoles: - ajouter des `print` dans mon programme - utiliser la directive `#trace` dans un toplevel - utiliser `ocamldebug`

1.8.1 La directive `#trace`

Elle permet d'afficher les paramètres qui sont passés à une fonction à chaque fois qu'elle est appelée, et d'afficher le résultat de la fonction

[2]:
```
let rec fact = function 0 -> 1 | n -> n * fact (n-1)
```

[2]:
```
val fact : int -> int = <fun>
```

[3]:
```
#trace fact
```

[4]:
```
fact 2
```

```
fact is now traced.
```

[4]:
```
- : int = 2
```

```
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
```

```
fact --> 1
fact --> 2
```

Il y a quelques petites subtilités qu'il faut connaitre pour : - si la fonction tracée prend plusieurs arguments, ils sont tracés un à un. - il faut spécifier le type d'un argument et éviter le polymorphisme pour que le toplevel sache quels pretty-printers utiliser.

[5]:
```
let somme a b = a+b
```

[5]:
```
val somme : int -> int -> int = <fun>
```

[6]:
```
#trace somme
```

[7]:
```
somme 1 2
```

```
somme is now traced.
somme <-- 1
```

[7]:
```
- : int = 3
```

```
somme --> <fun>
somme* <-- 2
somme* --> 3
```

[8]:
```
let f x = [x]
```

[8]:
```
val f : 'a -> 'a list = <fun>
```

[9]:
```
#trace f
```

[10]:
```
f 1
```

```
f is now traced.
```

[10]:
```
- : int list = [1]
```

[11]:
```
let f (x:int) = [x]
```

```
f <-- <poly>
f --> [<poly>]
```

[11]:
```
val f : int -> int list = <fun>
```

[12]:
```
#trace f
```

[13]:
```
f 1
```

```
f is now traced.
f <-- 1
f --> [1]
```

```
[13]: - : int list = [1]
```

1.8.2 Le debugger `ocamldebug`

Il s'agit d'un utilitaire qui permet:

- d'insérer des **points d'arrêts** (breakpoints) dans le code
- d'inspecter les valeurs des variables, la pile d'appel, le code autour du point d'arrêt…
- de reprendre l'exécution ou de revenir en arrière

C'est un outil similaire à `gdb` pour C. Il n'est pas spécialement convivial, même si on peut (un peu) arranger cela en l'utilisant depuis `emacs`…

```
(* fichier loop.ml *)
let rec f i =
  print_int i;
  print_newline ();
  f (i+1)

let () = f 0
```

Je compile avec l'option `-g`:

```
ocamlc -g -o loop.byte loop.ml
```

Je lance une session de debugger

```
ocamldebug loop.byte
    OCaml Debugger version 4.07.0
(ocd)
```

J'installe un point d'arrêt à l'entrée de la fonction f, puis je lance l'exécution du programme.

```
(ocb) break @ Loop 1
Loading program... done.
Breakpoint 1 at 10492: file loop.ml, line 3, characters 3-45
(ocd) run
Time: 12 - pc: 10492 - module Loop
Breakpoint: 1
3   <|b|>print_int i;
(ocd)
```

Je demande à afficher la valeur de i, et je relance l'exécution

```
(ocd) print i
i: int = 0
(ocd) run
```

```
0
Time: 21 - pc: 10492 - module Loop
Breakpoint: 1

(ocd) print i
i: int = 1
(ocd)
```

1.9 La compilation séparée

Lorsque le projet sur lequel je travaille devient un peu important, je peux avoir envie de le découper en plusieurs fichiers: - pour pouvoir regrouper les fonctions qui vont ensemble et les retrouver plus facilement - pour ne pas avoir à tout recompiler à chaque fois - pour pouvoir travailler en équipe - pour pouvoir utiliser certaines fonctions sur plusieurs projets - ...

- Chaque fichier `.ml` de mon projet correspond à un **module**.
- Le nom du fichier correspond au nom du module: le fichier `params.ml` contient les définitions du module `Params`
- Je peux parler de l'élément `xxx` du module `Params` en utilisant la notation `Params.xxx`

```
(* fichier params.ml *)
let version = 17
let get_name () = "toto"

(* fichier principal.ml *)
let () = Printf.printf "%s %d " (Params.get_name ()) Params.version
```

Sur cet exemple, le module `Principal` dépend du module `Params` concernant la constante `version` et la fonction `get_name`.

Au moment de compiler mon programme, je vais lister tous les fichiers `.ml` dont il est constitué, **dans l'ordre de leurs dépendances**.

```
$ ocamlc -o toto.byte params.ml principal.ml
$ ./toto.byte
toto 17
$
```

Attention il ne peut pas y avoir de dépendances circulaires entre plusieurs fichiers `.ml`...

La **compilation séparée** permet de compiler indépendamment chaque fichier `.ml` en un *fichier objet* d'extension `.cmo` (bytecode) ou `.cmx` (code natif). On demande une compilation séparée en utilisant l'option `-c`.

```
$ ocamlc -c params.ml
$ ls
params.cmo params.ml principal.ml ...
$ ocamlopt -c params.ml
$ ls
params.cmo params.cmx params.ml principal.ml ...
```

Remarque: il y a aussi un fichier `params.cmi`, on y vient…

Qu'est-ce qu'un fichier objet? C'est un fragment du fichier final dans lequels les appels de fonctions *ne sont pas résolus*. Par exemple, dans `principal.cmo`, il y a un appel à une fonction `Params.get_name` dont on ne connait pas encore l'adresse dans le code consolidé final.

C'est **l'édition de liens** qui consolide les différents fichiers objets en un seul exécutable.

```
$ ocamlc -c params.ml
$ ocamlc -c principal.ml
$ ocamlc -o toto.byte params.cmo principal.cmo
$ ./toto.byte
```

Remarques - Je ne peux pas lier du bytecode avec du code natif, je dois utiliser le même compilateur (ici `ocamlc`) du début à la fin. - La commande `ocamlc -o toto.byte params.ml principal.ml` est équivalente à la suite de commandes ci-dessus.

1.10 Les librairies

Ce sont des fichiers compilés `.cma` (bytecode) ou `.cmxa` (natif) qui contiennent des modules. La librairie standard `stdlib.cma` contient de nombreuses fonctions et types "standards"; tout ce que nous avons vu jusqu'à présent est défini dans la librairie standard, à l'exception du module graphics. La librairie standard est automatiquement liée aux fichiers compilés.

Si je veux utiliser une librairie "non standard", comme la librairie graphics ou la libraire unix, je dois la lier aux autres fichiers `.cmo` au moment de l'édition de liens.

```
$ cat echo_hello_world.ml
Unix.system "echo \"hello, world!\""
$ ocamlc -c echo_hello_world.ml
$ ocamlc unix.cma echo_hello_world.cmo
$ ./a.out
hello, world!
$ ocamlopt -c echo_hello_world.ml
$ ocamlopt unix.cmxa echo_hello_world.cmx
```

Où `ocamlc` va-t-il chercher les fichiers `.cmo` et `.cma`? - d'abord dans le répertoire courant - ensuite dans les répertoires éventuellement spécifiés par une option `-I` - ensuite dans le répertoire qui contient la librairie standard. La commande `ocamlc -where` indique ce répertoire

```
$ mkdir sous_repertoire
$ ocamlc -c echo_hello_world.ml
$ mv echo_hello_world.cmo sous_repertoire/
$ ocamlc -I sous_repertoire unix.cma echo_hello_world.cmo
$ ls $(ocamlc -where)/unix.cma
/Users/lozes/.opam/default/lib/ocaml/unix.cma
$
```

Parfois les libraires installées par `opam` ne se trouvent pas dans le répertoire de la librairie standard, il vous faudra peut-être chercher un peu.

Par exemple, chez moi, pour compiler le fichier pi.ml, qui utilise la librairie `Zarith`, je dois écrire

```
ocamlc -I ~/.opam/default/lib/zarith zarith.cma pi.ml
```

C'est parfois un casse-tête de trouver les bonnes options à passer à `ocamlc`. Certains outils comme `ocamlfind` ou `ocamlbuild` tentent de simplifier la compilation en introduisant une notion de paquetage. En un exemple (lisez leurs documentation au besoin)

```
ocamlfind ocamlc -package zarith -linkpkg pi.ml
```

1.11 Module, paquetages, et toplevel

Il est possible de charger un module ou une librairie dans le toplevel avec la directive `#load`. Si `ocamlfind` est installé, on peut aussi charger un paquetage avec `#require`

```
[14]: #load "unix.cma";;
```

```
[15]: Unix.system "echo \"hello, world!\""
```

```
hello, world!
[15]: - : Unix.process_status = Unix.WEXITED 0
```

```
[16]: #require "zarith";;
```

```
/Users/lozes/.opam/default/lib/zarith: added to search path
/Users/lozes/.opam/default/lib/zarith/zarith.cma: loaded
```

```
[17]: Q.to_string (Q.of_ints 3 7)
```

```
[17]: - : string = "3/7"
```

1.12 Implémentation et interface

Avec l'option `-i`, je peux demander au compilateur de m'afficher toutes les définitions qui ont été faites à l'intérieur du module, et qui sont susceptibles d'être utilisées par un autre module.

```
$ ocamlc -i params.ml
val version : int
val get_name : unit -> string
$
```

On voit que le module `Params` "exporte" deux définitions de valeurs:

- une constante `version` de type `int`, et
- une fonction `get_name` sans argument qui renvoie une chaîne de caractères.

Cette liste constitue l'**interface** du module: c'est un résumé du fichier `params.ml` qui contient tout ce qu'il me suffit de savoir pour pouvoir utiliser le module `Params` dans un autre module.

Je peux écrire un fichier `params.mli` qui décrit l'interface du module `Params`.

C'est un analogue des fichiers *headers* `.h` en C.

```
(* fichier params.mli *)
val version : int
val get_name : unit -> string
```

Typiquement, j'écris d'abord mon fichier `params.mli` pour fixer les idées, puis je rédige ensuite une **implémentation** `params.ml` de cette interface.

Le fichier `.mli` est un contrat que devra respecter le fichier `.ml` de même nom: toutes les définitions qui sont annoncées dans `.mli` doivent se concrétiser dans `.ml`, en respectant les types annoncés.

Je dois compiler le fichier `.mli` avant le fichier `.ml`. Le compilateur vérifie que j'ai bien respecté le contrat.

Sur un exemple: introduisons volontairement une violation de contrat dans `params.ml`.

```
$ ocamlc params.mli
$ cat broken_params.ml
let version = "13" (* <- entier attendu! *)
let get_name () = "hello"
$ cp broken_params.ml params.ml
```

```
$ ocamlc params.ml
Error: The implementation params.ml does not match the interface params.cmi:
       Values do not match:
         val version : string
       is not included in
         val version : int
```

Autre exemple:

```
$ cat broken_params2.ml
let version = 13
(* manquant: get_name *)
$ cp broken_params2.ml params.ml; ocamlc -c params.ml
Error: The implementation params.ml does not match the interface params.cmi:
       The value `get_name' is required but not provided
```

En résumé, pour respecter le contrat du `.mli`, le fichier `.ml` doit

- définir toutes les valeurs, tous les types, toutes les exceptions… annoncés dans le `.mli`
- respecter les types annoncés dans le `.mli`

Lorsqu'un fichier `.ml` n'est pas accompagné d'un fichier `.mli` du même nom, le fichier `.mli` est implicitement celui que produit l'option `-i`.

Par exemple, si `toto.mli` n'existe pas la commande `ocamlc -c toto.ml` est équivalente à la suite de commandes

```
$ ocamlc -i toto.ml > toto.mli
$ ocamlc -c toto.mli
$ ocamlc -c toto.ml
$ rm toto.mli
```

1.13 La compilation séparée sur un exemple

Le fichier `params.mli` me permet de compiler le module `principal.ml` même si je n'ai pas encore vraiment écrit le module `params.ml` qui est utilisé dans `principal.ml`

```
$ ocamlc -c params.mli
$ ocamlc -c principal.ml
[... je crée le fichier params.ml ...]
$ ocamlc -c params.ml
$ ocamlc params.cmo principal.cmo
```

1.14 Abstraction et masquage d'information

Dans l'implémentation, je suis obligé de définir **au moins** tout ce que j'ai annoncé dans l'interface… mais

- je peux faire d'autres définitions non annoncées.
- je peux aussi avoir du code pour initialiser mon module.

```
(* fichier params.mli *)
val version : int
val get_name : unit -> string
```

```
(* fichier params.ml *)
let version = 2
let names = ["toto"; "titi"; "tata"]   (* <- privé! *)
let () = names.(1) <- "tutu"           (* <- initialisation *)
let get_name () = names.(version)
```

Les définitions non annoncées sont **privées**: je ne peux accéder à ces définitions qu'à l'intérieur du module `Params`, le module `Principal`, lui n'y a pas accès.

```
(* fichier principal.ml *)
let () = print_string (Params.get_name ())
let () = print_int (Params.version)

let () = Params.names.(1)  (* <- interdit! *)
```

Au moment de la compilation de `principal.ml`, je vais avoir une erreur…

```
$ ocamlc -c params.mli
$ ocamlc -c params.ml
$ ocamlc -c principal.ml
[...]
```

```
Error: Unbound value Params.names
```

1.15 Les modules ne sont pas que des fichiers

Pour le moment, on a confondu la notion de module et de fichier `.ml` et la notion d'interface et de fichier `.mli`.

On peut cependant définir des modules et des interfaces sans passer par des fichiers spécifiques:

- `module Bla = struct [...] end` correspond à un fichier `bla.ml` qui contient `[...]`
- `module type BLA = sig [...] end` correspond à un fichier `bla.mli` qui contient `[...]`

Refaisons l'exemple précédent en **un seul fichier**.

```
(* la "signature" qui correspond à params.mli *)
module type PARAMS = sig
  val version : int
  val get_name : unit -> string
end

(* le module qui correspond à params.ml *)
module Params = struct
  let version = 3
  let get_name () = "hello"
  let secret = "..."
end

(* le test de compatibilité et le masquage *)
module Params_signed = (Params : PARAMS)

(* l'ancien fichier principal.ml *)
let () = Printf.printf "%s" (Params.get_name ())
let () = print_string Params.secret        (* <- ok! *)
let () = Print_string Params_signed.secret (* <- interdit! *)
```

1.16 Conclusion

- Compiler, c'est traduire un langage de programmation dans un autre de langage de programmation.
- Il y a beaucoup de sortes de langages de programmation!
- La compilation séparée, c'est pouvoir compiler des modules sans avoir à tout recompiler à chaque fois
- Pour que cela soit possible, il faut connaitre la signature d'un module, mais son implémentation n'est nécessaire qu'au moment de l'édition de liens
- L'édition de liens consiste à consolider tous les fichiers objets des modules et des librairies en un seul fichier.

1.17 La suite

Il y a beaucoup d'outils de développement très intéressants dans la suite `Ocaml`. Citons notamment:

- `ocamlprofile` qui permet de mesurer la consommation en temps et en espace d'un programme afin de l'optimiser
- l'interfaçage de C et OCaml: il est possible d'avoir un programme qui mélange du C et du Caml!
- le test aléatoire: `ocamlopt -afl-instrument` permet de compiler un programme pour qu'il soit utilisable par le générateur de test afl-fuzz. Plus besoin d'écrire vos tests, ils sont tirés au hasard "intelligemment"

1.17.1 La semaine prochaine : approfondissements sur les modules