Programmation Fonctionnelle
Cours 7
Compilation et Modules

Delia Kesner

Table de matières

Compilation

- Compilation : traduction du code source (écrit dans un langage source, ici OCaml) en code exécutable écrit dans un langage cible.
- Le compilateur génère du code exécutable dans un fichier, ce code peut être ensuite exécuté à partir d'une ligne de commande.
- Avantages de la compilation :
  - On obtient un exécutable autonome qu'on peut exécuter sans avoir besoin de tout l'environnement de programmation.
  - Le code exécutable sera plus efficace à exécuter.

Étapes de compilation (simplifié)

1. Analyse syntaxique : peut détecter une erreur de syntaxe.
2. Analyse sémantique : peut détecter une erreur de typage, filtrages non exhaustives, etc.
3. Génération de code intermédiaire.
4. Éventuellement optimisations diverses du code.
5. Production du code cible, peut nécessiter la liaison avec des bibliothèques.

Pour en savoir plus : cours complémentaires

Langages cibles

OCaml supporte deux langages cible différents :

- **Code-octet** (angl.: *byte-code*) : Code destiné à être exécuté sur une « machine virtuelle » : une espèce d'interpréteur destinée exclusivement à exécuter ce type de code, sans boucle d'interaction avec le programmeur (pareil que pour Java).
  `ocamlc`
- **Code native** : Code machine qui est spécifique au type de micro-processeur; le code est donc complètement autonome (pareil que pour C ou C++).
  `ocamlopt`

Avantages du code-octet

- Portabilité: Le code-octet peut être exécuté sur n'importe quel ordinateur, pourvu que la machine virtuelle soit installée.
- Efficacité:
  - La compilation vers code-octet produit un fichier plus petit que la compilation vers code natif.
  - Gain de vitesse par le fait que le code-octet est chargé plus rapidement en mémoire (car plus petit).
  - Le compilateur vers code-octet est plus rapide que le compilateur vers code natif.

Avantages du code natif

- Les compilateurs vers code natif n'existent que pour quelques types de micro-processeur (Intel 32bit, Intel 64bit, ...).
- L'exécution du code natif est plus rapide que celle du code-octet. Avantage surtout pour des applications très intensives en calcul.
- Le fichier exécutable est entièrement autonome.

Conclusion : normalement on préfère la compilation vers code-octet, sauf quand on a vraiment besoin d'un plus de vitesse.

Comment utiliser le compilateur code-octet

- Écrire le code source dans un fichier dont le nom se termine par `ml`, par exemple `prog.ml`.
- Exécuter (ligne de commande) `ocamlc prog`.
- S'il n'y a pas d'erreur, le code exécutable est mis dans le fichier `a.out`.
- Exécuter, en lançant (ligne de commande) `./a.out`.
- On peut spécifier directement le nom du fichier exécutable, par exemple : `ocamlc -o prog prog.ml` pour obtenir un exécutable du nom `prog`.
- Cette dernière manière de compiler produit également des fichiers `prog.cmi` et `prog.cmo` (voir plus tard pour des explications).

Comment utiliser le compilateur vers code natif

- Exécuter (ligne de commande) `ocamlopt prog`
- Plein d'options : voir le manuel OCaml.

Interaction avec un programme compilé

- Il n'y a pas de fonction main, mais une liste de phrases OCaml qui sont évaluées en ordre séquentiel.
- L'interaction avec un programme compilé se fait à travers des canaux d'entrée/sortie, de l'interface graphique, d'une connexion réseau (pas de boucle d'interprétation).
- Par conséquent, un programme compilé doit prévoir du code pour lire explicitement les entrées, les traiter et ensuite écrire les résultats. Pour cela il faut se servir des canaux d'entrée/sortie.

Accéder aux options et arguments d'un programme

Voici quelques exemples typiques:

- des options : `ls -l -r`
- des arguments : `lpr fichier1 fichier2`
- des options avec des arguments: `make -f mymakefile`

Comment faire en OCaml qu'un programme compilé puisse « voir » ses options ?

Accéder directement à la ligne de commande

- Module `Sys` : Interface système, mais indépendant du système d'exploitation (ne va donc pas très loin).
- Valeur `Sys.argv` du type `string array` : tableau qui contient les éléments de la ligne de commande
- `Array.length Sys.argv` : nombre d'éléments dans la ligne de commande (y compris le nom de la commande).
- `Sys.argv.(0)` : le nom de la commande à la position 0 du tableau
- `Sys.argv.(i)` : les arguments de la commande à la position i ($i > 0$) du tableau
- Pour en savoir plus : voir les Modules `Sys` et `Array`.

Les arguments d'une ligne de commande

- File argsimple.ml:

```
let nbargs = Array.length Sys.argv;;

for i=0 to nbargs-1 do (* attention, decalage par 1 ! *)
  Printf.printf "Argument␣numero␣\%d␣␣:␣\%s␣\n" i Sys.argv.(i);
done;;
```

- Compilation du programme: `ocamlc -o argsimple argsimple.ml`
- Exécution du programme: `./argsimple arg1 arg2 arg3`
- Résultat:

```
Argument numero 0:  ./argsimple
Argument numero 1:  arg1
Argument numero 2:  arg2
Argument numero 3:  arg3
```

Décomposition en modules

- La structure logique d'un programme peut induire un découpage en plusieurs fichiers, où chaque fichier correspond à un module
- Un *module* est une unité de programme qui regroupe des définitions.
- Avantages:
  - Code plus lisible
  - L'écriture et la compilation des modules deviennent indépendantes
  - Un même module peut être utilisé par plusieurs programmes différents

Modules comme unités de compilation

- Relation entre modules : *graphe de dépendance* (acyclique).
- Le module A dépend du module B ssi A utilise un nom défini par B.
- Le module B *exporte* des informations (par ex. une fonction), le module A les *importe*.

Modules et compilation séparée

- Si module A dépend du module B, alors la compilation de A a besoin de connaître les définitions effectuées par B : il faut donc compiler B avant la compilation de A.
- La compilation est séparée module par module et doit suivre l'ordre de dépendance.

```
ocamlc -c B.ml
ocamlc -c A.ml
```

- À la fin, il y a un assemblage des morceaux de code et une résolution des symboles (en angl.: *linking*): `ocamlc -o prog A.cmo B.cmo`

Les Interfaces en OCaml

Une interface (fichier `.mli`) peut contenir les éléments suivants :

- Des déclarations d'exceptions. Exemple :
  `exception Mon_exception`
- Des déclaration d'identificateurs avec leur type. Exemples :
  `val x: int`
  `val f: int -> int`
- Des définitions de types *concrets*. Exemple :
  `type environnement = (string*int) list`
- Des définitions de types *abstraits*. Exemple :
  `type environnement`
- Et bien sûr des *commentaires !*

Générer une Interface

- Éxecuter la commande `ocaml -i A.ml` pour générer l'interface `A.mli`
- Avantage: Si A dépend de B, alors on fixe d'abord l'interface de B. Puis, on peut écrire (et même compiler) les corps de A et B indépendamment.

Génération de l'interface d'un fichier I

```
exception Argument_negatif;;
# exception Argument_negatif

let rec fact_nn n =
  assert (n >=0);
  if n <=1 then 1 else n*fact_nn(n-1);;
# val fact_nn : int -> int = <fun>

let fact n =
  if n<0
  then raise Argument_negatif
  else fact_nn n;;
# val fact : int -> int = <fun>
```

Interface: example

```
exception Argument_negatif
val fact_nn : int -> int
val fact : int -> int
```

Importation dans les modules de OCaml

Deux possibilités:

1. Directive `open B` au début du module A : fait accessible en `A` toutes les définitions exportées par `B`.
   Avantage: plus court.
2. Préfixer par le nom du module: `B.f` dénote l'identificateur f exportée par le module B.
   Avantage: plus explicite, et pas d'ambiguïté (plusieurs modules peuvent exporter le même identificateur).

Compilation des modules en OCaml

- `ocamlc module.mli` produit `module.cmi`
- `ocamlc -c module` produit `module.cmo`
- `ocamlc -o program module_1.cmo module_2.cmo ... module_n.cmo` fait l'édition des liens et crée l'exécutable `program`.
  Il ne faut pas que `module_i` dépend de `module_j` pour $i < j$.

C'est quoi une mauvaise interface ?

- Pas de *commentaires*
  - L'interface est un contrat entre le programmeur et son utilisateur
    - Elle doit contenir toutes les informations nécessaires pour bien utiliser le module
    - Le codage d'une module peut être changé sans que cette modification ne soit visible dans l'interface
- Pas d'*encapsulation*
  - L'interface d'un module devrait être plus abstraite que son corps entier (cacher les outils auxiliares)
  - Le corps peut contenir des fonctions, types, exceptions privés.

Exemple

- Un module qui construit des tours d'entier (des piles d'entiers), où la valeur décroit vers le sommet (comme pour les Tours de Hanoï).
- On veut permettre seulement la construction des tours qui satisfont l'invariant « les valeurs décroissent vers le sommet ».
- Solution : Déclarer un type abstrait, et n'autoriser la modification d'une tour que par une fonction exportée par le module.

Interface de la pile: tour.mli

```
(* interface of the module for towers of integers. A tower is a stack of
   integers which are strictly decreasing from bottom to top. *)
type tower
exception Operation_illegal
  (* the empty tower *)
val empty: tower
(* (push i t) returns a new tower consisting of t with additionally i at
   the top, provided that result is still a tower. Otherwise raise
    Operation_illegal .*)
val push: int -> tower -> tower
(* (pop t) returns a tower which consists of t without its top element.
   raises Operation_illegal when t is empty. *)
val pop: tower -> tower
(* (top t) returns the top element of t, raises Operation_illegal when
   t is empty. *)
val top: tower -> int
```

Implémentation de la pile: tour.ml I

```
(* implementation of module tour *)
type tower = int list;;
# type tower = int list
exception Operation_illegal;;
# exception Operation_illegal

let empty = [];;
# val empty : 'a list = []
let top = function
    h::r -> h
  | [] -> raise Operation_illegal;;
# val top : 'a list -> 'a = <fun>
```

Implémentation de la pile: `tour.ml` II

```
(* (can_be_pushed i t) is true iff i can be pushed on t *)
let can_be_pushed i = function
    [] -> true
  | h::r -> i<h;;
# val can_be_pushed : 'a -> 'a list -> bool = <fun>
let push i t =
  if can_be_pushed i t then i::t else raise Operation_illegal;;
# val push : 'a -> 'a list -> 'a list = <fun>
let pop = function
    h::r -> r
  | [] -> raise Operation_illegal;;
# val pop : 'a list -> 'a list = <fun>
```

L'utilisation du module: `main.ml`

```
let a = Tour.empty;;
let b = Tour.pop (Tour.push 17 (Tour.push 42 a));;
print_int (Tour.top b);;
print_newline ();;
```

L'utilisation alternative du module

```
open Tour;;

let a = empty;;
let b = pop (push 17 (push 42 a));;
print_int (top b);;
print_newline ();;
```

Ordre de compilation sur l'exemple

`ocamlc tour.mli`	compile l'interface `tour.mli`
`ocamlc -c main.ml`	compile le corps `main.ml`
`ocamlc -c tour.ml`	compile le corps `tour.ml`
`ocamlc -o main tour.cmo main.cmo`	édite les liens

Cas particuliers

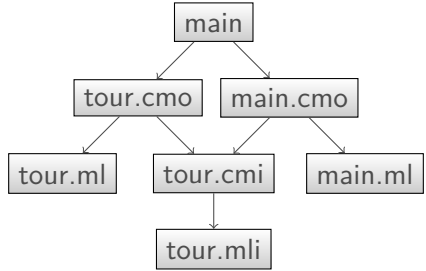
- Attention, l'utilisation de certains modules de la bibliothèques nécessitent des options supplémentaires lors de l'édition de liens.
- C'est en particulier le cas pour les modules `Graphics`, `Str`, `Unix`.
- Voir le manuel pour connaître les options nécessaires.
- Dans le cas du module `Graphics` :

```
ocamlc graphics.cma module_1.cmo ... module_n.cmo
```

Comment automatiser le processus de compilation

- Problème : quand on change le code source d'un module il faut recompiler (dans le bon ordre) ce module et les modules qui en dépendent, et ensuite refaire l'exécutable.
- Première solution : un *script shell* (un programme) qui compile tout.
- Inconvénient : Solution ad-hoc, pas de ré-compilation partielle.
- Meilleure solution : `make`
  - Outil universel (C, C++, Java, LaTeX) de développement sous UNIX
  - Principe simple

Dépendances dans la création des fichiers



- Avant de créer un fichier il faut s'assurer que toutes ses dépendances existent.
- Le graphe doit être acyclique.

Compilation automatique avec *make*

Principe de base:

- Règles qui décrivent des dépendances entre des *cibles*
- Une cible est un nom de fichier, ou un nom symbolique
- Avec chaque règle : instructions (commandes shell) pour mettre à jour une cible.

Le Makefile sur l'exemple I

```
maindirect: tour.ml tour.mli main.ml
	ocamlc -o maindirect tour.mli tour.ml main.ml

.PHONY: clean

clean:
	rm -f maindirect tour.cmi tour.cmo main.cmi main.cmo
```

Le Makefile sur l'exemple II

```
tour.cmi: tour.mli
	ocamlc -c tour.mli

tour.cmo: tour.cmi tour.ml
	ocamlc -c tour.ml

main.cmo: tour.cmi main.ml
	ocamlc -c main.ml

main: tour.cmo main.cmo
	ocamlc -o main tour.cmo main.cmo

.PHONY: clean

clean:
	rm -f main tour.cmi tour.cmo main.cmi main.cmo
```

Utilisation du Makefile

- Écrire un fichier `Makefile`
- Éxecuter la commande `make`
- Quand `make` est lancé après la modification d'un fichier : il y a seulement la re-génération des cibles qui ne sont plus à jour.

Une solution alternative avec `ocamlbuild`

- Dans la distribution standard de *OCaml* il existe un outil specialisé `ocamlbuild`.
- Cet outil gère seul toute la compilation dans des cas simples, et il est très configurable pour des cas complexes.

Comment trouver le bon découpage en modules ?

- Analyse descendante: du plus général vers le plus particulier.
- Commencer avec le programme entier : quelle est l'entrée, quelle est le résultat ?
- Puis, couper le fonctionnement du programme en sous tâches.
  - identifier les fonctionnalités principales de la sous tâche ($\Rightarrow$ fonctions exportées)
  - les types de données
  - les fonctionnalités partagées par les sous tâches ($\Rightarrow$ modules partagés)
  - dessiner le graphe de dépendance entre les modules.
- Il est utile d'avoir une idée grossière de l'implémentation des modules.

Remarques

- Formes caricaturales à éviter:
  - Un seul module pour le programme entier
  - Un module par définition de type ou fonction
  - Découpage arbitraire: un module pour tous les types, un autre pour toutes les fonctions, etc.
- Bon découpage:
  - Correspond à la logique du programme
  - Modules d'une taille raisonnable
  - Définition auxiliaires cachées dans les modules (l'organisation en modules simplifie la structure du programme)
  - Réutilisation du code au lieu de duplication

Quelques mots sur la documentation

- Documenter la structure globale du programme (graphe de dépendance)
- Interfaces des modules: Documenter l'*utilisation* du module :
  - Son rôle général
  - Que représentent les types ?
  - Spécifier les fonctions : expliquer les rôles des arguments, les hypothèses sur leurs valeurs (par ex: entier positif, liste triée, etc.), et bien sûr le résultat. N'oubliez pas les cas d'erreur.
- En général: La doc de l'interface doit contenir toutes les informations nécessaires pour l'utilisation du module.
- Dans les corps des modules :
  - Spécifier les fonctions privées.
  - Expliquer l'algorithme utilisée quand il n'est pas évident.
  - Donner des invariants des fonctions (utiliser des construction du langage comme *assertion*)

Le langage des modules en OCaml

En vérité : Un module n'est pas toujours une unité de compilation.
Il y a en OCaml des constructions pour définir des modules (à part des modules définis implicitement par unité de compilation). Cela permet par exemple de :

- définir un module avec plusieurs interfaces
- définir des modules qui sont *paramétrés par des modules* (par exemples des tables de hachage)

```
module Nom = struct
   <définitions des types, valeurs, et exceptions>
end
```

Pour en savoir plus : Cours *Programmation Fonctionnelle Avancée* du M1