UJF **POLYTECH**
RICM1 **Année 2006-07**

Langages et programmation 2

Précis de compilation CAML

1 La compilation avec `ocamlc`

Le tableau suivant récapitule l'ensemble des différents fichiers mis en jeux lors de la compilation de programmes CAML. Les fichiers sources ont été séparés de ceux produits par la compilation.

`.ml`	code source d'un ensemble de définitions considérées dans l'ordre de leur déclaration
`.mli`	description du type de toutes les définitions du `.ml` correspondant que l'on souhaite rendre *visible* de l'extérieur (fichier d'interface)
`.cmo`	*bytecode* résultant de la compilation d'un `.ml`
`.cmi`	version compilée d'un fichier d'interface `.mli`
`.cma`	collection de `.cmo` (librairie)

Reste l'exécutable final (que l'on notera `xxx`) qui est produit à partir d'un ensemble de `.cmo` et de `.cma`. Cet exécutable est en fait du *bytecode* qu'interprète la machine virtuelle `ocamlrun` ; ce n'est donc pas un véritable *standalone executable.*

1.1 La commande `ocamlc`

Le compilateur `ocamlc` s'utilise de la façon suivante. On compile un fichier source `f.ml` par la commande :

`ocamlc -c f.ml` *effectue* `.ml` $\longrightarrow$ `.cmo`

Lors de cette compilation il est vérifié que le type *inféré* d'une définition est bien en accord avec celui déclaré dans le fichier d'interface.

Si dans le fichier source on souhaite utiliser une définition rendue visible par un autre `.mli`, il faut indiquer au compilateur où elle est spécifiée (*e.g.* `autre.mli`). Deux choix possibles : soit utiliser la directive `open Autre` au début du fichier source, soit *qualifier* toute utilisation de la définition par le préfixe « `Autre.` » (ce qui peut s'avérer meilleur pour la lisibilité du programme).

On compile de la même manière un fichier d'interface :

`ocamlc -c f.mli` *effectue* `.mli` $\longrightarrow$ `.cmi`

Pour gagner du temps dans l'écriture de vos fichiers d'interface vous pouvez générer automatiquement le `.mli` correspondant à l'ensemble d'un fichier source ainsi :

`ocamlc -i f.ml > f.mli` *effectue* `.ml` ⟶ `.mli`

La création d'une librairie s'effectue simplement en donnant l'ensemble des `.cmo` qui la compose.

`ocamlc -a f1.cmo f2.cmo f3.cmo -o lib.cma` *effectue* `.cmo` ⟶ `.cma`

Enfin on crée l'exécutable par :

`ocamlc f4.cmo f5.cmo lib.cma f6.cmo -o xxx` *effectue* `.cma + .cmo` ⟶ `xxx`

L'ordre dans lequel on liste les `.cmo` a une importance. Si un certain `.cmo` utilise une définition introduite dans un autre `.cmo`, ce dernier doit figurer avant. Les `.cmi` interviennent donc ici. On notera d'ailleurs que si vous n'avez pas fourni de `.mli` pour un certain fichier source, le compilateur crée automatiquement un `.cmi` rendant visible toutes les définitions décrites dans ce dernier.

On peut s'interroger sur la réelle valeur ajoutée des librairies, qui ne sont qu'une concaténation de `.cmo`. Elle réside dans l'inclusion des seules définitions réellement utilisées dans l'exécutable final.

1.2 Compilation directe

La chaîne de compilation décrite précédemment permet la compilation séparée des différentes parties qui composent l'exécutable final. Hors de ce cadre, on peut directement crée l'exécutable à partir des sources comme suit :

`ocamlc f1.ml f2.ml f3.ml -o xxx` *effectue* `.ml` ⟶ `xxx`

2 La compilation avec `ocamlopt`

Le compilateur `ocamlopt` permet la production d'un exécutable plus efficace, qui est un véritable *standalone executable*.

Pour ce qui est des fichiers générés, les `.cmo` prennent l'extension `.cmx` et les `.cma` celle de `.cmxa`. Enfin un fichier `.o` (fichier objet compatible avec les compilateurs C) est crée avec tout `.cmx`.

2.1 Différences avec `ocamlc`

Chacun des deux compilateurs a sa raison d'être.
- l'efficacité du code produit est l'affaire d'`ocamlopt`.
- la rapidité de la compilation revient à `ocamlc`.
- la portabilité du code c'est le principe d'`ocamlc`.

L'utilisation d'`ocamldebug` n'est possible que sur les exécutables compilés avec `ocamlc` et l'option de déboguage `-g`.

3 OCaml et l'utilitaire `make`

On donne ici un exemple de `Makefile` standard pour un projet OCaml. Il permet de construire :
- `prog1` un fichier *bytecode* exécutable composé des trois unités `mod1`, `mod2` et `mod3`.
- et `prog2` un fichier *standalone executable* composé des unités `mod4` et `mod5`.

```
OCAMLC=ocamlc
OCAMLOPT=ocamlopt
OCAMLDEP=ocamldep

INCLUDES=                        # all relevant -I options here
OCAMLFLAGS=$(INCLUDES)           # add other opions for ocamlc here
                                 # (e.g. -g for ocamldebug)
OCAMLOPTFLAGS=$(INCLUDES)        # add other opions for ocamlopt here

PROG1_OBJS=mod1.cmo mod2.cmo mod3.cmo

prog1: $(PROG1_OBJS)
	$(OCAMLC) -o prog1 $(OCAMLFLAGS) $(PROG1_OBJS)

PROG2_OBJS=mod4.cmx mod5.cmx

prog2: $(PROG2_OBJS)
	$(OCAMLOPT) -o prog2 $(OCAMLOPTFLAGS) $(PROG2_OBJS)

# common rules

.SUFFIXES: .ml .mli .cmo .cmi .cmx

.ml.cmo:
	$(OCAMLC) $(OCAMLFLAGS) -c $<

.mli.cmi:
	$(OCAMLC) $(OCAMLFLAGS) -c $<

.ml.cmx:
	$(OCAMLOPT) $(OCAMLOPTFLAGS) -c $<

# clean up

clean:
	rm -f prog1 prog2
	rm -f *.cm[iox]

# dependencies

depend:
	$(OCAMLDEP) $(INCLUDES) *.mli *.ml > .depend

include .depend
```