Modules

Compilateur(s) OCaml

- jusqu'ici nous avons évalué des expressions dans le *toplevel* OCaml

  interprétation des termes

- OCaml propose aussi un compilateur, ou plutôt deux
  - les exécutables doivent faire des effets de bord, sinon ils ont peu d'intérêt
  - typiquement, `main.ml` se termine par `let _ = main ()`
- qui dit compilation dit compilation séparée
  - retour à la notion de *module*

Compilation séparée

- comment?

  un fichier `.ml` contient des définitions OCaml *(plus besoin de ;;)*

  `ocamlc -c toto.ml` ⇝ `toto.cmo`, fichier *objet*

- on fait référence à l'objet en en capitalisant le nom:
  - à toplevel, `#load` pour charger le fichier objet

    ```
    # #load "toto.cmo";;
    Toto.f;;
    - :  int -> int = <fun>
    ```
  - pour compiler, et engendrer un exécutable:
    - dans `titi.ml`, j'utilise des fonctions définies dans `toto.ml`, p.ex. `Toto.f 51`
    - `ocamlc toto.cmo titi.ml -o prog`

      compile et fait les liens, `prog` fichier exécutable

Remarques sur la compilation séparée

- lorsque l'on produit l'exécutable (`ocamlc ...  -o ..`),
  il faut passer les `.cmo` *dans l'ordre* des dépendances

(pas de dépendance croisée – ∃ outils pour mettre les `.cmo` dans l'ordre)

- on peut faire

```
ocamlc -c toto.ml

ocamlc toto.cmo titi.ml -o prog
```

ou bien

```
ocamlc -c toto.ml                    (option -c: compiler seulement)
ocamlc -c titi.ml

ocamlc toto.cmo titi.cmo -o prog
```

- deux manières d'accéder aux valeurs:
  - ▷ `let a = Toto.f x in ...`
  - ▷ `open Toto` en début de fichier

    ⇝ `let a = f x in ...`

Modularité: remarques

- le découpage en fichiers se plaque sur la notion de module

  un fichier ↔ un module
- en particulier, les modules structurent l'espace des noms:
  `Heap.iter` et `Stack.iter`

  NB: idem pour les exceptions: `try Toto.f x with | Toto.Raté -> ...`

  et les constructeurs
- signatures (interfaces): fichiers `.mli`
  - si pas de fichier `.mli`,
    `ocamlc -c toto.ml` ⇝ `toto.cmi` *et* `toto.cmo`
  - si fichier `.mli`,

    `ocamlc -c toto.mli`

    puis `ocamlc -c toto.ml`: vérification

Interfaces: exemples

- piles en OCaml
  - interface: `stack.mli` Démo
  - implémentation: `stack.ml` Démo
- utilisation des *types abstraits*: `<abstr>`

('pudeur', comme dans `<fun>`)

```
# let mapile = Stack.create();;
mapile :  int Stack.t = <abstr>
# Stack.push 32 mapile;;
- :  unit = ()
```