Programmation Fonctionnelle
Cours 9

Michele Pagani

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

pagani@pps.univ-paris-diderot.fr

21 novembre 2016

Modules
ou Structures

Modules ou Structures

```
module NomModule =
struct
    .....
    .....
    .....
end
```

- on peut regrouper des ensembles de définitions dans des structures ou modules et leur donner un nom
  - définition des valeurs et des fonctions
    let x =
  - définition des types
    type t =
  - définitions des exceptions
    exception E [of ...]
  - ...
- decoupage en “morceaux” correspondant à la logique interne du projet

Modules ou Structures

```
module NomModule =
struct
    .....
    .....
    .....
end
```

- fichiers, packages, . . . modules !

  C / C++ : on utilise les fichiers avec leur interfaces `.h`

  Java: les packages organisent les classes dans un espace de nommage hiérarchique

  OCaml: les système des modules d'OCaml est l'un des plus puissants et aboutis.

Exemple: le module Counter

```
module Counter =
  struct
    let c = ref 0
    let incr () = c:= !c + 1
    let show () = !c
  end ;;
```

Le top-level réponds avec la signature (type ou interface du module):

```
module Counter :
  sig
    val c : int ref
    val incr : unit -> unit
    val show : unit -> int
  end
```

Notions importantes:

- nom du module: doit commencer par un majuscule
- structure: code regroupé entre `struct` et `end`
- signature: l'interface, délimitée par `sig` et `end`

Exemple: le module Counter

```
module Counter =
  struct
    let c = ref 0
    let incr () = c:= !c + 1
    let show () = !c
  end ;;
```

Le top-level réponds avec la signature (type ou interface du module):

```
module Counter :
  sig
    val c : int ref
    val incr : unit -> unit
    val show : unit -> int
  end
```

Notions importantes:

- nom du module: doit commencer par un majuscule
- structure: code regroupé entre `struct` et `end`
- signature: l'interface, délimitée par `sig` et `end`

Accès à un composant d'un module

```
# (*les definitions ne sont pas accessibles directement*)
  show ();;
Characters 0-4:
  show ();;
  ^^^^
Error: Unbound value show

# (*il faut utiliser la notation pointee*)
  Counter.show ();;
- : int = 0

# (*sinon rendre visible les definitions avec open*)
  open Counter;;

# show ();;
- : int = 0
```

Définition et annotation de signature

- Une signature peut être inférée automatiquement,
- mais elle peut aussi être définie explicitement:

```
# module type CounterFullItf =
      sig
        val c : int ref
        val incr : unit -> unit
        val show : unit -> int
      end;;

module type CounterFullItf =
 sig val c: int ref val incr: unit->unit val show: unit->int end
```

- et utilisée pour déclarer l'interface d'un module

```
# module Counter : CounterFullItf =
      struct
        let c = ref 0
        let incr () = c := !c + 1
        let show () = !c
     end;;

 module Counter : CounterFullItf
```

Une autre syntaxe pour l'annotation

- On peut aussi écrire:

```
# module Counter =
      (struct
        let c = ref 0
        let incr () = c := !c + 1
        let show () = !c
      end : CounterFullItf );;

module Counter : CounterFullItf
```

N.B. Le parenthèses sont obligatoires : même notation que les annotations de type des valeurs comme:

```
# let f x = (x : int);;
val f : int -> int = <fun>
```

(⇒ voir plus tard)

Encapsulation

Cacher des parties d'un module

- La signature inférée automatiquement contient tous les détails de la structure qui l'implémente
- Dans le cas de notre `Compteur`, on peut voir la variable `c`, et l'altérer! Cela peut très bien arriver involontairement, si vous avez une autre variable `c` dans le programme.

```
# open Counter;;
# incr ();;
- : unit = ()

# (* ce qui suit ne devrait pas etre permis*)
  c := !c + 32;;
- : unit = ()

# show ();;
- : int = 34
```

- On a besoin d'empêcher cet accès aux détails d'implémentation:

$\Rightarrow$ mécanisme d'encapsulation.

Utiliser une signature pour l'encapsulation

- l'interface d'un module peut être plus restreinte que son corps
  $\Rightarrow$ on cache des fonctions, types, exceptions auxiliaires
- Dans notre exemple:

```
# module type CounterItf =
  sig
    val incr : unit -> unit
    val show : unit -> int
  end;;

module type CounterItf =
  sig val incr : unit -> unit val show : unit -> int end
```

- pour cacher une partie du code d'un module existant:

```
# module CounterHide = (Counter : CounterItf);;
module CounterHide : CounterItf
# CounterHide.show ();;
- : int = 0
# CounterHide.c;;
Characters 0-13:
  CounterHide.c;;
  ^^^^^^^^^^^^^
Error: Unbound value CounterHide.c
```

Types abstraits

Le principe d'abstraction

```
module MultiCounter =
 struct
   type counter = int ref
   let create () = ref 0
   let incr c = c:= !c+1
   let show c = !c
 end
```

- Comment pouvons nous restreindre l'accès aux données de type `counter` dans l'exemple ci-dessous ?

- avec les types abstraits:

  ```
  type counter
  ```

  - l'interface contient seulement la déclaration `type counter`
  - la définition du type est donné dans le corps du module
  - ainsi, la seule façon de manipuler une valeur de type `counter` à l'extérieur du module est d'appeler les fonctions du module

Le principe d'abstraction

```
module MultiCounter =
 struct
   type counter = int ref
   let create () = ref 0
   let incr c = c:= !c+1
   let show c = !c
 end
```

- Comment pouvons nous restreindre l'accès aux données de type `counter` dans l'exemple ci-dessous ?

- avec les types abstraits:

  ```
  type counter
  ```

  - l'interface contient seulement la déclaration `type counter`
  - la définition du type est donné dans le corps du module
  - ainsi, la seule façon de manipuler une valeur de type `counter` à l'extérieur du module est d'appeler les fonctions du module

Types abstraits: exemple (I)

```
module MultiCounter =
 struct
   type counter = int ref
   let create () = ref 0
   let incr c = c:= !c+1
   let show c = !c
 end

module type MultiCounterInt =
  sig
    type counter
    val create : unit -> counter
    val incr : counter -> unit
    val show : counter -> int
  end

module MultiCounterHide = (MultiCounter : MultiCounterInt)
```

Types abstraits: exemple (II)

```
# let c = MultiCounter.create () ;;
val c : int ref = {contents = 0}

# (*ce qui suit devrait etre interdit*)
c := 34;;
- : unit = ()

# let d = MultiCounterHide.create () ;;
val d : MultiCounterHide.counter = <abstr>

# d := 34;;
Characters 0-1:
  d := 34;;
  ^
Error: This expression has type MultiCounterHide.counter
       but an expression was expected of type 'a ref
```

Exercice: le module `Stack` du TP6

Définir module et interface de:

Une pile LIFO

Le but de cette section est de mettre en œuvre la structure de données des piles (*stack* an anglais) regroupant leurs éléments dans une liste suivant le principe du "dernier arrivé, premier sorti", (*Last In, First Out* en anglais). La différence par rapport aux listes définissables en fonctionnel pur est que les primitives d'une pile peuvent modifier son état pendant leur execution.

Par example, si `s` est une pile non vide, un appel de la forme `pop s` renverra l'élément en tête de `s` (comme `List.hd` appliqué à une liste non vide), mais en plus, cet appel modifiera `s` en retirant cet élément de la pile.

Exercice 1. Définir le type `'a stack` comme le type des réferences vers `'a list`. Écrire une fonction `create : unit -> 'a stack` renvoyant une nouvelle pile vide. Écrire une fonction `empty : 'a stack -> boolean` renvoyant `true` si et seulement si son argument est une pile vide.

Exercice 2. Écrire une fonction `pop : 'a stack -> 'a` renvoyant, s'il existe, l'élément en tête d'une pile, et retirant cet élément de la pile. Par example, si `s` vaut `{contents = [4; 3; 2]}`, `pop s` renverra `4` et modifiera `s` en `{contents = [3; 2]}`. Que doit faire `pop` si la pile est vide ?

Exercice 3. Écrire une fonction `peek : 'a stack -> 'a` renvoyant, comme `pop`, le premier élément d'une pile, mais sans retirer cet élément de la pile.

Exercice 4. Écrire une fonction `push : 'a -> 'a stack -> unit` permettant d'ajouter un élément en tête d'une pile. Par example, si `s` vaut `contents = [4; 3; 2]`, `push 1 s` renverra `()` et modifiera la valeur de `s` en `{contents = [1; 4; 3; 2]}`.

Exercice 5. Écrire une fonction `is_equal: 'a stack -> 'a stack -> bool` renvoyant `true` si et seulement les deux piles passées en arguments contiennent la même suite d'éléments. Cette fonction peut s'écrire en moins de 50 caractères, espaces compris.

Exercice 6. Écrire une fonction `clone : 'a stack -> 'a stack` renvoyant une copie de la pile passée en argument. Attention, `clone s` doit créer une *nouvelle* pile contenant les mêmes éléments que `s`, et non partager la référence `s`.

Les foncteurs
ou modules paramétriques

Modules ou Structures

```
module NomFoncteur =
  functor (S : signature) ->
    struct
      .....
      .....
      .....
    end
```

```
module NomFoncteur (S : signature) =
    struct
      .....
      .....
      .....
    end
```

- un foncteur est une fonction des modules dans les modules
- le corps du foncteur dépends du module S en entrée

```
module F (S:  signature) =
    struct
      type u = S.t * int
      let y = S.g(S.x)
    end
```

- pas accès aux champs functeur, il faut l'appliquer à une implémentation de la signature S:

```
module Module1 = F (S1)
module Module2 = F (S2)
```

- Module1 et Module2 sont des modules standards

```
Module1.y * Module2.y
```

Exemple: (une version de) Set

```
1   module type OrderedType =
2     sig
3       type t
4       val compare : t -> t-> int
5     end;;
6
7   module Set =
8     functor (Elt: OrderedType) ->
9     struct
10      type element = Elt.t
11      type set = element list
12      let empty = []
13      let rec add x s =
14        match s with
15          [] -> [x]
16        | hd::tl ->
17           if (Elt.compare x hd = 0) then s (*x belongs to s*)
18           else if  (Elt.compare x hd < 0) then x::s (*x smaller all elt in s*)
19           else hd :: (add x tl)
20      let rec member x s =
21        match s with
22          [] -> false
23        | hd::tl ->
24           if (Elt.compare x hd = 0) then true (*x belongs to s*)
25           else if  (Elt.compare x hd < 0) then false (*x smaller all elt in s*)
26           else member x tl
27    end
```

Compilation

Compilation

traduction de code source (écrit dans un langage source, ici OCaml) en code exécutable écrit dans un langage cible.

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 d'un 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 *Machines virtuelles* (L3 2nd semestre),
cours *Interpretation programmes* et *Compilation* du M1

Compilation

traduction de code source (écrit dans un langage source, ici OCaml) en code exécutable écrit dans un langage cible.

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 d'un 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 *Machines virtuelles* (L3 2nd semestre),
cours *Interpretation programmes* et *Compilation* du M1

Compilation

traduction de code source (écrit dans un langage source, ici OCaml) en code exécutable écrit dans un langage cible.

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 d'un 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 *Machines virtuelles* (L3 2nd semestre),
cours *Interpretation programmes* et *Compilation* du M1

Compilation

traduction de code source (écrit dans un langage source, ici OCaml) en code exécutable écrit dans un langage cible.

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 d'un 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 *Machines virtuelles* (L3 2nd semestre),
cours *Interpretation programmes* et *Compilation* du M1

Compilation

traduction de code source (écrit dans un langage source, ici OCaml) en code exécutable écrit dans un langage cible.

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 d'un 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 *Machines virtuelles* (L3 2nd semestre),
cours *Interpretation programmes* et *Compilation* du M1

Compilation

traduction de code source (écrit dans un langage source, ici OCaml) en code exécutable écrit dans un langage cible.

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 d'un 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 *Machines virtuelles* (L3 2nd semestre),
cours *Interpretation programmes* et *Compilation* du M1

Compilation

traduction de code source (écrit dans un langage source, ici OCaml) en code exécutable écrit dans un langage cible.

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 d'un 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 *Machines virtuelles* (L3 2nd semestre),
cours *Interpretation programmes* et *Compilation* du M1

Langages cibles

OCaml supporte deux langages cible différents :

- Code-octet (angl.: *byte-code*) : `ocamlc`
  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).
- Code native : `ocamlopt`
  code machine qui est spécifique au type de micro-processeur; le code est donc complètement autonome.
  (Pareil que pour C ou C++).

Langages cibles

OCaml supporte deux langages cible différents :

- Code-octet (angl.: *byte-code*) : `ocamlc`
  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).

- Code native : `ocamlopt`
  code machine qui est spécifique au type de micro-processeur; le code est donc complètement autonome.
  (Pareil que pour C ou C++).

Langages cibles

OCaml supporte deux langages cible différents :

- Code-octet (angl.: *byte-code*) : `ocamlc`
  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).
- Code native : `ocamlopt`
  code machine qui est spécifique au type de micro-processeur; le code est donc complètement autonome.
  (Pareil que pour C ou C++).

Interaction avec un programme compilé

- Il n'y a pas de fonction `main` (au contraire qu'en `C`, ou `Java`)
  - un programme de OCaml est une liste de phrases qui sont évaluées en ordre séquentiel

- l'interaction avec un programme compilé est uniquement à travers les canaux d'entrée/sortie
  (standard input/output, fichiers, réseau,. . .)
  - on ne peut pas entrer des expressions à interpreter, comme dans la boucle d'interprétation,
  - on doit donc prévoir du code pour lire explicitement les entrées, les traiter et écrire le résultat

- on peut lire des options, en utilisant `Sys.argv`
  - tableau de `string` contenant les éléments de la commande:

    `Sys.argv.(0)` : la commande elle-même
    `Sys.argv.(1)` : premiere option
    `Sys.argv.(2)` : deuxième option
    ⋮

Interaction avec un programme compilé

- Il n'y a pas de fonction `main` (au contraire qu'en `C`, ou `Java`)
  - un programme de OCaml est une liste de phrases qui sont évaluées en ordre séquentiel

- l'interaction avec un programme compilé est uniquement à travers les canaux d'entrée/sortie
  (standard input/output, fichiers, réseau,. . .)
  - on ne peut pas entrer des expressions à interpreter, comme dans la boucle d'interprétation,
  - on doit donc prévoir du code pour lire explicitement les entrées, les traiter et écrire le résultat

- on peut lire des options, en utilisant `Sys.argv`
  - tableau de `string` contenant les éléments de la commande:
    `Sys.argv.(0)` : la commande elle-même
    `Sys.argv.(1)` : premiere option
    `Sys.argv.(2)` : deuxième option
    ⋮

Interaction avec un programme compilé

- Il n'y a pas de fonction `main` (au contraire qu'en `C`, ou `Java`)
  - un programme de OCaml est une liste de phrases qui sont évaluées en ordre séquentiel

- l'interaction avec un programme compilé est uniquement à travers les canaux d'entrée/sortie
  (standard input/output, fichiers, réseau,. . .)
  - on ne peut pas entrer des expressions à interpreter, comme dans la boucle d'interprétation,
  - on doit donc prévoir du code pour lire explicitement les entrées, les traiter et écrire le résultat

- on peut lire des options, en utilisant `Sys.argv`
  - tableau de `string` contenant les éléments de la commande:

    `Sys.argv.(0)` : la commande elle-même
    `Sys.argv.(1)` : premiere option
    `Sys.argv.(2)` : deuxième option
    ⋮

Structuration des programmes en unités de compilation

An old friend...

```
1  type joueur = Croix | Rond
2  type pion = P of joueur | Vide
3  type ligne = pion*pion*pion
4  type grille = ligne*ligne*ligne
5  exception MoveForbidden
6  exception NoMoreMove
7
8  (* Fonction pour changer un pion de la grille *)
9  let move (l,c) ((l1,l2,l3): grille) j : grille =
10  let move_ligne c (p1,p2,p3) j = match c with
11   | 'a' -> if (p1=Vide) then (P j, p2, p3) else raise  MoveForbidden
12   | 'b' -> if (p2=Vide) then (p1, P j, p3) else raise  MoveForbidden
13   | 'c' -> if (p3=Vide) then (p1, p2, P j) else raise  MoveForbidden
14   | _ -> raise  MoveForbidden
15  in match l with
16   | 1 -> ((move_ligne c l1 j),l2,l3)
17   | 2 -> (l1, (move_ligne c l2 j),l3)
18   | 3 -> (l1,l2, (move_ligne c l3 j))
19   | _ -> raise MoveForbidden
20
21 (* Fonction pour tester un Vide *)
22 let aVide (l1,l2,l3) =
23  let aVide_ligne (p1,p2,p3) = (p1=Vide) ||(p2=Vide) || (p3=Vide) in
24  aVide_ligne l1 || aVide_ligne l2 || aVide_ligne l3
25
26 (* Fonction qui verifie si un joueur a gagne *)
27 let gagne (((p1,p2,p3),(p4,p5,p6),(p7,p8,p9)):grille) =
28  if aVide ((p1,p2,p3),(p4,p5,p6),(p7,p8,p9)) then match p1 with
29   | P j when (p1=p2 && p2=p3) || (p1=p4 && p4=p7) || (p1=p5 && p5=p9) -> P j
30   | _ -> match p2 with
31    | P j when (p2=p5 && p5=p8) -> P j
32    | _ -> match p3 with
33     | P j when (p3=p6 && p6=p9) || (p3=p5&&p5=p7) -> P j
34     | _ -> match p4 with
35      | P j when p4=p5 && p5=p6 -> P j
36      | _ -> match p7 with
37       | P j when p7=p8 && p8=p9 -> P j
38       | _ -> Vide
39  else raise NoMoreMove
```

An old friend...

```
1   (* Fonction qui affiche un pion *)
2   let affichePion = function
3         P Croix -> print_char 'X'
4       | P Rond -> print_char 'O'
5       | Vide -> print_char ' '
6
7   (* Fonction qui affiche une ligne *)
8   let afficheLigne (p1, p2, p3) =
9       print_string "| "; affichePion p1 ;
10      print_string " | " ; affichePion p2 ;
11      print_string " | " ; affichePion p3 ;
12      print_string " |\n"
13
14  (* Fonction qui affiche la grille *)
15  let afficheGrille = fun (l1, l2, l3) ->
16      print_string "\n      a   b   c\n" ;
17      print_string "   +---+---+---+\n" ;
18      print_string " 1 " ; afficheLigne l1;
19      print_string "   +---+---+---+\n" ;
20      print_string " 2 " ; afficheLigne l2;
21      print_string "   +---+---+---+\n" ;
22      print_string " 3 " ; afficheLigne l3;
23      print_string "   +---+---+---+\n" ;;
```

An old friend...

```
1  (*entreer coordonees*)
2  let rec read_coordonees g j =
3   try
4    print_string "Ligne␣?␣";
5    let l=read_int () in
6    print_string "Colonne␣?␣";
7    let c=(read_line ()).[0] in
8    move (l,c) g j
9   with
10   | MoveForbidden ->
11   (print_endline "Choix␣interdite␣!" ; read_coordonees g j)
12
13 let tour g j =
14  afficheGrille g;
15  print_string "Bonjour␣"; affichePion (P j) ;print_newline ();
16  read_coordonees g j
17
18 (*negation joueur*)
19 let autre j = if j=Croix then Rond else Croix
20
21 (*fonction principale*)
22 let rec morpion g j=
23  try
24   let x = gagne g in
25   match x with
26   | P winner -> print_string "Bravo␣:␣"; affichePion (P winner) ; print_endline
27   | Vide -> let g_next = (tour g j) in morpion g_next (autre j)
28  with
29   | NoMoreMove -> print_endline ("Aucun␣gagnant,␣desole␣!")
30
31 (* grille initiale *)
32 let init = ((Vide,Vide,Vide),(Vide,Vide,Vide),(Vide,Vide,Vide));;
33
34 (*main*)
35 morpion init Croix;;
```

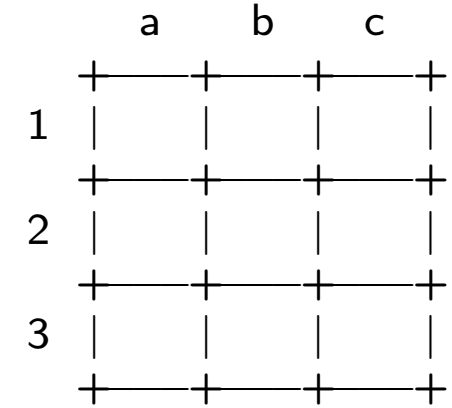
Écrire, compiler, exécuter

```
$ ls
morpion.ml

$ ocamlc -o morpion morpion.ml

$ ls
morpion
morpion.cmo
morpion.cmi
morpion.ml

$ ./morpion
```




```
Bonjour X
Ligne ?
```

Décomposition en modules

- On peut faire émerger la structure logique d'un programme en le découpant en plusieurs fichiers
  - chaque fichier corresponds à un module
    - interface: extension `.mli` (optionelle)
      résumé des définitions d'un module
    - implantation: extension `.ml`
      code réalisant les définitions

- par example, `morpion.ml` peut être découpé en:

  `jeu.ml` qui regroupe les fonctions définissant le jeu

  `affichage.ml` qui regroupe les fonctions qui s'occupent de l'affichage à l'écran

  `programme.ml` qui regroupe les fonctions interagissant avec l'utilisateur

- avantages :
  - code plus lisible
  - on peut rédiger (même compiler) les modules indépendamment
  - le même module peut être utilisé par plusieurs programmes

Décomposition en modules

- On peut faire émerger la structure logique d'un programme en le découpant en plusieurs fichiers
  - chaque fichier corresponds à un module
    - interface: extension `.mli` (optionelle)
      résumé des définitions d'un module
    - implantation: extension `.ml`
      code réalisant les définitions

- par example, `morpion.ml` peut être découpé en:

  `jeu.ml` qui regroupe les fonctions définissant le jeu

  `affichage.ml` qui regroupe les fonctions qui s'occupent de l'affichage à l'écran

  `programme.ml` qui regroupe les fonctions interagissant avec l'utilisateur

- avantages :
  - code plus lisible
  - on peut rédiger (même compiler) les modules indépendamment
  - le même module peut être utilisé par plusieurs programmes

Décomposition en modules

- On peut faire émerger la structure logique d'un programme en le découpant en plusieurs fichiers
  - chaque fichier corresponds à un module
    - interface: extension `.mli` (optionelle)
      résumé des définitions d'un module
    - implantation: extension `.ml`
      code réalisant les définitions

- par example, `morpion.ml` peut être découpé en:

  `jeu.ml` qui regroupe les fonctions définissant le jeu

  `affichage.ml` qui regroupe les fonctions qui s'occupent de l'affichage à l'écran

  `programme.ml` qui regroupe les fonctions interagissant avec l'utilisateur

- avantages :
  - code plus lisible
  - on peut rédiger (même compiler) les modules indépendamment
  - le même module peut être utilisé par plusieurs programmes

Implantation jeu.ml

```
type joueur = Croix | Rond
type pion = P of joueur | Vide
type ligne = pion*pion*pion
type grille = ligne*ligne*ligne
exception MoveForbidden
exception NoMoreMove

(* Fonction pour changer un pion de la grille *)
let move (l,c) ((l1,l2,l3): grille) j : grille =
 let move_ligne c (p1,p2,p3) j = match c with
  | 'a' -> if (p1=Vide) then (P j, p2, p3) else raise  MoveForbidden
  | 'b' -> if (p2=Vide) then (p1, P j, p3) else raise  MoveForbidden
  | 'c' -> if (p3=Vide) then (p1, p2, P j) else raise  MoveForbidden
  | _ -> raise  MoveForbidden
 in match l with
  | 1 -> ((move_ligne c l1 j),l2,l3)
  | 2 -> (l1, (move_ligne c l2 j),l3)
  | 3 -> (l1,l2, (move_ligne c l3 j))
  | _ -> raise MoveForbidden

(* Fonction pour tester un Vide *)
let aVide ((l1,l2,l3):grille) =
 let aVide_ligne (p1,p2,p3) = (p1=Vide) ||(p2=Vide) || (p3=Vide) in
 aVide_ligne l1 || aVide_ligne l2 || aVide_ligne l3

(* Fonction qui verifie si un joueur a gagne *)
let gagne (((p1,p2,p3),(p4,p5,p6),(p7,p8,p9)):grille) =
 if aVide ((p1,p2,p3),(p4,p5,p6),(p7,p8,p9)) then match p1 with
  | P j when (p1=p2 && p2=p3) || (p1=p4 && p4=p7) || (p1=p5 && p5=p9) -> P j
  | _ -> match p2 with
   | P j when (p2=p5 && p5=p8) -> P j
   | _ -> match p3 with
    | P j when (p3=p6 && p6=p9) || (p3=p5&&p5=p7) -> P j
    | _ -> match p4 with
     | P j when p4=p5 && p5=p6 -> P j
     | _ -> match p7 with
      | P j when p7=p8 && p8=p9 -> P j
      | _ -> Vide
 else raise NoMoreMove
```

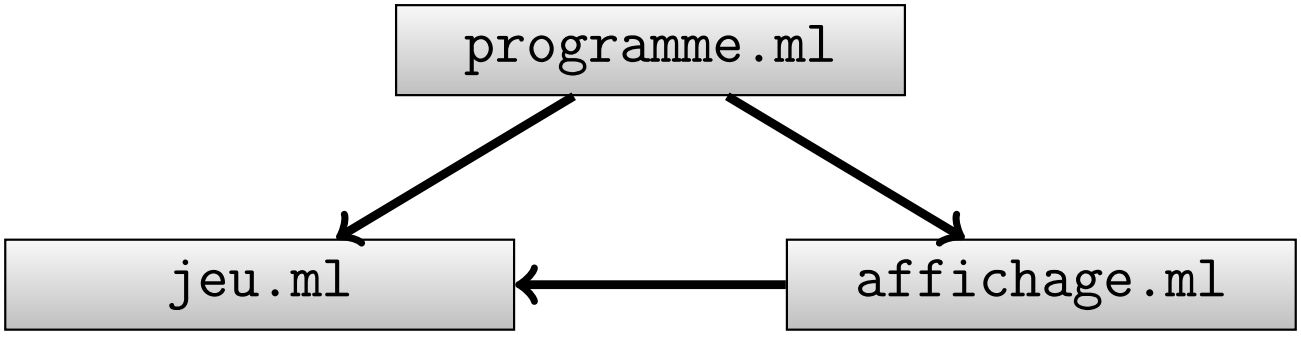
Implantation affichage.ml

```
1   open Jeu
2
3   (* Fonction qui affiche un pion *)
4   let affichePion = function
5         P Croix -> print_char 'X'
6       | P Rond -> print_char 'O'
7       | Vide -> print_char ' '
8
9   (* Fonction qui affiche une ligne *)
10  let afficheLigne ((p1, p2, p3):ligne) =
11      print_string "| "; affichePion p1 ;
12      print_string " | " ; affichePion p2 ;
13      print_string " | " ; affichePion p3 ;
14      print_string " |\n"
15
16  (* Fonction qui affiche la grille *)
17  let afficheGrille = fun ((l1, l2, l3):grille) ->
18      print_string "\n      a   b   c\n" ;
19      print_string "   +---+---+---+\n" ;
20      print_string " 1 " ; afficheLigne l1;
21      print_string "   +---+---+---+\n" ;
22      print_string " 2 " ; afficheLigne l2;
23      print_string "   +---+---+---+\n" ;
24      print_string " 3 " ; afficheLigne l3;
25      print_string "   +---+---+---+\n" ;;
```

Implantation programme.ml

```
1  open Jeu
2  open Affichage
3
4  (*entreer coordonees*)
5  let rec read_coordonees g j =
6   try
7    print_string "Ligne ? ";
8    let l=read_int () in
9    print_string "Colonne ? ";
10   let c=(read_line ()).[0] in
11   move (l,c) g j
12  with
13  | MoveForbidden ->
14  (print_endline "Choix interdite !" ; read_coordonees g j)
15
16 let tour g j =
17  afficheGrille g;
18  print_string "Bonjour "; affichePion (P j) ;print_newline ();
19  read_coordonees g j
20
21 (*negation joueur*)
22 let autre j = if j=Croix then Rond else Croix
23
24 (*fonction principale*)
25 let rec morpion g j=
26  try
27   let x = gagne g in
28   match x with
29   | P winner -> print_string "Bravo : "; affichePion (P winner) ; print_endline
30   | Vide -> let g_next = (tour g j) in morpion g_next (autre j)
31  with
32  | NoMoreMove -> print_endline ("Aucun gagnant, desole !")
33
34 (* grille initiale *)
35 let init:grille = ((Vide,Vide,Vide),(Vide,Vide,Vide),(Vide,Vide,Vide));;
36
37 (*main*)
38 morpion init Croix;;
```

Graphe de dépendance



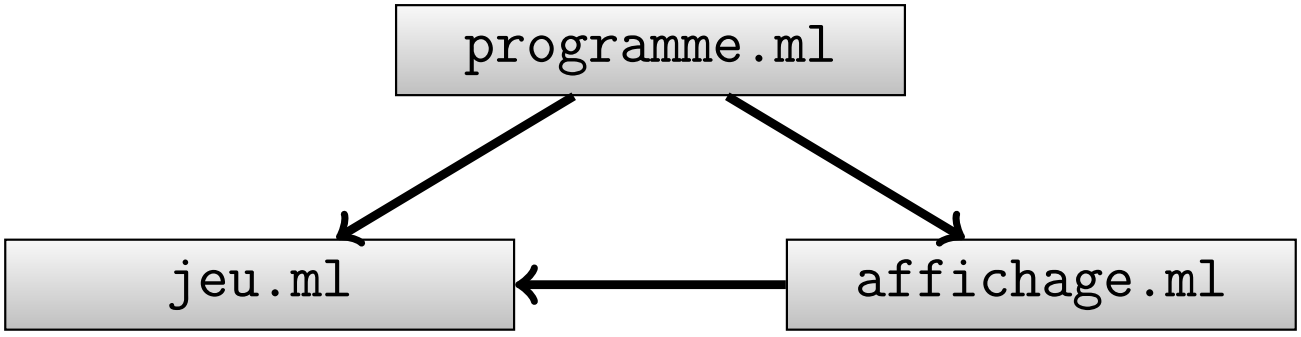
- Le graphe de dépendance doit être acyclique
- le compilation est séparée module par module et doit suivre l'ordre de dépendance:
  - si `A` dépend de `B`, alors on compile d'abord `B` et puis `A`

```
ocamlc -c jeu.ml
ocamlc -c affichage.ml
ocamlc -c programme.ml
```

- À la fin : assemblage des morceaux de code et résolution des symboles (en angl.: linking)

```
ocamlc -o morpion jeu.cmo affichage.cmo programme.cmo
```

Graphe de dépendance



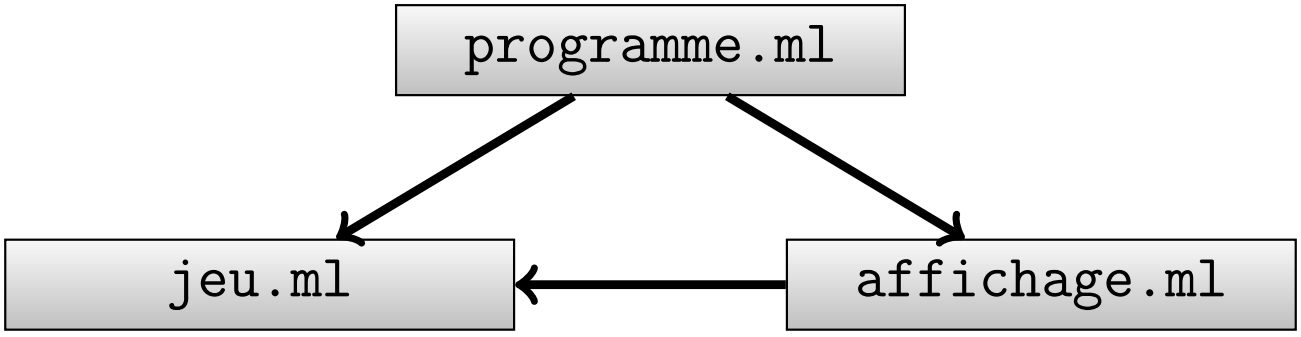
- Le graphe de dépendance doit être acyclique
- le compilation est séparée module par module et doit suivre l'ordre de dépendance:
  - si `A` dépend de `B`, alors on compile d'abord `B` et puis `A`

```
ocamlc -c jeu.ml
ocamlc -c affichage.ml
ocamlc -c programme.ml
```

- À la fin : assemblage des morceaux de code et résolution des symboles (en angl.: linking)

```
ocamlc -o morpion jeu.cmo affichage.cmo programme.cmo
```

Graphe de dépendance



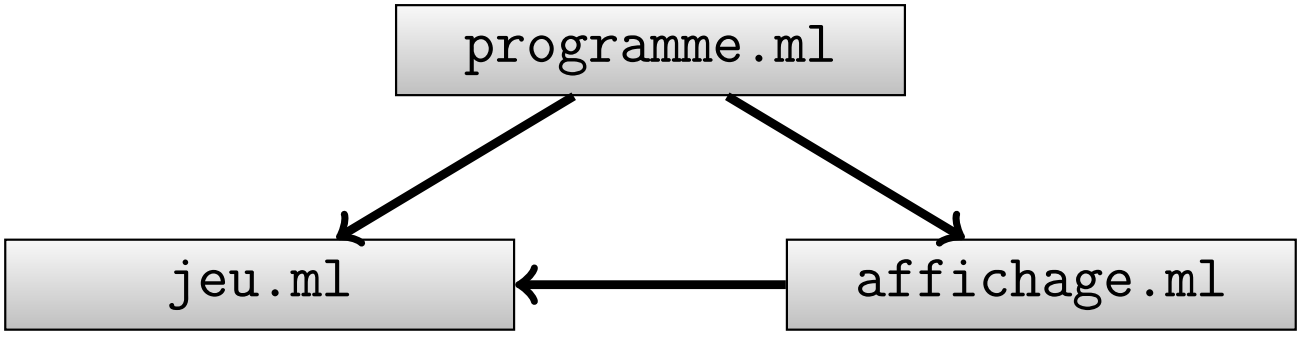
- Le graphe de dépendance doit être acyclique
- le compilation est séparée module par module et doit suivre l'ordre de dépendance:
  - si `A` dépend de `B`, alors on compile d'abord `B` et puis `A`

```
ocamlc -c jeu.ml
ocamlc -c affichage.ml
ocamlc -c programme.ml
```

- À la fin : assemblage des morceaux de code et résolution des symboles (en angl.: linking)

```
ocamlc -o morpion jeu.cmo affichage.cmo programme.cmo
```

Graphe de dépendance



- Le graphe de dépendance doit être acyclique
- le compilation est séparée module par module et doit suivre l'ordre de dépendance:
  - si `A` dépend de `B`, alors on compile d'abord `B` et puis `A`

```
ocamlc -c jeu.ml
ocamlc -c affichage.ml
ocamlc -c programme.ml
```

- À la fin : assemblage des morceaux de code et résolution des symboles (en angl.: linking)

```
ocamlc -o morpion jeu.cmo affichage.cmo programme.cmo
```

Interface et encapsulation

Les interfaces de OCaml

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

- des définition de types:
  - définitions concretes, p. ex:
    ```
    type joueur = Croix | Rond
    ```
  - définitions abstraites, p. ex:
    ```
    type grille
    ```
- des déclaration d'exceptions, p. ex.:
  ```
  exception MoveForbidden
  ```
- des déclarations d'identificateurs avec leur type, p. ex:
  ```
  val gagne :  grille -> pion
  ```
- et bien sûr des commentaires !

La commande `ocamlc -i` permet de générer automatiquement des interfaces

Les interfaces de OCaml

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

- des définition de types:
  - définitions concretes, p. ex:
    `type joueur = Croix | Rond`
  - définitions abstraites, p. ex:
    `type grille`
- des déclaration d'exceptions, p. ex.:
  `exception MoveForbidden`
- des déclarations d'identificateurs avec leur type, p. ex:
  `val gagne : grille -> pion`
- et bien sûr des commentaires !

La commande `ocamlc -i` permet de générer automatiquement des interfaces

Les interfaces de OCaml

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

- des définition de types:
  - définitions concretes, p. ex:
    `type joueur = Croix | Rond`
  - définitions abstraites, p. ex:
    `type grille`
- des déclaration d'exceptions, p. ex.:
  `exception MoveForbidden`
- des déclarations d'identificateurs avec leur type, p. ex:
  `val gagne : grille -> pion`
- et bien sûr des commentaires !

La commande `ocamlc -i` permet de générer automatiquement des interfaces

Les interfaces de OCaml

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

- des définition de types:
  - définitions concretes, p. ex:
    `type joueur = Croix | Rond`
  - définitions abstraites, p. ex:
    `type grille`
- des déclaration d'exceptions, p. ex.:
  `exception MoveForbidden`
- des déclarations d'identificateurs avec leur type, p. ex:
  `val gagne : grille -> pion`
- et bien sûr des commentaires !

La commande `ocamlc -i` permet de générer automatiquement des interfaces

Les interfaces de OCaml

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

- des définition de types:
  - définitions concretes, p. ex:
    `type joueur = Croix | Rond`
  - définitions abstraites, p. ex:
    `type grille`
- des déclaration d'exceptions, p. ex.:
  `exception MoveForbidden`
- des déclarations d'identificateurs avec leur type, p. ex:
  `val gagne : grille -> pion`
- et bien sûr des commentaires !

La commande `ocamlc -i` permet de générer automatiquement des interfaces

Une interface de jeu

```
$ ocamlc -i jeu.ml
type joueur = Croix | Rond
type pion = P of joueur | Vide
type ligne = pion * pion * pion
type grille = ligne * ligne * ligne
exception MoveForbidden
exception NoMoreMove
val move : int * char -> grille -> joueur -> grille
val aVide : grille -> bool
val gagne : grille -> pion
```

Pourquoi ne serait ce pas une bonne interface ?

– Parce qu'il n'y a pas de commentaire !

+ Interface est un "contrat" entre programmeur d'un module et son utilisateur:
  - elle doit contenir toutes les informations qui sont nécessaires pour utiliser le module (avec des commentaires !!!)
  - le codage d'un module peut être changé sans que cette modification ne soit visible de l'extérieur

– Parce qu'elle exporte toutes les définitions du module !

+ Principe d'encapsulation:
  - l'interface d'un module peut être plus abstraite que son implantation. Cacher les fonctions, types, exceptions auxiliaires.
  - le corps peut contenir des fonctions, types, exceptions privés

Pourquoi ne serait ce pas une bonne interface ?

– Parce qu'il n'y a pas de commentaire !

\+ Interface est un "contrat" entre programmeur d'un module et son utilisateur:

- elle doit contenir toutes les informations qui sont nécessaires pour utiliser le module (avec des commentaires !!!)
- le codage d'un module peut être changé sans que cette modification ne soit visible de l'extérieur

– Parce qu'elle exporte toutes les définitions du module !

\+ Principe d'encapsulation:

- l'interface d'un module peut être plus abstraite que son implantation. Cacher les fonctions, types, exceptions auxiliaires.
- le corps peut contenir des fonctions, types, exceptions privés

Pourquoi ne serait ce pas une bonne interface ?

– Parce qu'il n'y a pas de commentaire !

\+ Interface est un "contrat" entre programmeur d'un module et son utilisateur:

- elle doit contenir toutes les informations qui sont nécessaires pour utiliser le module (avec des commentaires !!!)
- le codage d'un module peut être changé sans que cette modification ne soit visible de l'extérieur

– Parce qu'elle exporte toutes les définitions du module !

\+ Principe d'encapsulation:

- l'interface d'un module peut être plus abstraite que son implantation. Cacher les fonctions, types, exceptions auxiliaires.
- le corps peut contenir des fonctions, types, exceptions privés

Pourquoi ne serait ce pas une bonne interface ?

– Parce qu'il n'y a pas de commentaire !

\+ Interface est un "contrat" entre programmeur d'un module et son utilisateur:

- elle doit contenir toutes les informations qui sont nécessaires pour utiliser le module (avec des commentaires !!!)
- le codage d'un module peut être changé sans que cette modification ne soit visible de l'extérieur

– Parce qu'elle exporte toutes les définitions du module !

\+ Principe d'encapsulation:

- l'interface d'un module peut être plus abstraite que son implantation. Cacher les fonctions, types, exceptions auxiliaires.
- le corps peut contenir des fonctions, types, exceptions privés

Une bonne interface `jeu.mli`

```
1  type joueur = Croix | Rond
2  type pion = P of joueur | Vide
3  type ligne = pion * pion * pion
4  type grille = ligne * ligne * ligne
5
6  exception MoveForbidden
7  exception NoMoreMove
8
9  (* Change un pion de la grille de vide a' joueur*)
10 (* leve exception MoveForbidden si changement n'est pas legale*)
11 val move : int * char -> grille -> joueur -> grille
12
13
14 (* Verifie si un joueur a gagne', etant donne' une grille *)
15 (* leve exception NoMoreMove si la partie est termine' *)
16 val gagne : grille -> pion
```

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 ? quel est le résultat ?
- Puis, couper le fonctionnement du programme en sous tâches:
  - identifier les fonctionnalités principales de la sous tâches ($\Rightarrow$ fonctions exportées / privées)
  - les types de données
  - les fonctionnalités partagées pas les sous tâches ($\Rightarrow$ modules partagés)
  - dessiner le graphe de dépendance entre modules
- Il est utile d'avoir une idée grossière de l'implémentation des modules.

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 ? quel est le résultat ?
- Puis, couper le fonctionnement du programme en sous tâches:
  - identifier les fonctionnalités principales de la sous tâches ($\Rightarrow$ fonctions exportées / privées)
  - les types de données
  - les fonctionnalités partagées pas les sous tâches ($\Rightarrow$ modules partagés)
  - dessiner le graphe de dépendance entre modules
- Il est utile d'avoir une idée grossière de l'implémentation des modules.

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 ? quel est le résultat ?
- Puis, couper le fonctionnement du programme en sous tâches:
  - identifier les fonctionnalités principales de la sous tâches ($\Rightarrow$ fonctions exportées / privées)
  - les types de données
  - les fonctionnalités partagées pas les sous tâches ($\Rightarrow$ modules partagés)
  - dessiner le graphe de dépendance entre modules
- Il est utile d'avoir une idée grossière de l'implémentation des modules.

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 ? quel est le résultat ?
- Puis, couper le fonctionnement du programme en sous tâches:
  - identifier les fonctionnalités principales de la sous tâches ($\Rightarrow$ fonctions exportées / privées)
  - les types de données
  - les fonctionnalités partagées pas les sous tâches ($\Rightarrow$ modules partagés)
  - dessiner le graphe de dépendance entre modules
- Il est utile d'avoir une idée grossière de l'implémentation des modules.

Comment trouver le bon découpage en modules ?

- Cas 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 autres pour toutes les fonctions, etc. . .
- Bon découpage:
  - correspond à la logique du programme
  - modules d'une taille raisonnable
  - définition auxiliaires cachées dans les modules
  - réutilisation du code au lieu de duplication

Comment trouver le bon découpage en modules ?

- Cas 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 autres pour toutes les fonctions, etc. . .
- Bon découpage:
  - correspond à la logique du programme
  - modules d'une taille raisonnable
  - définition auxiliaires cachées dans les modules
  - réutilisation du code au lieu de duplication

Comment trouver le bon découpage en modules ?

- Cas 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 autres pour toutes les fonctions, etc. . .
- Bon découpage:
  - correspond à la logique du programme
  - modules d'une taille raisonnable
  - définition auxiliaires cachées dans les modules
  - réutilisation du code au lieu de duplication

Comment trouver le bon découpage en modules ?

- Cas 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 autres pour toutes les fonctions, etc. . .
- Bon découpage:
  - correspond à la logique du programme
  - modules d'une taille raisonnable
  - définition auxiliaires cachées dans les modules
  - réutilisation du code au lieu de duplication

Comment trouver le bon découpage en modules ?

- Cas 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 autres pour toutes les fonctions, etc. . .
- Bon découpage:
  - correspond à la logique du programme
  - modules d'une taille raisonnable
  - définition auxiliaires cachées dans les modules
  - réutilisation du code au lieu de duplication

Comment trouver le bon découpage en modules ?

- Cas 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 autres pour toutes les fonctions, etc. . .
- Bon découpage:
  - correspond à la logique du programme
  - modules d'une taille raisonnable
  - définition auxiliaires cachées dans les modules
  - réutilisation du code au lieu de duplication

Comment trouver le bon découpage en modules ?

- Cas 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 autres pour toutes les fonctions, etc. . .
- Bon découpage:
  - correspond à la logique du programme
  - modules d'une taille raisonnable
  - définition auxiliaires cachées dans les modules
  - réutilisation du code au lieu de duplication

Comment trouver le bon découpage en modules ?

- Cas 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 autres pour toutes les fonctions, etc. . .
- Bon découpage:
  - correspond à la logique du programme
  - modules d'une taille raisonnable
  - définition auxiliaires cachées dans les modules
  - réutilisation du code au lieu de duplication

Comment trouver le bon découpage en modules ?

- Cas 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 autres pour toutes les fonctions, etc. . .
- Bon découpage:
  - correspond à la logique du programme
  - modules d'une taille raisonnable
  - définition auxiliaires cachées dans les modules
  - réutilisation du code au lieu de duplication

Quelques mots sur la documentation

- Documenter la structure globale du programme
  ($\Rightarrow$ 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.
- Implantation des modules :
  - Spécifier les fonctions privées.
  - Expliquer l'algorithme utilisée quand pas évident.
  - Donner des invariants des fonctions (utiliser des constructions du langage (assertions si possible))

Quelques mots sur la documentation

- Documenter la structure globale du programme
  ($\Rightarrow$ 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.
- Implantation des modules :
  - Spécifier les fonctions privées.
  - Expliquer l'algorithme utilisée quand pas évident.
  - Donner des invariants des fonctions (utiliser des constructions du langage (assertions si possible))

Quelques mots sur la documentation

- Documenter la structure globale du programme
  ($\Rightarrow$ 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.
- Implantation des modules :
  - Spécifier les fonctions privées.
  - Expliquer l'algorithme utilisée quand pas évident.
  - Donner des invariants des fonctions (utiliser des constructions du langage (assertions si possible))

Quelques mots sur la documentation

- Documenter la structure globale du programme
  ($\Rightarrow$ 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.
- Implantation des modules :
  - Spécifier les fonctions privées.
  - Expliquer l'algorithme utilisée quand pas évident.
  - Donner des invariants des fonctions (utiliser des constructions du langage (assertions si possible))

Quelques mots sur la documentation

- Documenter la structure globale du programme
  ($\Rightarrow$ 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.
- Implantation des modules :
  - Spécifier les fonctions privées.
  - Expliquer l'algorithme utilisée quand pas évident.
  - Donner des invariants des fonctions (utiliser des constructions du langage (assertions si possible))

Quelques mots sur la documentation

- Documenter la structure globale du programme
  ($\Rightarrow$ 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.
- Implantation des modules :
  - Spécifier les fonctions privées.
  - Expliquer l'algorithme utilisée quand pas évident.
  - Donner des invariants des fonctions (utiliser des constructions du langage (assertions si possible))

Quelques mots sur la documentation

- Documenter la structure globale du programme
  ($\Rightarrow$ 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.
- Implantation des modules :
  - Spécifier les fonctions privées.
  - Expliquer l'algorithme utilisée quand pas évident.
  - Donner des invariants des fonctions (utiliser des constructions du langage (assertions si possible))

Quelques mots sur la documentation

- Documenter la structure globale du programme
  ($\Rightarrow$ 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.
- Implantation des modules :
  - Spécifier les fonctions privées.
  - Expliquer l'algorithme utilisée quand pas évident.
  - Donner des invariants des fonctions (utiliser des constructions du langage (assertions si possible))

Quelques mots sur la documentation

- Documenter la structure globale du programme
  ($\Rightarrow$ 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.
- Implantation des modules :
  - Spécifier les fonctions privées.
  - Expliquer l'algorithme utilisée quand pas évident.
  - Donner des invariants des fonctions (utiliser des constructions du langage (assertions si possible))

Quelques mots sur la documentation

- Documenter la structure globale du programme
  ($\Rightarrow$ 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.
- Implantation des modules :
  - Spécifier les fonctions privées.
  - Expliquer l'algorithme utilisée quand pas évident.
  - Donner des invariants des fonctions (utiliser des constructions du langage (assertions si possible))

Dépendances et `Makefile`

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, plus refaire l'exécutable.
- Une solution : `make`
  - un outil de développement sous UNIX, il peut aussi être utilisé pour compiler du C, C++, Java, etc. . .
  - Principe de base:
    - Règles qui décrivent des dépendances entre des cibles
    - un cible est un nom de fichier
    - avec chaque règle : instructions (commandes shell) pour mettre à jour un cible.

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, plus refaire l'exécutable.
- Une solution : `make`
  - un outil de développement sous UNIX, il peut aussi être utilisé pour compiler du `C`, `C++`, `Java`, etc...
  - Principe de base:
    - Règles qui décrivent des dépendances entre des cibles
    - un cible est un nom de fichier
    - avec chaque règle : instructions (commandes shell) pour mettre à jour un cible.

Makefile sur l'exemple du Morpion

```
#Edition des liens et creation de l'executable
morpion: jeu.cmo affichage.cmo programme.cmo
	ocamlc -o morpion jeu.cmo affichage.cmo programme.cmo

#Compilation du corps du module jeu
jeu.cmo: jeu.ml jeu.cmi
	ocamlc -c jeu.ml

#Compilation de l'interface du module jeu
jeu.cmi: jeu.mli
	ocamlc jeu.mli

#Compilation du corps du module affichage
affichage.cmo: affichage.ml jeu.cmi
	ocamlc -c affichage.ml

#Compilation du corps du module programme
programme.cmo: programme.ml jeu.cmi affichage.cmo
	ocamlc -c programme.ml

#Effacer fichiers auxiliares
clean:
	rm *.cmi *.cmo
```

Exemple de utilisation de `Makefile`

```
$ make
ocamlc jeu.mli
ocamlc -c jeu.ml
ocamlc -c affichage.ml
ocamlc -c programme.ml
ocamlc -o morpion jeu.cmo affichage.cmo programme.cmo
```

- Il existe aussi un outil spécialisé dans la distribution standard de OCaml: `ocamlbuild`. Dans de cas simple gère lui seul toute la compilation, et il est très configurable pour des cas complexes.

  $\Rightarrow$ voir Partie III du manual de OCaml