

Programmation Fonctionnelle

Cours 07

Michele Pagani



Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

`pagani@pps.univ-paris-diderot.fr`

6 novembre 2014

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 *Analyse syntaxique et compilation* (L3 2nd semestre),

cours *Compilation* du M1 et M2

Compilation

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

Étapes de compilation (simplifié):

- ➊ Analyse syntaxique : peut détecter une erreur de syntaxe.
- ➋ Analyse sémantique : peut détecter une erreur de typage, filtrages non exhaustives, etc.
- ➌ Génération d'un code intermédiaire.
- ➍ Éventuellement optimisations diverses du code.
- ➎ Production du code cible, peut nécessiter la liaison avec des bibliothèques.

Pour en savoir plus :

cours *Analyse syntaxique et compilation* (L3 2nd semestre),

cours *Compilation* du M1 et M2

Compilation

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

Étapes de compilation (simplifié):

- ➊ Analyse syntaxique : peut détecter une erreur de syntaxe.
- ➋ Analyse sémantique : peut détecter une erreur de typage, filtrages non exhaustives, etc.
- ➌ Génération d'un code intermédiaire.
- ➍ Éventuellement optimisations diverses du code.
- ➎ Production du code cible, peut nécessiter la liaison avec des bibliothèques.

Pour en savoir plus :

cours *Analyse syntaxique et compilation* (L3 2nd semestre),

cours *Compilation* du M1 et M2

Compilation

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

Étapes de compilation (simplifié):

- ➊ Analyse syntaxique : peut détecter une erreur de syntaxe.
- ➋ Analyse sémantique : peut détecter une erreur de typage, filtrages non exhaustives, etc.
- ➌ Génération d'un code intermédiaire.
- ➍ Éventuellement optimisations diverses du code.
- ➎ Production du code cible, peut nécessiter la liaison avec des bibliothèques.

Pour en savoir plus :

cours *Analyse syntaxique et compilation* (L3 2nd semestre),

cours *Compilation* du M1 et M2

Compilation

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

Étapes de compilation (simplifié):

- ➊ Analyse syntaxique : peut détecter une erreur de syntaxe.
- ➋ Analyse sémantique : peut détecter une erreur de typage, filtrages non exhaustives, etc.
- ➌ Génération d'un code intermédiaire.
- ➍ Éventuellement optimisations diverses du code.
- ➎ Production du code cible, peut nécessiter la liaison avec des bibliothèques.

Pour en savoir plus :

cours *Analyse syntaxique et compilation* (L3 2nd semestre),

cours *Compilation* du M1 et M2

Compilation

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

Étapes de compilation (simplifié):

- ➊ Analyse syntaxique : peut détecter une erreur de syntaxe.
- ➋ Analyse sémantique : peut détecter une erreur de typage, filtrages non exhaustives, etc.
- ➌ Génération d'un code intermédiaire.
- ➍ Éventuellement optimisations diverses du code.
- ➎ Production du code cible, peut nécessiter la liaison avec des bibliothèques.

Pour en savoir plus :

cours *Analyse syntaxique et compilation* (L3 2nd semestre),

cours *Compilation* du M1 et M2

Compilation

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

Étapes de compilation (simplifié):

- ➊ Analyse syntaxique : peut détecter une erreur de syntaxe.
- ➋ Analyse sémantique : peut détecter une erreur de typage, filtrages non exhaustives, etc.
- ➌ Génération d'un code intermédiaire.
- ➍ Éventuellement optimisations diverses du code.
- ➎ Production du code cible, peut nécessiter la liaison avec des bibliothèques.

Pour en savoir plus :

cours *Analyse syntaxique et compilation* (L3 2nd semestre),

cours *Compilation* du M1 et M2

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)` : première 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)` : première 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)` : première option
 - `Sys.argv.(2)` : deuxième option
 - `⋮`

Structuration des programmes en unités de compilation


```

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

```

An old friend...

```

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  (*entree 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
```

	a	b	c
1			
2			
3			

```
Bonjour X  
Ligne ?
```

Décomposition en modules

- On peut faire émerger la **structure logique** d'un programme en le découpant en plusieurs fichiers
 - **unités de compilation** (ou mieux **modules**)
 - **interface**: extension `.mli` (optionelle)
résumé des définitions d'un module
 - **implantation**: extension `.ml`
code réalisant les définitions
- par exemple, `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
 - **unités de compilation** (ou mieux **modules**)
 - **interface**: extension `.mli` (optionelle)
résumé des définitions d'un module
 - **implantation**: extension `.ml`
code réalisant les définitions
- par exemple, `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
 - **unités de compilation** (ou mieux **modules**)
 - **interface**: extension `.mli` (optionelle)
résumé des définitions d'un module
 - **implantation**: extension `.ml`
code réalisant les définitions
- par exemple, `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

```
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): grille) =
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
```

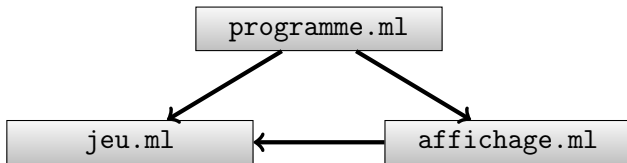

Implantation affichage.ml

```
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

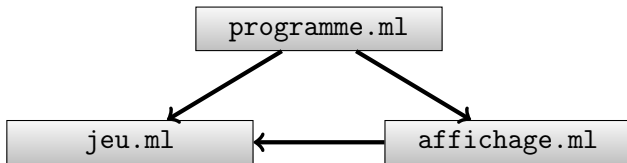
```
1 open Jeu
2 open Affichage
3
4 (*entree coordonees*)
5 let rec read_coordonees g j =
6   try
7     print_string "Ligne? ";
8     let l=read_int () in
9     print_string "Colonne? ";
10    let c=(read_line ())[0] in
11    move (l,c) g j
12  with
13  | MoveForbidden ->
14    (print_endline "Choix 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



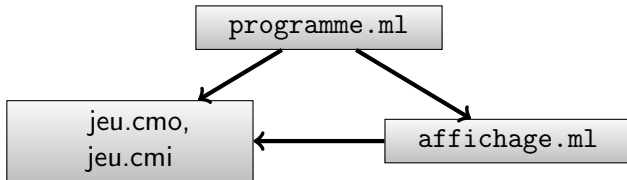
- Le graphe de dépendance doit être **acyclique**
- la compilation est séparée module par module et doit suivre l'ordre de dépendance:
 - si A dépend de B, alors on compile d'abord B et puis A
- À la fin : assemblage des morceaux de code et résolution des symboles (en angl.: **linking**)
`ocamlc -o morpion jeu.cmo affichage.cmo programme.cmo`

Graphe de dépendance



- Le graphe de dépendance doit être **acyclique**
- la 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
- 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



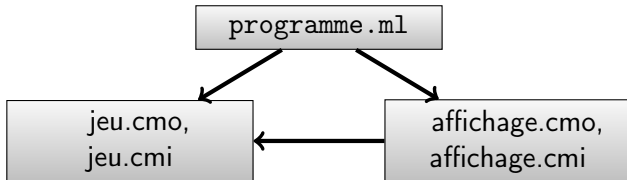
- Le graphe de dépendance doit être **acyclique**
- la compilation est séparée module par module et doit suivre l'ordre de dépendance:
 - si A dépend de B, alors on compile d'abord B et puis A

`ocamlc -c jeu.ml`

- 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



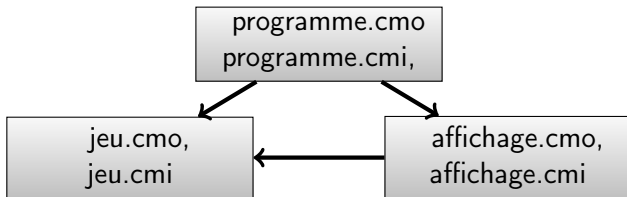
- Le graphe de dépendance doit être **acyclique**
- la compilation est séparée module par module et doit suivre l'ordre de dépendance:
 - si A dépend de B, alors on compile d'abord B et puis A

`ocamlc -c affichage.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



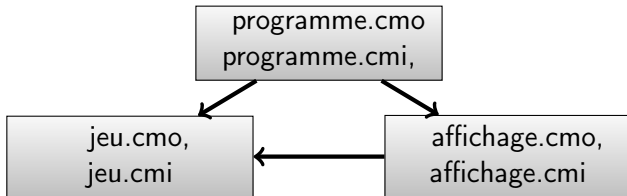
- Le graphe de dépendance doit être **acyclique**
- la compilation est séparée module par module et doit suivre l'ordre de dépendance:
 - si A dépend de B, alors on compile d'abord B et puis A

`ocamlc -c 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**
- la 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
- 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 concrètes, 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 concrètes, 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 concrètes, 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 concrètes, 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 concrètes, 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 (⇒ fonctions exportées / privées)
 - les types de données
 - les fonctionnalités partagées pas les sous tâches (⇒ 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 (⇒ fonctions exportées / privées)
 - les types de données
 - les fonctionnalités partagées pas les sous tâches (⇒ 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.

Exemple de Makefile

```
# 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 implantation module jeu
jeu.cmo : jeu.ml jeu.cmi
    ocamlc -c jeu.ml

# Compilation interface module jeu
jeu.cmi : jeu.mli
    ocamlc -c jeu.mli

# Compilation implantation module affichage
affichage.cmo : affichage.ml
    ocamlc -c affichage.ml

# Compilation implantation module programme
programme.cmo : programme.ml
    ocamlc -c programme.ml
```

Exemple de utilisation de Makefile

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

- Pour en savoir plus:
cours [Environnements et outils de développement](#)
(L3, 2nd semestre)