Cours de Programmation 2

Programmation à moyenne et large échelle

1. Programmation modulaire
2. Programmation orientée objet
3. Programmation concurrente, distribuée
4. Programmation monadique
5. Les programmes dans leur environnement : l'approche UNIX

Langage de programmation : Objective Caml

Programmation à petite échelle

(*programming in the small*)

- programmation structurée : structuration de la sémantique du programme par la syntaxe (procédures, boucles `while` au lieu des `goto`)
- Sémantique dénotationnelle
- Programmer avec des invariants (*logique de Hoare*)

Le problème de la programmation à large échelle

(*programming in the large*)

Projets de programmation importants :
- $\geq 10^6$ lignes de code
- travail en groupe (la composition du groupe peut changer, tous les membres ne sont pas au même endroit, etc.)
- modifications du programme au long de son temps de vie
- conception, programmation, inspection, tests, documentation par des équipes différentes

The art of programming is the art of organizing complexity

(E. Dijkstra)

$\Rightarrow$ Découpage en « morceaux », mais pas n'importe comment :

- Découpage logique correspondant à la logique interne du projet
- Compilation séparée[a]
- Faciliter la maintenance
- Faciliter les extensions du programme
- Réutilisation du code (bibliothèques)

Mais pas seulement… on veut aussi, autant que possible, *ne pas écrire de code du tout !*

---

[a] La compilation de OpenOffice prend plus de 24h

Plan du chapitre *Modules*

1. Modules comme unités de compilation
2. Encapsulation et types abstraits
3. Analyse descendante
4. Le langage des modules
5. Modules paramétrés
6. Encapsulation et valeurs mutables ($\Rightarrow$*objets*)

1 Modules comme unités de compilation

Un *module* est une unité de programme qui regroupe des définitions[a].

- OCaml : types, valeurs (aussi de type fonctionnel), exceptions

- Pascal : constantes, variables, fonctions, procédures, etc.

Première approche : Module = Unité de compilation.

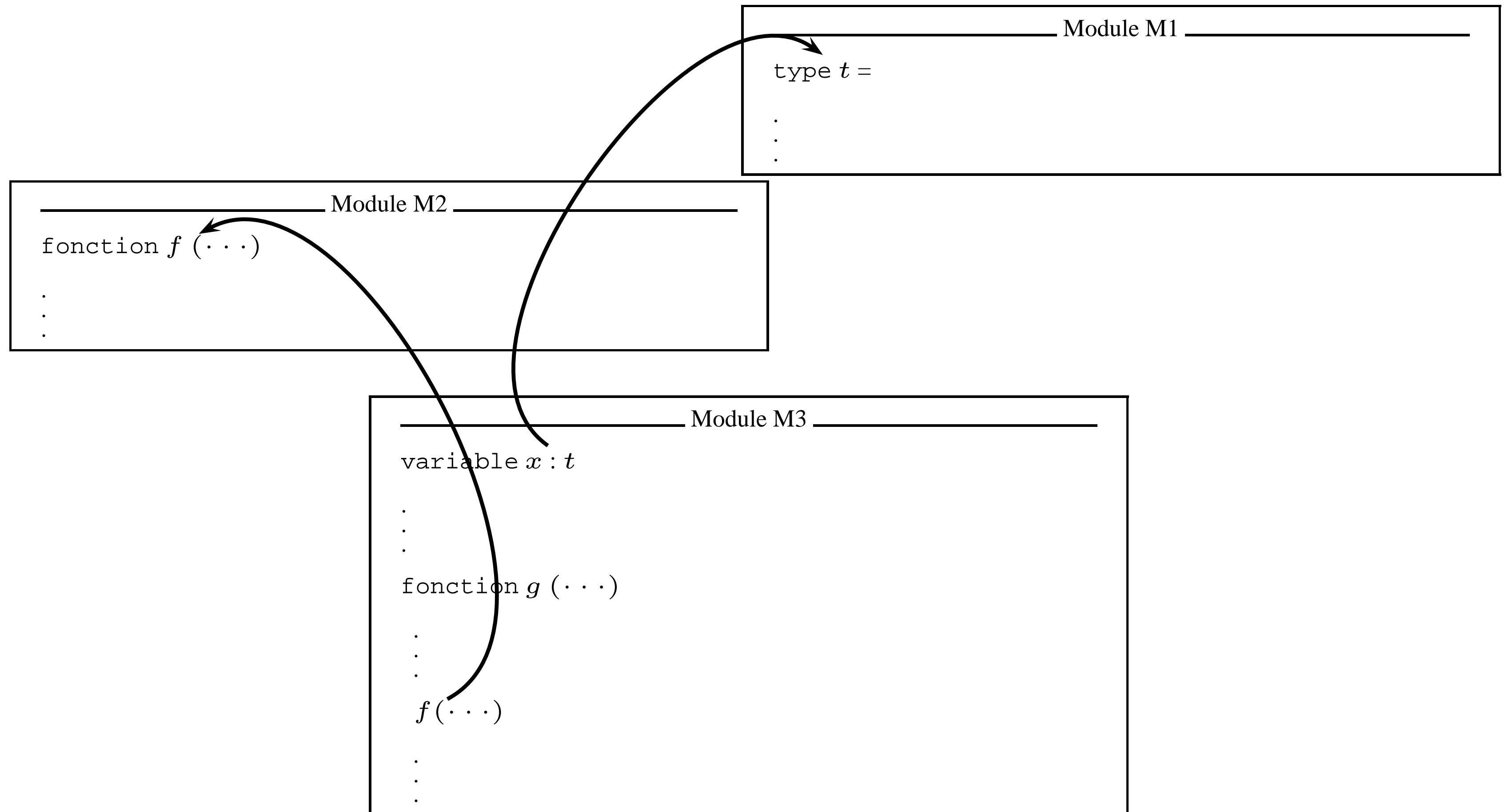
Le module B *exporte* quelque chose (par ex. une fonction), le module A l'*importe*.

---

[a]de quoi ?ça dépend du langage)

Relation entre modules : *Dépendance*. Module A dépend du module B ssi A utilise un nom défini par B.

Dépendances entre modules



Modules et compilation séparée

Si le module A dépend du module B, alors la *compilation* de A a besoin de connaître les définitions *effectuées* par B :

- soit textuellement (le compilateur regarde dans le source de B)
- soit en forme compilée (cas normal) $\Rightarrow$ compiler B avant de compiler A.

Dans le deuxième cas, la compilation d'un module A engendre[a]

- un « résumé » des définitions effectuée par A
- du code, avec des adresses symboliques pour les identificateurs définis dans des autres modules.

À la fin : assemblage des morceaux de code et résolution des symboles (édition des liens, angl. : *linking*).

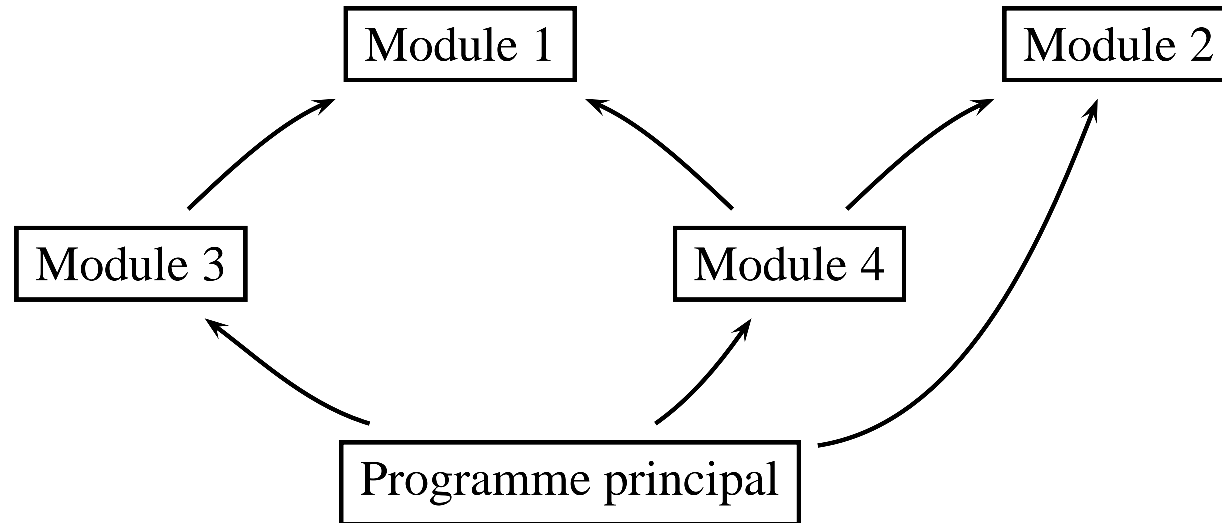
La relation de dépendance doit être *acyclique*.[b]

---

[a] On savait faire celadéjà aux temps de FORTRAN.

[b] pour l'instant, voir *mixins*

Un graphe de dépendances



Dans quel ordre peut-on compiler ces modules ? Réponse, on utilise un tri topologique, qui est à la base d'outils comme `make`.

Interface et corps d'un module

- Interface : Résumé des définitions d'un module
- Corps (Implantation) : Code réalisant les définitions

Possibilités pour organiser un module en interface et corps :

- Turbo Pascal : dans un seul fichier.
- OCaml : séparé dans un fichier interface `.mli` et un fichier corps `.ml`.
  Avantage : Si A dépend de B, alors on fixe d'abord l'interface de B. Puis, on peut rédiger (et même compiler) les corps de A et B indépendamment.

Exemple d’un module en Turbo Pascal

```
UNIT aut1;

INTERFACE

type    states  = (q0, q1, q2);
        symbols = 'a'..'b';

var
        initialstate     : states;
        finalstates      : set of states;
        transition       : array [states,symbols] of states;

procedure autinit();
```

(suite du fichier)

```
IMPLEMENTATION

procedure autinit;
begin
  initialstate  := q0;

  finalstates   := [q0,q1];

  transition[q0,'a'] := q1;
  transition[q0,'b'] := q0;
end;

END.
```

Exemple d'un module en OCaml

Fichier Interface aut1.mli

```
(* Module for the automaton ... *)

type states = Q0 | Q1 | Q2
type symbols = char

val initialstate: states
val finalstates: states list
val transition: states * symbols -> states

exception Transition_undefined
```

Fichier Corps aut1.ml

```
type states = Q0 | Q1 | Q2
type symbols = char
exception Transition_undefined

let initialstate = Q0
let finalstates = [Q0; Q1]
let transition = function
    Q0, 'a' -> Q1
  | Q0, 'b' -> Q1
  | _ -> raise Transition_undefined
```

Importation dans les modules de OCaml

Deux constructions :

1. Directive `open B` au début du module importeur A : rend accessible en `A` toutes les définitions exportées par `B`.

   Avantage : plus court.

2. Préfixer les noms par le nom du module : `B.f` dénote l'identificateur f exportée par le module B.

   Avantage : plus explicite, et pas d'ambiguïté (plusieurs module peuvent exporter le même identificateur).

Dépendance d’une interface d’un module d’un autre module

L’interface d’un module A peut dépendre d’un module B, c’est précisément le cas quand A exporte un type *concret* dont la définition utilise un type exporté par B.

Exemple : Le module `Expressions` exporte un type `expr`, et l’interface du module `Instruction` contient

```
type instr =
  Print of Expression.expr
| Affect of string * Expression.expr
| While of Expression.expr *instr list
```

Compilation des modules en OCaml

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

2 Encapsulation

(et plus sur la compilation séparée)

On peut toujours obtenir une interface d'un module OCaml en compilant le corps avec l'option `-i` :

```
% ocamlc -i -c aut1.ml
type states = Q0 | Q1 | Q2
and symbols = char
exception Transition_undefined
val initialstate : states
val finalstates : states list
val transition : states * char -> states
```

Pourtant, cette interface n’est pas satisfaisante pour deux raisons :

- Pas de commentaire (l’interface doit contenir une spécification des valeurs etc. définies sous forme de commentaire).
- Souvent on ne veut pas exporter toutes les définitions d’un module. Il n’y a pas d’*encapsulation*.

2.1 Le principe d'encapsulation

Exporter aussi peu de définitions que possible : l'interface d'un module peut être plus abstraite que son corps. On cache les fonctions, types, exceptions auxiliaires. En OCaml

- Une interface peut exporter un type *abstrait* : L'interface ne contient que la déclaration `type : t`, et le corps contient sa définition complète : `type t = ...`
- Si un type *concret* est exporté par l'interface, alors le corps doit[a] contenir la même définition.
- Tout identificateur (ou exception) exporté doit être défini par le corps, et cela avec un type égal ou plus général, éventuellement en utilisant les définitions des types abstraits.

Le corps peut contenir des types, fonctions, exceptions *privés* (pas exportés).

[a] très fatigant, des fois...

2.2 Intérêt de l’encapsulation

- L’interface comme « contrat » entre programmeur d’un module et son utilisateur : liberté de réalisation au programmeur.
  L’interface contient 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 vers l’extérieur.
- Types abstrait : maintenance d’un invariant puisque modifications que par des fonctions exportées.

L’encapsulation dans un module s’ajoute à la possibilité d’encapsulation par définition locale (à une fonction) d’un identificateur.

2.3 Exemple

Un module pour des tours d’entier (des piles d’entiers, où la valeur décroit vers le sommet, comme pour les Tours de Hanoï).

On ne veut permettre que la construction des tours qui satisfont l’invariant : les valeurs décroissent vers le sommet.

Solution : Déclarer un type abstrait, et ne permettre la modification d’un tour que par une fonction exportée par le module.

```
(* interface of the module for towers of integers. A tower i
   integers which are strictly decreasing from bottom to top

type tower
exception Operation_illegal
  (* the empty tower *)
val empty: tower
(* (push i t) returns a new tower consisting of t with addit
   the top, provided that result is still a tower. Otherwise
   Operation_illegal.*)
val push: int -> tower -> tower
(* (pop t) returns a tower which consists of t without its t
   raises Operation_illegal when t is empty. *)
val pop: tower -> tower
(* (top t) returns the top element of t, raises Operation_il
   t is empty. *)
val top: tower -> int
```

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

let empty = []
let top = function
    h::r -> h
  | [] -> raise Operation_illegal
(* (can_be_pushed i t) is true iff i can be pushed on t *)
let can_be_pushed i = function
    [] -> true
  | h::r -> i<h
let push i t =
  if can_be_pushed i t then i::t else raise Operation_illega
let pop = function
    h::r -> r
  | [] -> raise Operation_illegal
```

Programme principal

```
open Tour

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

ou, équivalent

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

Un Makefile pour compiler le tout

```
# Édition des liens et création de l'exécutable
main: tour.cmo main.cmo
	ocamlc -o main tour.cmo main.cmo

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

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

# Compilation du corps du module main
main.cmo: main.ml tour.cmi
	ocamlc -c main.ml
```

Calculer automatiquement les dépendances

```
% ocamldep *.mli *.ml
main.cmo: tour.cmi
main.cmx: tour.cmx
tour.cmo: tour.cmi
tour.cmx: tour.cmi
```

Un Makefile générique

```
.SUFFIXES: .ml .mli .cmo .cmi .cmx
OBJECTS = tour.cmo main.cmo

main: $(OBJECTS)
	ocamlc -o main $(OBJECTS)

.ml.cmo:
	ocamlc -c $<

.mli.cmi:
	ocamlc $<

include .depend

depend:
```

```
	ocamldep *.mli *.ml > .depend

clean:
	-rm *.cmo *.cmi main
```

Pour plus d'info sur `make` : en emacs, taper CONTROL-H I, puis M MAKE

3 Analyse descendante (top down)

(ou : Comment trouver le bon découpage en modules ?)

Analyse du plus général vers le plus particulier.

Commencer avec le programme entier : quelle est l'entrée, quelle est le résultat ?

Puis, couper le fonctionnement du programme en sous-tâches. Identifier les fonctionnalités principales de la sous-tâche ($\Rightarrow$ fonctions exportées), les types de données et des fonctionnalités partagés par les sous-tâches ($\Rightarrow$ modules auxiliaires utilisés par des autres modules).

Il faut déjà avoir une idée grossière de l'implémentation des modules.

Exemple

Un interpréteur pour un langage de programmation avec déclaration de variables typées (par ex. PASCAL).

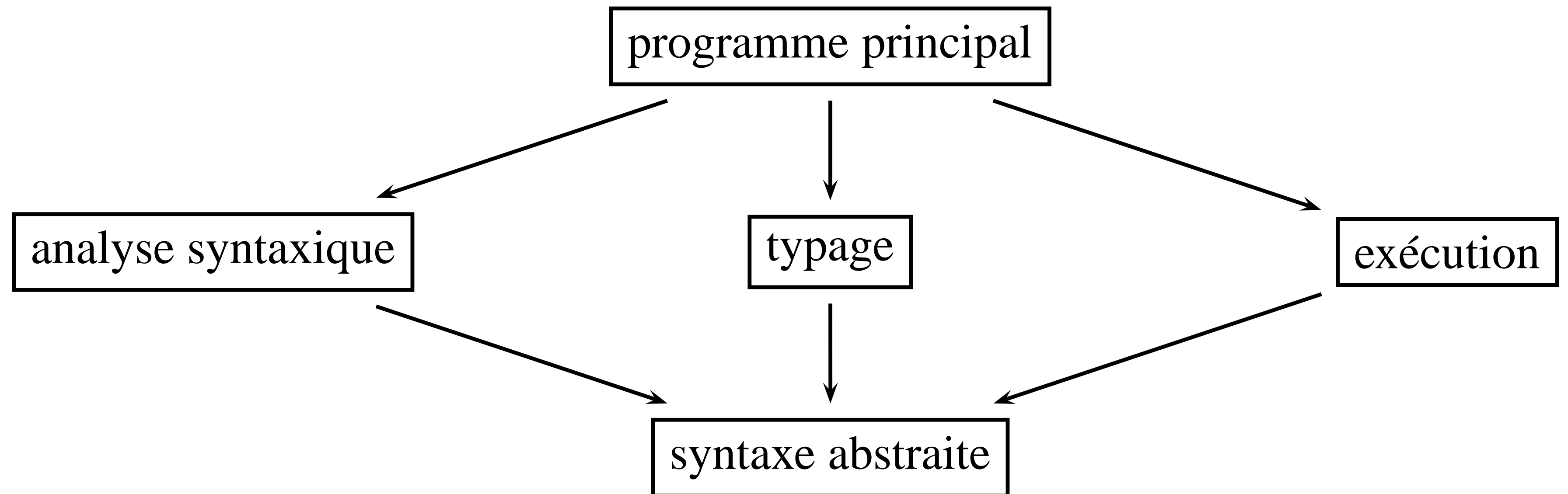
Fonctionnalité principale : lit un programme, déclenche une erreur si le programme n'est pas correcte, sinon l'exécute et affiche le résultat.

Découpage en sous-tâches :

1. Analyse syntaxique
2. Vérification des types et déclarations des variables
3. Exécution

Les trois modules travaillent sur la *syntaxe abstraite* des programmes.

Premier graphe de dépendance



Interface du module *Syntaxe*

La syntaxe abstraite est un type inductif, pas de raison de cacher sa définition.

```
type typ = Entier | Réel | Bool
type expr = Var of string
          | EConst of int
          | Plus of expr * expr
          ...
type instr = Affect of string * expr
           | While of expr * instr list
           ...
type decl = string * typ
type prog = decl list * instr list
```

Interface du module *typage*

```
val bien_typé : Syntaxe.prog -> bool
```

Interface du module *exécution*

```
exception: Division_par_zero
val: exec: Syntaxe.prog -> string
```

Fonctions privées de ces modules :

- module *typage* : typ_expr : typenv -> Syntaxe.expr -> Syntaxe.typ
- module *exécution* : eval_expr : valenv -> Syntaxe.expr -> valeur

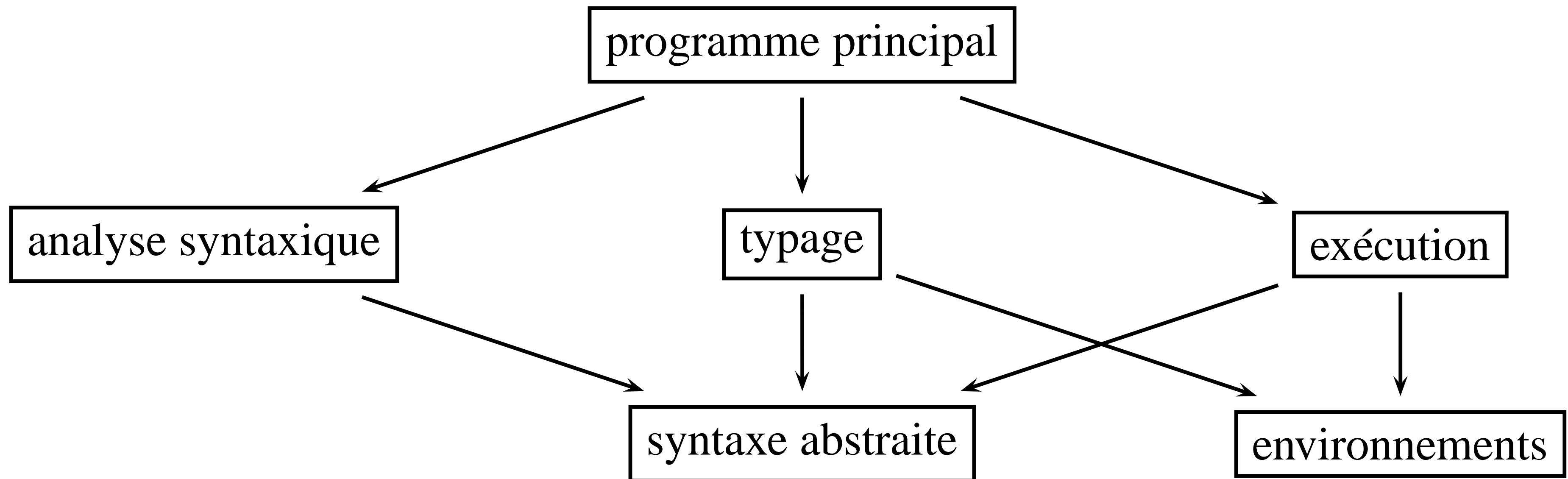
où

- *typenv* : Type des environnements de types
- *valenv* : Type des environnements de valeurs

Le type *valeur* doit être un type privé du module *exécution* (type somme regroupant int, real, etc.)

Par contre les deux types d'environnements peuvent avantageusement être généralisés : un type *polymorphe* d'environnement, défini dans un module à part.

Graphe de dépendance rectifié



Interface du module *Environnements*

```
(* type polymorphes des environnements *)
type 'a env
```

(plus des spécifications des exceptions et fonctions).

Est-ce que le module *Environnements* doit dépendre du module *Syntaxe* (qui défini le type `typ`) quand on veut construire des environnements de typage ?

On pourrait continuer le découpage, et

- couper l'analyse syntaxique en analyse lexicale et analyse syntaxique propre (mais là le découpage dépend plutôt des outils utilisés)
- couper le module *Syntaxe* en deux : Expressions et Instructions
- pareil couper les modules de typage et d'évaluation en deux (expressions et instructions)

mais c'est un peu excessif pour ce problème.

Remarques

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 autre pour toutes les fonctions, etc.

Bon découpage :

- Correspond à la logique du programme
- Modules d'une taille raisonnable
- Définition auxiliaires cachées dans les modules (l'organisation en modules simplifie la structure du programme)
- Réutilisation du code au lieu de duplication

Quelques mots sur la documentation

- Documenter la structure globale du programme (graphe de dépendance)
- Interfaces des modules : Documenter l'*utilisation* du module :
  - Son rôle général
  - Que représentent les types ?
  - Spécifier les fonctions : Expliquer les rôles des arguments, les hypothèses sur leurs valeurs (par ex : entier positif, liste triée, etc.), et bien sûr le résultat. N'oubliez pas les cas d'erreur.

  En général : La doc de l'interface doit contenir toutes les informations nécessaires pour l'utilisation du module.
- Corps des modules :
  - Spécifier les fonctions privées.
  - Expliquer l'algorithme utilisée quand pas évident.
  - Donner des invariants des fonctions (utiliser des construction du langage (*assertions* si possible)

4 Le langage des modules en OCaml

En vérité : Module $\neq$ Unité de compilation.

Il y a en OCaml des constructions pour définir des modules (à part des modules définis implicitement par unité de compilation).

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

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

```
module IntStack =
  struct
    type stack = int list
    let empty = []
    let push l i = i::l
    let rec somme l = match l with
        [] -> 0
      | h::r -> h+(somme r)
    exception Empty_Stack
    let top l = match l with
        h::r -> h
      | [] -> raise Empty_Stack
    let pop l = match l with
        h::r -> r
      | [] -> raise Empty_Stack
  end
```

Réponse de OCaml :

```
module IntStack :
 sig
   type stack = int list
   val empty : 'a list
   val push : 'a list -> 'a -> 'a list
   val somme : int list -> int
   exception Empty_Stack
   val top : 'a list -> 'a
   val pop : 'a list -> 'a list
 end
```

Les types dans les signatures.

Par défaut : la signature (interface) d’une structure (module) contient tout ce qui est défini par la structure.

On peut définir une nouvelle signature et puis masquer une partie de la structure par la nouvelle signature :

```
module type STACK =
  sig
    type stack
    val push : stack -> int -> stack
    val empty : stack
    exception Empty_Stack
    val top : stack -> int
    val pop: stack -> stack
    val somme : stack -> int
  end
```

Restriction d’une structure par une signature

Sur l’exemple :

```
module Stack = (IntStack : STACK)
```

Attention : deux types abstraits ayant la même implémentation sont incompatibles !

Quand doit-on considérer équivalents deux types ?

On a deux choix :

équivalence structurelle on considère un type `t1` et un type `t2` équivalents si leur *structure* est identique. Sin on fait ce choix, le programme suivant est bien typé :

```
type  rectype = {name:string, age:int} in
let   rec1 = {name="Nobody", age=1000} in
type  rectype' = {name:string, age:int} in
let   rec2=  {name="Somebody", age=2000} in
rec1 = rec2
```

Mais cela demande un effort considérable au compilateur, en particulier si on permet des types récursifs (on sait en décider l'équivalence, mais cela sort du cadre du cours de cette année)

types génératifs

on considère *tous* les types distincts, même s'ils ont la même structure[a]. Cela signifie que chaque nouvelle définition d'un type utilisateur doit être distinguèe de toutes les précédentes.

Pour obtenir cet effet, on associe à chaque définition de type une valeur unique (cela peut être un numero de série, en litérature on parle de "time-stamp"), qui permettra de le distinguer facilement et rapidement des autres.

On parle aussi de types *génératifs*, parce-que chaque déclaration de type produit (génére) une nouvelle valeur unique.

[a] Les langages qui font ça comportenten général une notion d'*abréviation*pour introduire des nouveaux nomspour le même type

Le choix fait dans les langages

Langage	types génératifs	Notes
Ocaml	oui	mais attention pour les modules
C/C++		oui pour structures, union, tableaux non pour le reste
Pascal	oui	sauf pour SET
Ada	oui	
Algol 68	non	le langage non génératif le plus complexe
Modula-3		génératif sur les types abstrait, structurel sur les types concrets

Rappel : deux types abstraits ayant la même implémentation sont incompatibles (générativité) ! Si on veut garder la compatibilité, il faut un mechanisme ad hoc.

Partage de type

```
module Nom1 = (Nom2 : SIG with type t1 = t2 and ...)
```

(voir la demo)