# Modules

## Objectifs

— Manipulation d'interface et de module
— Manipulation de foncteurs

## Réalisations attendues

— Indispensable : Exercices 1 à 4
— Bonus : Exercice 5

## Rappel : tester à l'aide de Utop

— `dune utop` – lance utop
— `open Tp5 ;;` – charge le module `Tp5`
— `EvalSimple.plus 1 1;;` – par exemple pour tester la fonction `plus` du module `EvalSimple`
— `exit 0;;` – pour quitter

## 1 Problème de l'Expression

Le problème de l'expression est un problème classique en programmation, où l'on cherche à permettre une réutilisabilité et une extensibilité maximale du code créé. Ce problème possède deux dimensions : d'une part, on veut pouvoir ajouter facilement des cas à un type de données (typiquement des sous-classes dans le cas de langages à objets) ; d'autre part on veut également pouvoir ajouter des parcours/-traitements sur ces structures. L'idéal serait de ne pas modifier le code existant.

Nous allons aborder une solution avec des modules, appelée "sémantique finale". Le principe est de se débarrasser des constructeurs/données et de les remplacer par des appels de fonctions paramétrables par l'utilisateur, i.e. d'appliquer le principe d'abstraction de code au maximum. Cela fait disparaître tout constructeur et tout *pattern-matching*.

## 2 Application : Expressions arithmétiques

Illustrons tout d'abord la formulation simple de ce problème dans le cas des expressions arithmétiques. Le type des expressions ainsi qu'une fonction d'évaluation seront définis dans la figure **??**.

À partir de cette solution, on peut vouloir deux choses :
— ajouter des traitements (par exemple un affichage, une dérivation symbolique par rapport à une variable, etc).

```
type expr =
| Const of int
| Plus of expr * expr
| Mult of expr * expr

let rec eval e =
 match e with
 | Const c       -> c
 | Plus (e1, e2) -> (eval e1) + (eval e2)
 | Mult (e1, e2) -> (eval e1) * (eval e2)
```

Figure 1 – Le type des expressions et la fonction d'évaluation

— ajouter des cas au type (par exemple les définitions locales de variables) sans devoir modifier le code des fonctions déjà existantes.

Ici, il est facile d'ajouter un nouveau traitement, mais l'ajout de cas impose la modification du code de tous les traitements déjà réalisés. Dans une solution orientée objet classique, la difficulté est inversée : l'ajout de cas se fait facilement, par création de nouvelles sous-classes. Par contre l'ajout de traitements nécessite d'ajouter autant de méthodes dans chacune des sous-classes (qui doivent donc être modifiées). Il existe bien un patron de conception, appelé Visiteur, qui améliore l'extensibilité, mais il ne répond pas entièrement au problème.

# 3 Solution avec sémantique "finale"

## 3.1 Des expressions simples et un unique traitement

La sémantique "finale" consiste à abstraire le type des expressions, avec ses différents constructeurs, en autant de constructeurs "abstraits" (i.e. des fonctions), au besoin répartis dans plusieurs interfaces (pour la modularité).

L'interface ExprSimple (voir figure **??**) permet d'abstraire les expressions.

```
module type ExprSimple =
sig
  type t
  val const : int -> t
  val plus : t -> t -> t
  val mult : t-> t -> t
end
```

Figure 2 – Interface abstrayant les expressions simples

On se propose d'évaluer les expressions simples. Il s'agit ici simplement de donner un sens aux différents éléments des signatures, i.e. de permettre leur évaluation sous la forme d'une évaluation d'expression en l'occurrence.

Le module EvalSimple (voir figure **??**), conforme à ExprSimple, qui permet d'évaluer les expressions.

```
module EvalSimple : ExprSimple with type t = int =
struct
  type t = int
  let const c = c
  let plus e1 e2 = e1 + e2
  let mult e1 e2 = e1 * e2
end
```

Figure 3 – Module d'évaluation des expressions simples

## 3.2 Tests

Nous voulons maintenant définir des expressions et vérifier que notre évaluateur se comporte correctement. Pour cela nous utiliserons des foncteurs.

### Utilisation de foncteurs

1. Il est possible d'utiliser un module paramétré classique pour décrire des expressions quelconques. Par exemple le code permettant de décrire `1+(2*3)` et `(5+2)*(2*3)` est donné dans la figure **??**.

```
module ExemplesSimples (E:ExprSimple) =
struct
  (* 1+(2*3) *)
  let exemple1 = E.(plus (const 1) (mult (const 2) (const 3)) )
  (* (5+2)*(2*3) *)
  let exemple2 = E.(mult (plus (const 5) (const 2)) (mult (const 2) (const 3)) )
end
```

Figure 4 – Foncteur définissant deux expressions : "1+(2*3)" et "(5+2)*(2*3)"

2. Il est ensuite possible d'écrire des tests unitaires pour vérifier que les expressions définies précédemment s'évaluent correctement. Le code dans le figure **??** instancie le module de tests et vérifie que les évaluations donnent le bon résultat.

```
module EvalExemples = ExemplesSimples (EvalSimple)

let%test _ = (EvalExemples.exemple1 = 7)
let%test _ = (EvalExemples.exemple2 = 42)
```

Figure 5 – Tests de l'évaluation des expressions

## 3.3 Ajout d'un traitement

On souhaite maintenant pouvoir afficher les expressions.

▷ **Exercice 1**

1. *Écrire un module* PrintSimple*, conforme à* ExprSimple*, qui permet de convertir les expressions en chaîne de caractères.*
2. *Tester votre module avec les expressions définies précédemment.*

▷ **Exercice 2**

1. *Écrire un module* CompteSimple*, conforme à* ExprSimple*, qui permet de compter les opérations d'une expression.*
2. *Tester votre module avec les expressions définies précédemment.*

### 3.4 Ajout des variables aux expressions

Nous souhaitons maintenant ajouter les variables aux expressions. Il faut donc pouvoir les définir ("let .. = .. in ..") et les utiliser. Dans la version initiale, cela reviendrait à compléter le type expr comme montré dans la figure **??**.

```
type expr =
| Const of int
| Plus of expr * expr
| Mult of expr * expr
| Def of string * expr * expr
| Var of string
```

Figure 6 – Le type des expressions complété avec les variables

▷ **Exercice 3**

1. *Écrire une interface* ExprVar *qui permet d'abstraire la présence de variable dans les expressions. Elle ne doit traiter que la définition et l'utilisation de variable et ne doit pas redéfinir les expressions simples.*
2. *Par inclusion d'interfaces*[1]*, écrire l'interface* Expr *qui permet d'abstraire les expressions dans leur intégralité.*

### 3.5 Impact sur les traitements déjà réalisés

Nous souhaitons maintenant que les traitements réalisés précédemment soient étendus pour pouvoir traiter les expressions avec variables.

▷ **Exercice 4 (Printer)**

1. *Écrire un module* PrintVar*, conforme à* ExprVar*, qui permet de convertir les expressions avec variables en chaîne de caractères.*
2. *Par inclusion de modules*[2]*, écrire un module* Print *qui permet de traiter les expressions dans leur intégralité.*
3. *En utilisant un foncteur, décrire des expressions quelconques, par exemple : "let x = 1+2 in x*3".*

1. voir section **??**
2. voir section **??**

4. *Tester votre module avec les expressions définies précédemment.*
5. *Réaliser des tests de non régression : s'assurer que les expressions simples s'évaluent toujours de la même façon.*

▷ **Exercice 5 (Evaluateur)**

*Dans la version initiale, l'évaluateur s'écrirait comme montré dans la figure* **??**.

```
let rec eval e env=
 match e with
  | Const c        -> c
  | Plus (e1, e2) -> (eval e1 env) + (eval e2 env)
  | Mult (e1, e2) -> (eval e1 env) * (eval e2 env)
  | Def (x,e1,e2) -> eval e2 ((x,eval e1 env)::env)
  | Var x          -> List.assoc x env
```

Figure 7 – L'évaluation des expressions avec variables

*Notons que le type de* eval *est alors* expr -> (string*int) list -> int *soit* expr -> ( (string*int) list -> int).

1. *Hors de tout module, ajouter un type définissant le type de l'environnement d'évaluation.*
2. *Écrire un module* EvalVar*, conforme à* ExprVar*, qui permet d'évaluer les expressions avec variables.*
3. *L'écriture du module précédent à fait apparaître la nécessité de la gestion d'un environnement d'évaluation, non présent dans les expressions simples. Écrire un module* EvalSimpleEnv*, conforme à* ExprSimple*, qui permet d'évaluer les expressions simples, en propageant l'environnement.*
4. *Par inclusion de module, écrire un module* Eval *qui permet d'évaluer les expressions dans leur intégralité.*
5. *Réaliser des tests de non régression : s'assurer que les expressions simples s'évaluent toujours de la même façon.*

# 4 Inclusion d'interfaces et de modules

Il est possible d'inclure une interface (resp. un module) dans une autre interface (resp. un module) à l'aide du mot-clé **include**. Lors de l'inclusion de plusieurs interfaces (resp. modules) les signatures peuvent toutes contenir la même déclaration de type t, une simple inclusion de toutes ces interfaces (resp. modules) introduirait une duplication de la déclaration de t, ce qui est interdit. Pour éviter ceci, on utilise le mécanisme de substitution de type dans les interfaces.

Le mécanisme de substitution de type dans les signatures permet de macro-expanser une déclaration de type par une définition de type. Ainsi, Interface **with type** t:= def est l'interface Interface où l'on a remplacé le type t par la définition def, partout où il apparaît. La déclaration de t disparaît également de l'interface résultat. À titre d'exemple, supposons données les interfaces suivantes :

```
module type I1 = sig type t ... end
module type I2 = sig type t ... end
```

On pourra les inclure dans une même interface en identifiant les deux types t de I1 et I2, à condition de substituer (i.e. remplacer et faire disparaître) l'un des deux types t par l'autre, ce qui donne :

```
module type I1plusI2 =
sig
  include I1
  (* le type t de I2 est defini comme étant egal au type t de I1 *)
  include (I2 with type t := t)
end
```

De la même manière, l'inclusion des deux module suivants M1 et M2 dans un module M1plusM2 est obtenue en masquant un des deux types t :

```
module M1 : I1 =
struct
  type t ...
end
module M2 : I2 =
struct
  type t ...
end
module M1plusM2 : I1plusI2 =
struct
  include M1
  (* le type t de M2 est masqué et doit être égal au type t de M1 *)
  include (M2 : I2 with type t := t)
end
```