1

Ocaml
Pratique et principes.

Plan

1. Le langage
2. Évaluation des programmes
3. Typage des programmes

Pointeurs utiles

– La page de Ocaml
– Les pages de référence : manuel de référence, librairie du noyau et librairies standards.
– le tutorial, les questions fréquentes (FAQ).
– les librairies développées par les utilisateurs.
– Les quelques livres sur Ocaml.

Les notes de cours

Toutes les notes de cours et les exercices corrigés sont en ligne (voir le plan ci-dessus). En voici une archive et une version postscript sans les exercices.

Les notes de cours sont écrites en TeX puis traduites automatiquement en HTML avec HeVeA. Utilisez plutôt la version postscript pour les imprimer. Le rendu de la version HTML est moins bon, mais elle se consulte beaucoup plus facilement. Les exercices corrigés ne figurent que dans la version HTML.

Pour visualiser correctement les symboles mathématiques, quelques réglages de votre brouteur peuvent être nécessaires.

2

Ocaml
Le langage.

Cours	**Exercices**
Le langage Ocaml	
1. Introduction 2. Syntaxe 3. Types et opérations élémentaires 4. Enregistrements et references 5. Entrées sorties 6. Fonctions 7. Variantes et filtrage 8. Exceptions 9. Mise au point 10. Polymorphisme 11. Modules 12. Compilation séparée	1. Mise en oeuvre (a) Mode compilé (b) Mode interprété 2. Mode Emacs 3. Petits exemples (1, 2, 3, 4) 4. La commande cat 5. La Calculette 6. La commande wc 7. Utilisation de la librairie Set 8. Polynômes

Avertissement

Le double but de ce cours est

1. de vous faire découvrir le langage Ocaml et de vous donner de bonnes bases pour que vous puissiez ensuite l'apprendre par vous-même,
2. de vous donner quelques principes sous-jacents à la définition du langage Ocaml (et des langages de programmation en général) qui vous permettant de mieux comprendre Ocaml et plus généralement la notion de langage de programmation.

Ce cours est organisé en trois parties. Dans un premier temps, nous ferrons un survol du langage Ocaml, en deux étapes. Ensuite nous étudierons le typage, en prenant un sous-ensemble du langage Ocaml comme exemple. Enfin, dans la dernière partie, nous verrons comment décrire l'évaluation des programmes, toujours avec un sous-ensemble de Ocaml comme exemple. Nous ne pouvons bien sûr pas entrés dans les détails.

Apprentissage d'un langage

Se fait par la pratique : exercez-vous ! y compris par couper/coller : n'ayez pas peur de regarder les solutions des exercices, de réutiliser du code.

Garder un pointeur sur

- la documentation : le manuel de référence, la librairie du noyau et les librairies standards.
- le tutorial, l'expérience des autres (FAQ).
- les librairies développées par les utilisateurs (voir aussi la page ocaml).

Il existe aussi quelques livres.

`file :/usr/local/remy/cours/index.html`

La petite histoire

Le langage Caml est le fruit de développements récents et continus pour améliorer l'expressivité et la sûreté de langages de programmation.

1978 Robin Milner propose ML comme un Méta-Langage pour décrire des stratégies dans un démonstrateur de théorèmes.
Il apparaît vite comme un langage de programmation à part entière.

1981 Premières implémentations de ML.

1985 Développement de Caml à l'INRIA, de Standard ML à Edimbourg, puis à Bell Labs (New Jersey, USA).

1990 Développement de Caml-Light par Xavier Leroy et Damien Doligez, à l'INRIA.

1995 Ajout d'un compilateur natif et d'un système de module.

1996 Ajout des objets.

2000 Ajout des arguments étiquetés et optionnels.

Domaines d'utilisation du langage Ocaml

Langage d'usage général

Domaines de prédilection

- Calcul symbolique : Preuves mathématiques, compilation, interprétation, analyses de programmes.
- Prototypage rapide.
- Langage de script.
- Programmation distribuée (bytecode rapide).

Enseignement et recherche

- Classes préparatoires.
- Utilisé dans de grandes universités (Europe, US, Japon, etc.).

Industrie

- Startups, CEA, EDF, France Telecom, Simulog,...

Quelques mots clés

Le langage Ocaml est

– *fonctionnel* : les fonctions sont des valeurs de première classe, elles peuvent être retournées en résultat et passées en argument à d'autres fonctions.
– *à gestion mémoire automatique* (comme JAVA) : Un gestionnaire mémoire récupère les données qui ne sont plus accessibles ; l'utilisateur n'a plus à s'en préoccuper.
– *fortement typé* (comme JAVA) : Le typage garantit l'absence d'erreur (de la machine) à l'exécution.
– *avec synthèse de types* : Les types sont facultatifs car ils sont synthétisés par le système.
– *compilé ou interactif*

3 Premiers pas en Ocaml

Le mode compilé

Éditer le source

```
print_string "Bienvenue !\n";;
```

Compiler et exécuter

```
% ocamlc -o bienvenue bienvenue.ml
% ./bienvenue
Bienvenue !
```

Le mode interprété

```
% ocaml
        Objective Caml version 2.04

# print_string "Bienvenue !\n";;
Bienvenue !
- : unit = ()
# let euro x = floor (100.0 *. x /. 6.55957) *. 0.01;;
val euro : float -> float = <fun>
# let baguette = 4.20;;
val baguette : float = 4.2
# euro baguette;;
- : float = 0.64
# exit 0;;
```

Utilisation du mode interprété

Comme une grosse calculette

Pour la mise au point des programmes

Évaluation des phrases du programme en cours de construction (utilisation avec un éditeur)

Comme langage de script...

`% ocaml < bienvenue.ml`	équivalent à la version interactive
`% ocaml bienvenue.ml`	idem, mais suppression des messages

Le mode caml pour Emacs

Installation :

Insérer le contenu de `../emacs/emacs.el` à la fin de votre fichier `~/.emacs`.

Version interactive :

au choix

- Appeler `Ocaml` par la commande `run-caml`
- Ouvrir un fichier `bienvenue.ml`,
  Écrire les phrases dans ce fichier,
  Exécuter la commande `eval-phrase` dans le menu `caml`.

Version compilée

Exécuter la commande `compile` dans le menu `caml`.

Pour pointer sur le source de l'erreur, se positionner sur le message d'erreur et presser la touche `return`.

Les programmes

Un *programme* est une suite de *phrases* :

– définition de valeur	`let` $x = e$
– définition de fonction	`let` $f\ x\ \ldots\ x_n = e$
– définition de fonctions mutuellement récursives	`let` [`rec`] $f_1\ x_1\ \ldots = e_1\ \ldots$ [`and` $f_n\ x_n\ \ldots = e_n$]
– définition de type(s)	`type` $q_1 = t_1 \ldots$ [`and` $q_n = t_n$]
– expression	e

Les phrases se terminent par `;;` optionnel en général.

```
(* Cela est un commentaire (* et ceci un commentaire à
   l'intérieur d'un commentaire *) sur plusieurs lignes *)
```

Les expressions

`-- définition locale`	`let` $x = e_1$ `in` e_2
	(définition locale de fonctions mutuellement récursives)
`-- fonction anonyme`	`fun` $x_1\ \ldots\ x_n$ `->` e
`-- appel de fonction`	`f` $x_1\ \ldots\ x_n$
`-- variable`	x ($M.x$ si x est défini dans M)
`-- valeur construite`	(e_1, e_2)
`dont les constantes`	`1, 'c', "aa"`
`-- analyse par cas`	`match` e `with` p_1 `->` $e_1 \ldots \mid p_n$ `->` e_n
`-- boucle for`	`for` $i = e_0$ [`down`]`to` e_f `do` e `done`
`-- boucle while`	`while` e_0 `do` e `done`
`-- conditionnelle`	`if` e_1 `then` e_2 `else` e_3
`-- une séquence`	$e; e'$
`-- parenthèses`	(e) `begin` e `end`

Remarques

Il n'y a ni instructions ni procédures. Toutes les expressions retournent une valeur. La valeur unité `()` de type `unit` ne porte aucune information (unique valeur possible dans son type).

L'expression e dans les boucles `for` et `while` et dans la séquence doit être de type `unit`.

On peut ignorer le message d'avertissement, mais il est préférable de jeter explicitement le résultat inutile :

– en utilisant une fonction

```
# let ignore x = ();;
ignore : 'a -> unit
# ignore e1; e2 : unit
```

– en utilisant une définition :

```
# let _ = e1 in e2;;
```

Types, constantes et primitives de base

`unit`	`()`	`pas d'opération !`
`bool`	`true false`	`&& || not`
`char`	`'a' '\n' '\097'`	`code chr`
`int`	`1 2 3`	`+ - * / max_int`
`float`	`1.0 2. 3.14`	`+. -. *. /. cos`
`string`	`"a\tb\010c\n"`	`^ s.[i] s.[i] <- c`

Polymorphes

`tableaux`	`[| 0; 1; 2; 3 |]`	`t.(i) t.(i) <- v`
`tuples`	`(1, 2) (1, 2, 3, 4)`	`fst snd`

Un infix entre parenthèses devient préfixe. Par exemple, `(+)` x_1 x_2 est équivalent à x_1 `+` x_2

Tableaux

Les opérations sur les tableaux sont polymorphes : on peut créer des tableaux *homogènes* d'entiers, de booléens, etc.

```
[| 0; 1; 3 |] : int array
[| true; false |] : bool array
```

Les indices de tableaux varient de 0 à $n-1$ où n est la taille.

Les projections sont *polymorphes* : elles opèrent aussi bien sur des tableaux d'entiers que sur des tableaux de booléens.

```
fun x -> x.(0) : 'a array -> 'a
fun t k x = t.(k) <- x : 'a array -> int -> 'a -> unit
```

Les tableaux sont toujours initialisées :

```
Array.create : int -> 'a -> 'a array
```

Le type du tableau sera le type du second argument, qui sera utilisé pour initialiser toutes les cases du tableau.

Les tuples sont hétérogènes, mais leur arité est fixée par leur type :

– une paire `(1, 2) : int * int`, et

– un triplet `(1, 2, 3) : int * int * int` sont incompatibles.

Les projections sont polymorphes sur les tuples de même arité :

```
fun (x, y, z) -> x : ('a * 'b * 'c) -> 'a
```

Les paires ne sont qu'un cas particulier de tuple, pour lequel les deux projections `fst` et `snd` sont pré-définies.

```
fst : 'a * 'b -> 'a
snd : 'a * 'b -> 'b
```

Les fonctions

Les arguments sont passés *sans parenthèses*.

```
let plus x y = x + y : int -> int -> int
let plus (x, y) = x + y : int * int -> int
```

La seconde fonction a un seul argument qui est une paire.

L'application de fonction se fait par juxtaposition des arguments :

```
plus 1 3
```

Les fonctions peuvent être récursives :

```
let rec fact n = if n > 1 then n * fact (n -1) else 1;;
```

Et mutuellement récursives :

```
let rec ping n = if n > 0 then pong (n - 1) else "ping"
and pong n = if n > 0 then ping (n - 1) else "pong";;
```

Les enregistrements

Il faut les déclarer au préalable

```
type monnaie = {nom : string; valeur : float}
let x = {nom = "euro"; valeur = 6.55957}
x.valeur
```

Champs polymorphes

```
type 'a info = {nom : string; valeur : 'a}
fun x -> x.valeur;;
- : 'a info -> 'a
```

L'étiquette `nom` désigne la projection de l'enregistrement `'a info`, le dernier dans lequel elle a été définie.

(L'ancienne projection est devenue inaccessible)

Les structures mutables

Dans les définitions de type enregistrements, on peut déclarer des champs mutables.

```
type personne = {nom : string; mutable âge : int}
```

permet de modifier la première composante, mais pas la seconde.

```
# let p = {nom = "Paul"; âge = 23};;
val p : personne = {nom="Paul"; âge=23}
# let anniversaire x = x.âge <- x.âge + 1;;
val anniversaire : personne -> unit = <fun>
# p.nom <- "Clovis";;
Characters 0-17:
The label nom is not mutable
```

Les références

Le passage d'arguments se fait toujours par valeur.

Il n'existe pas de construction tels que `&x` ou `*x` en C qui retourne l'adresse à laquelle se trouve une valeur ou la valeur qui se trouve à une adresse.

On ne peut pas modifier la valeur d'une variable

Mais, on peut modifier la valeur des champs d'un enregistrement

```
type 'a boîte = { mutable contenu : 'a };;
```

En C	En Ocaml	Raccourci :
`int x = 0;`	`let x = {contenu = 1};;`	`let x = ref 1;;`
`x = x + 1;`	`x.contenu <- x.contenu +1;;`	`x := !x +1;;`

En pratique, peu de valeurs ont besoin d'être ainsi encapsulées.

La ligne de commande

Les arguments passés à un exécutable

Ils sont placés dans le tableau `Sys.argv : string array`

Le nom de la commande est compté.

Exemple :

La commande Unix `echo`

```
for i = 1 to Array.length Sys.argv - 1
do print_string Sys.argv.(i); print_char ' ' done;
print_newline ();;
```

Usage avancé

La librairie `Arg` permet de lire ces arguments plus facilement.

Retourner un résultat

La fonction exit :`int -> unit`

Elle arrête le programme et retourne au système d'exploitation la valeur entière reçue en argument (modulo 256).

Par défaut, une exécution retourne la valeur 0 si elle se termine normalement et une valeur non nulle autrement (une exception n'est pas attrapée par exemple)

L'usage est de retourner 0 pour une évaluation normale, et de réserver les valeurs non nulles pour décrire un comportement anormal.

Exercice 1 *Écrire les commandes Unix* `true` *et* `false` *qui ne font rien la première retourne le code 0, la seconde le code 1.*

Les entrées-sorties

Canaux pré-définis	**Ouverture de canaux**
`stdin : in_channel`	`open_out : string -> out_channel`
`stdout : out_channel`	`open_in : string -> in_channel`
`stderr : out_channel`	`close_out : out_channel -> unit`

Lecture sur stdin

```
read_line : unit -> string
read_int : unit -> int
```

Écriture sur stdout

```
print_string : string -> unit
print_int : int -> unit
```

Les entrées sorties avancées

- Voir la librairie noyau
- Voir la librairie Printf ; par exemple, on écrira comme en C : `Printf.printf "%s année %d\n" "Année" 1999`

4 Bien comprendre le noyau

Les fonctions sont des valeurs

Ocaml est un langage fonctionnel. Cela signifie que les fonctions sont prises au sérieux

En particulier, elles peuvent être retournées en résultat et passées en argument à d'autres fonctions :

```
# let compose f g = fun x -> f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> ('c -> 'b)
# let plus_deux = compose succ succ;;

# let symétrique f x y = f y x;;
val symétrique : (' a -> 'b -> 'c) -> 'b -> 'a -> 'c
```

Les fonctions peuvent être anonymes :

```
(fun x y -> x + y) 1 3
```

Deux notations équivalentes : `let f x = e` ou `let f = fun x -> e`

La puissance des fonctions

La fonction puissance

```
let rec puissance f n =
    if n <= 0 then fun x -> x
    else let g = puissance f (n - 1) in fun x -> f (g x);;
val puissance : (' a -> 'a) -> int -> 'a -> 'a
```

Quelques exemples

```
let plus u v = puissance ( succ ) u v
let fois u v = puissance (( + ) u) v 0
let puis u v = puissance (( * ) u) v 1

let marge = puissance (fun x -> "."^x);;
val marge : int -> string -> string
```

Les variantes

Les types concrets

```
type 'a peut_être = Voilà of ' a | Pas_encore;;
type carte = Carte of int | Valet | Dame | Roi;;
```

Les constructeurs sont toujours capitalisés.

Types récursifs

```
type 'a liste =
  Vide
| Cellule of 'a * 'a liste
```

Les listes de la librairie

'a liste	'a list
Vide	`[ ]`
Cellule (t,r)	`t :: r`

Définition alternative

```
type 'a liste   = Vide | Cellule of 'a contenu
and 'a contenu = {tête : 'a; reste : 'a liste }
```

Filtrage

Les types concrets sont examiné par filtrage :

```
let valeur = fun x ->
  match x with
   | Valet -> 11
   | Dame -> 12
   | Roi -> 13
   | Carte 1 -> 20
   | Carte x -> x
```

Raccourci :

```
let valeur = function
   | Valet -> 11
   | Dame -> 12
   | Roi -> 13
   | Carte 1 -> 20
   | Carte x -> x
```

Les fonctions opérant sur un type récursif sont souvent récursives :

```
let rec longueur = function
  Vide -> 0
| Cellule (tête, reste) -> 1 + longueur reste
```

Sémantique

Lorsqu'un argument est passé à un ensemble de clauses p_1 `->` $e_1 \mid \ldots \mid p_n$ `->` e_n

- la première clause qui filtre l'argument est exécutée, les autres sont ignorées.
- si aucune clause ne filtre l'argument, une exception est alors levé.

La définition est dite incomplète lorsque cela peut se produire. Lorsqu'un tel risque existe, un message d'avertissement est indiqué à la compilation.

Usage Il est préférable du moins dans un premier temps de ne pas écrire de définitions incomplètes.

Les motifs (patterns)

Ils peuvent

- être emboîtés, contenir des produits et des sommes
- ne pas énumérer tous les champs des enregistrements
- lier une valeur intermédiaire avec mot clé `as`
- n'imposer aucune contrainte (représentés par le caractère `_`)

```
function Voila ({valeur = Carte x} as z, _) -> z.nom
```

Une variable ne peut pas être liée deux fois dans le même motif.

Exercice 2 *Définir les listes à "double entrée" ayant un constructeur supplémentaire permettant d'ajouter un élément à la fin d'une autre liste. Écrire une fonction qui transforme une liste double en une liste simple (de la librairie).*

Exercice 3 (La fonction Sigma)

1. *Écrire une fonction* `sigma : ('a -> int) -> 'a list -> int` *telle que l'expression* `sigma f l` *calcule la formule* $\sum_{x \in 1} f(x)$.
2. *Écrire la fonction* `pi : 'a list -> ('a -> int) -> int` *analogue mais en remplaçant la somme par le produit.*
3. *Comment généraliser la fonction en une fonction*

   ```
   fold : (int -> int -> int) ->
         int -> 'a list -> ('a -> int) -> int
   ```

   pour ne pas dupliquer le code ? Donner deux versions de fold, *d'abord une seule fonction globale récursives, puis utiliser une fonction locale récursive auxilliaire en lui passant le minimum d'argument(s).*
4. *Réécrire la fonction* `sigma` *en utilisant une fonction de librairie.*

Les exceptions

Exemple

```
exception Perdu
let rec cherche_la_clé  k = function
    (h, v) :: t -> if h = k then v else cherche_la_clé k t
  | [] -> raise Perdu

let k =
  try cherche_la_clé "Georges"[ "Louis", 14; "Georges", 5;]
  with Perdu -> 10;;
```

Exercice 4 *Ecrire un programme analogue sans utiliser d'exception.*

Syntaxe

– Définition (phrase)	`exception` C [`of` t]
– Levée (expression)	`raise` e
– Filtrage (expression)	`try` e `with` p_1 `->` $e_1 \ldots \mid p_n$ `->` e_n

Remarquer l'analogie avec la construction de filtrage.

Exceptions pré-définis

Exception	Usage	gravité
`Invalid_argument of unit`	accès en dehors des bornes	forte
`Failure of string`	liste vide, retourne le nom de la fonction	moyenne
`Not_found of unit`	fonctions de recherche	nulle
`Match_failure of ...`	Échec de filtrage	fatale
`End_of_file`	Fin de fichier	nulle

Sémantique

- Le type `exn` est un type somme.
- La levée d'une exception arête l'évaluation en cours et retourne une valeur "exceptionnelle" (du type `exn`).
- Une exception ne peut être éventuellement rattrapée que si elle a été protégée par une expression `try` e_1 `with` m
  - si l'évaluation de e_1 retourne une valeur normale celle-ci est retournée sans passer par le filtre.
  - si l'évaluation de e_1 retourne une valeur exceptionnelle, alors celle-ci est passé au filtre m. Si une des clause des p_i `->` e_i filtre l'exception, alors e_i est évaluée, sinon l'exception n'est pas capturée, et la valeur exception est retournée (propagée).
- On peut observer une exception (l'attraper et la relancer) :

```
try f x with Failure s as x -> prerr_string s; raise x
```

Usage des exceptions

Il est préférable de définir de nouvelles exceptions plutôt que de réutiliser les exceptions pré-définies afin d'éviter tout ambiguïté.

On utilise les exceptions pour

- reporter une erreur et interrompre le calcul en cours ;
  Par exemple, l'exception `Match_failure` est levée à l'exécution lorsqu'une valeur n'est pas filtrée. Elle indique le nom du fichier et la position filtre en cause.
- communiquer une valeur de retour rare ou particulière ;
  Par exemple, les exceptions `Not_found` et `End_of_file` sont utilisées pour programmer et leur levées sont tout à fait normales.

Exemple : la commande Unix `cat`

Une version restreinte :

```
try
  while true
  do print_string (read_line ()); print_newline () done
with End_of_file -> ();;
```

Exercice 5 *Écrire la version complète, qui concatène l'ensemble des fichiers dont les noms sont passés en argument ou bien reproduit le flux d'entrée lorsque la commande est appelée sans argument.*

Exercices

Exercice 6 (La calculette) *- Écrire une fonction sigma qui prend une fonction* `f`*, un tableau et un indice et qui calcule* $\sum_{i=j}^{k} f(t.(i))$

- En déduire le calcul de la somme des éléments, des carrés ou la moyenne des éléments d'un sous-tableau.

- Faire un exécutable à partir de l'exercice précédent. Les arguments sont reçus sur la ligne de commande.

- Par défaut le programme calcule la somme. Si le premier argument est la chaîne `-carre` *ou* `-moyenne`*, alors il calcule la somme ou la moyenne des éléments restants.*

Exercice 7 (La commande Unix `wc`) *compte l'ensemble des caractères, mots et lignes de chacun des fichiers passés en arguments ainsi que les totaux respectifs et les imprimes dans la sortie standard.*

La trace

Fonctionne seulement en mode interprété à l'aide de directives

Mise en œuvre

```
#trace nom-de-fonction;;
#untrace nom-de-fonction;;
#untrace_all;;
```

Changer l'imprimeur par défaut

La trace se combine souvent avec l'ajout d'imprimeurs

```
#install_printer  imprimeur;;
#remove_printer imprimeur;;
```

L'argument *imprimeur* est le nom d'une fonction d'impression de type τ `-> unit`. Les valeurs du types τ sont alors imprimées avec le nouvelle fonction.

Le débogueur

Fonctionne seulement en mode compilé.

Permet de faire du pas en pas en avant **et en arrière**, de visualiser l'état de la pile et les valeurs des variables locales.

Mise en œuvre

Compiler avec `ocamlc -g`

Appeler `ocamldebug` *nom-du-programme*

Il existe une interface conviviale pour Emacs : appeler la commande `camldebug` (faire `ESC x camldebug` *nom-du-programme*)

5 Les modules

Polymorphisme

Le typeur infère les types d'un programme. Il existe souvent plusieurs solutions :

```
let identité x = x;;
  : int -> int
  : string -> string
  : 'a * 'a -> 'a * 'a
  : 'a -> 'a
```

Le typeur retourne le type le plus général (il en existe toujours un si le programme est typable).

```
let identité x = x;;
  : 'a -> 'a
```

Les autres se retrouvent par instantiation des variables de type. On dit que l'`identité` est polymorphe.

Support du polymorphisme

La construction de liaison introduit le polymorphisme.

```
let f = identité in f 1, f true;;
```

L'abstraction ne transmet pas le polymorphisme

```
(fun f -> f 1, f true) identité ;;                Erreur
```

est rejeté. Les deux occurrences de `f` doivent avoir le même type.
(Il n'existerait plus toujours de type le plus général)

L'application interrompt également le polymorphisme

```
let f = identité identité in f 1, f true;;        Erreur
```

est également rejeté.
(Une application pourrait produire des effets de bords — lecture et écriture — avec des types différents)

Variables de type "faibles"

Dans le mode interprété, on peut écrire :

```
# let f = identité identité ;;
f : '_a -> '_a
```

Mais la fonction `b` n'est pas polymorphe. Les variables `'_a` ne sont pas des variables polymorphes. Elles sont dites variables de type faible. Elles représentent un type particulier mais pas encore connu. L'expression

```
# f 1;;
```

est correcte, mais elle fige irréversiblement le type de `f` :

```
# f;;
f : int -> int
# f true;;                                        Erreur
```

La programmation traditionnelle

Les programmes sont écrits de façon linéaire.
On peut seulement ajouter de nouvelles définitions à la fin du programme.
Réutilisation du code par "couper-coller".
Pas de partage de code
– duplication du code, et des erreurs !
– difficulté de maintenance.

La programmation modulaire

Découpage du programme en morceaux compilables indépendamment
Rendre les gros programmes compilables

Donner de la structure au programme
Rendre les gros programmes compréhensibles

Spécifier les liens (interfaces) entre les composantes
Rendre les gros programmes maintenables

Identifier des sous-composantes indépendantes
Rendre les composantes réutilisables

Modules de base

Les structures sont collections de phrases

`struct` p_1 ...p_n `end`

Les phrases sont celles du langage de base, plus

définition de sous-modules	`module` $X = \ldots$
définition de type de module	`module type` $S = \ldots$

Le nommage d'un module se fait à l'aide de la liaison `module`.

```
module S =
  struct
    type t = int
    let x = 0
    let f x = x+1
  end
```

Utilisation d'un module

On fait référence aux composantes d'un module avec la notation pointée *module.composante*

```
... ( S.f S.x : S.t) ...
```

Autre possibilité : la directive `open` *module* permet d'omettre le préfixe et le point :

```
open S
... ( f x : t) ...
```

Modules emboîtés

Un module peut être composante d'un autre module :

```
module T =
  struct
    module R = struct let x = 0 end
    let y = R.x + 1
  end
```

La notation pointée et le `open` s'étendent naturellement et peuvent apparaître dans le corps d'autres modules :

```
module Q =
  struct
    let z = T.R.x
    open T.R
    ...
  end
```

NB : La notation `open T.R` *rend les composantes de* `T.R` *visibles dans la suite du corps de* `Q` *mais n'ajoute pas ces composantes à "Q".*

Les types des modules de base

Les *signatures* : collections de spécifications (de types).

`sig` *spécification* ...*spécification* `end`

spécification de valeurs	`val` $x : \sigma$
spécification de types abstraits	`type` t
spécification de types manifestes	`type` $t = \tau$
spécification d'exceptions	`exception` E
spécification de sous-modules	`module` $X : M$
spécification de type de module	`module type` $S\ [= M]$

Nommage d'un type de module : par la liaison `module type`

```
module type MA_SIGNATURE = sig ... end
```

Le système infère les signatures des modules (comme il infère les types des valeurs).

Module

```
module Exemple =
  struct
    type t = int
    module R =
      struct
        let x = 0
      end
    let y = R.x + 1
  end;;
```

Signature inférée

```
module Exemple :
  sig
    type t = int
    module R :
      sig
        val x : int
      end
    val y : int
  end
```

La construction (*structure* : *signature*)

- vérifie que la structure satisfait la signature (toutes les composantes spécifiées dans la signature doivent être définies dans la structure, avec des types au moins aussi généraux) ;
- rend inaccessibles les parties de la structure qui ne sont pas spécifiées dans la signature ;
- produit un résultat qui peut être lié par `module M =` ...

Sucre syntaxique :

`module M :` *signature* `=` *structure*

est équivalent à

`module M = (`*structure* `:` *signature*`)`

Exemple de restriction

Écritures équivalentes

```
module S =
  (struct
      type t = int
      let x = 1
      let y = x + 1
   end :
   sig
      type t
      val y : t
   end);;
```

```
module type SIG =
   sig
      type t
      val y : t
   end
module S : SIG =
  struct
      type t = int
      let x = 1
      let y = x + 1
   end;;
```

```
S.x;;            Erreur " S.x is unbound"
S.y + 1;;        Erreur " S.y is not of type int"
```

Il est parfois important de distinguer des types isomorphes. Par exemples Euros et Dollars sont tous deux représentés par des flottants. Pourtant, il ne faut pas les confondre.

Ce sont deux espaces vectoriels, isomorphes mais disjoints, avec pour unités respectives l'`euro` et le `dollar`.

```
module Float =
  struct
    type t = float
    let un = 1.0
    let plus = (+.)
    let prod = ( *. )
  end;;
```

```
module type MONNAIE =
  sig
    type t
    val un  : t
    val plus : t -> t -> t
    val prod : float -> t -> t
  end;;
```

La multiplication devient une opération externe sur les flottants.

Dans `Float` le type `t` est concret donc il peut être confondu avec "float". Par contre, il est abstrait dans les modules `Euro` et `Dollar` ci-dessous :

```
module Euro = (Float : MONNAIE);;
module Dollar = (Float : MONNAIE);;
```

Les types `Euro.t` et `Dollar.t` sont isomorphes mais incompatibles.

```
let euro x = Euro.prod x Euro.un;;
Euro.plus (euro 10.0) (euro 20.0);;
Euro.plus (euro 50.0) Dollar. un;;        Erreur
```

Pas de duplication de code entre `Euro` et `Dollar`.

On peut donner une interface restreinte pour, dans un certain contexte, ne permettre que certaines opérations (typiquement, interdire la création de valeurs) :

```
module type PLUS =
  sig
    type t
    val plus : t -> t -> t
  end;;
module Plus = (Euro : PLUS)
```

```
module type PLUS_Euro =
  sig
    type t = Euro.t
    val plus : t -> t -> t
  end;;
module Plus = (Euro : PLUS_Euro)
```

À gauche, le type `Plus.t` est incompatible avec `Euro.t`.

À droite, le type `t` est partiellement abstrait et compatible avec "Euro.t", et la vue `Plus` permet de manipuler les valeurs construites avec la vue `Euro`.

La notation `with`

La notation `with` permet d'ajouter des égalités de types dans une signature existante.

L'expression `PLUS with type t = Euro.t` est une abréviation pour la signature

```
sig
  type t = Euro.t
  val plus: t -> t -> t
end
```

On peut alors écrire

```
module Plus = (Euro : PLUS with type t = Euro.t);;
Plus.plus Euro.un Euro.un;;
```

Elle permet de créer facilement des signatures partiellement abstraites.

Modules et compilation séparée

Une unité de compilation A se compose de deux fichiers :

- Le fichier d'implémentation A`.ml` :
  une suite de phrases
  semblable à l'intérieur de `struct...end`
- Le fichier d'interface A`.mli` (optionnel) :
  une suite de spécifications
  semblable à l'intérieur de `sig...end`

Une autre unité de compilation B peut faire référence à A comme si c'était une structure, en utilisant la notation pointée A`.x` ou bien en faisant `open` A.

Compilation séparée d'un programme

Fichiers sources : `a.ml`, `a.mli`, `b.ml`

Étapes de compilation :

`ocamlc -c a.mli`	compile l'interface de `A`	crée `a.cmi`
`ocamlc -c a.ml`	compile l'implémentation de `A`	crée `a.cmo`
`ocamlc -c b.ml`	compile l'implémentation de `B`	crée `b.cmo`
`ocamlc -o monprog a.cmo b.cmo`	édition de liens finale	

Le programme se comporte comme le code monolithique :

```
module A =
  (struct contenu de a.ml end : sig contenu de a.mli end)
module B = struct contenu de b.ml end
```

L'ordre des définitions de modules correspond à l'ordre des fichiers objets `.cmo` sur la ligne de commande de l'éditeur de liens.

Modules paramétrés

Un *foncteur* est une fonction des modules dans les modules :

`functor (`S `:` *signature*`) ->` *module*

Le *module* (corps du foncteur) est explicitement paramétré par le paramètre de module S. Il fait référence aux composantes de son paramètre avec la notation pointée.

```
module T = functor(S : SIG) ->
  struct
    type u = S.t * S.t
    let y = S.g(S.x)
  end
```

Application de foncteur

On ne peut pas directement accéder dans `T`. Il faut au préalable l'appliquer explicitement à une implémentation de la signature `SIG` (tout comme on applique une fonction ordinaire).

```
module T1 = T(S1)
module T2 = T(S2)
```

`T1`, `T2` s'utilisent alors comme des structures ordinaires :

```
(T1.y : T2.u)
```

`T1` et `T2` partagent entièrement leur code.

Exemple (long) d'un compte bancaire

```
module type BANQUE =        (* vue du banquier *)
  sig
    type t
    type monnaie
    val créer : unit -> t
    val dépôt : t -> monnaie -> monnaie
    val retrait  : t -> monnaie -> monnaie
  end;;

module type CLIENT =        (* vue donnée au client *)
  sig
    type t
    type monnaie
    val dépôt : t -> monnaie -> monnaie
    val retrait  : t -> monnaie -> monnaie
  end;;
```

Une modélisation simple de la banque

Sur le modèle des actions, le compte est donné au client... de façon abstraite bien sûr (sa représentation est cachée) afin que seule la banque puisse modifier le compte du client.

```
module Banque_au_porteur (M : MONNAIE) :
    BANQUE with type monnaie = M.t =
  struct
    type monnaie = M.t
    type t = { mutable solde : monnaie }
    let zéro = M.prod 0.0 M.un and neg = M.prod (-1.0)

    let créer () = { solde = zéro }
    let dépôt c x =
      if x > zéro then c.solde <- M.plus c.solde x; c.solde
    let retrait  c x =
      if c.solde > x then dépôt c (neg x) else c.solde
  end;;

module Poste = Banque_au_porteur (Euro);;
```

La modularité dans l'exemple

– Les clients et le banquier ont des vues différents de la banque.

```
module Client : CLIENT
        with type monnaie = Poste.monnaie
        with type t = Poste.t
   = Poste;;

let mon_ccp = Poste.créer ();;
Poste.dépôt mon_ccp (euro 100.0);;
Client . dépôt mon_ccp (euro 100.0);;
```

– On peut créer des comptes dans différentes devises sans risque de confusion (elles seront détectées par le typage).

```
module Citybank = Banque_au_porteur (Dollar);;
let mon_compte_aux_US = Citybank.créer()
Citybank.dépôt mon_ccp;;                        Erreur
Citybank.dépôt mon_compte_aux_US (euro 100.0);; Erreur
```

Modularité (suite)

– On peut changer l'implémentation de la banque tout en en préservant l'interface.

Pour éviter la fraude, une banque donne seulement au client un numéro de compte et conserve l'état de son compte en interne.

La représentation d'un compte devient un simple numéro.

– Plusieurs banques européennes ont alors des banques de données indépendantes (protégées).

```
module Banque_centrale = Banque_au_porteur (Euro);;
module Poste = Banque_au_porteur (Euro);;
```

Exercice 8 (Polynômes à une variable)

1. *Réaliser une bibliothèque fournissant les opérations sur les polynômes à une variable. Les coefficients forment un anneau passé en argument à la bibliothèque.*
2. *Utiliser la bibliothèque pour vérifier par exemple l'égalité* $(1+X)(1-X) = 1-X^2$.
3. *Vérifier l'égalité* $(X+Y)(X-Y) = (X^2-Y^2)$ *en considérant les polynômes à deux variables comme polynôme en* X *à coefficients les polynôme en* Y.
4. *Écrire un programme qui prend un polynôme sur la ligne de commande et l'évalue en chacun des points lus dans* `stdin` *(un entier par ligne). Le résultat est imprimé dans* `stdout`.

Objets et classes

Pour mémoire. Voir le tutorial

6

Ocaml
Évaluation des programmes .

Cours	**Exercices**
Évaluation	
1. Introduction 2. Expressions arithmétiques 3. Noyau ML 4. Paires 5. Références	1. Expressions arithmétiques (eval, variables, eval') 2. Règles d'erreur 3. Paires 4. Conditionnelle

Un peu de recul

Questions

Qu'est-ce qu'un langage ? Qu'est-ce qu'un langage de programmtion ? Sont-ils tous équivalents ? Comment les comparer ?

Langages de programmations

Les langages de programmation ne sont que des approximations, rigoureuses mais incomplètes, du langage mathématique. Les langages généraux (non spécialisés à certains types de problèmes), sont complets au sens de Turing. Cela veut dire qu'ils permettent d'écrire tous les algorithmes calculables. Sinon, ils ne seraient pas intéressants. Par conséquent, il n'est pas possible de prévoir mécaniquement de la terminaison d'un programme.

Les langages de programmation ne sont cependant pas tous équivalents, car ils diffèrent énormément par leur expressivité, c'est-à-dire leur capacité d'exprimer succinctement certains algorithmes. Ce qui conduit à une recherche —sans fin ?— de nouvelles structures de programmation, plus expressives, permettant de décrire des algorithmes de façon plus concise.

Mais l'expressivité n'est pas a rechercher à tout prix. En particulier, il faut se limiter à des constructions essentielles, simples, bien formalisés et mais aussi sures.

En particulier La sûreté de l'exécution ne doit évidemment pas être abandonnée au profit de l'expressivité. Pour cela, on se limite souvent à une sous-classe (décidable) des programmes qui s'évaluent normalement.

Syntaxe concrète et syntaxe abstraite

En informatique, on distingue la syntaxe concrète (phrases composée d'une suite de caractères ou de lexèmes) définissant un programme et sa syntaxe abstraite qui est une représentation informatique/mathématique de la syntaxe, en générale sous une forme arborescente.

Par exemple, la phrase `let f = fun x -> x` est la syntaxe concrète d'un (bout) de programme. La syntaxe abstraite associé sera, par exemple, l'arbre

```
Liaison ( "f", Fonction ("x", Variable "x"))
```

Syntaxe et sémantique

Donner la syntaxe (concrète ou abstraite) d'un langage n'est qu'un préliminaire à sa définition. Pour parler de programme il faut donner un sens aux expressions du langage, c'est-à-dire définir leur sémantique. Par exemple, cela peut se faire en décrivant l'évaluation des expressions.

La sémantique *dénotationelle* est une sémantique abstraite. C'est la donnée d'une *fonction* qui associe à chaque programmme un objet mathématique appartenant à un domaine sémantique (un ensemble défini pour cette occasion).

Par exemple, la sémantique des expressions arithmétiques (sans variables) peut être définie en associant à chaque arbre de syntaxe abstraite représentant une expression un entier naturel correspondant à la valeur de cette expression.

La sémantique dite *opérationnelle* est plus concrète, en particulier les valeurs résultats de l'évalutaion sont des objets syntaxiques très proche des programmes, voir un sous-ensemble des programmes. La sémantique opérationnelle n'explique donc pas ce qu'est un programme, mais elle décrit précisément comme l'évaluer.

Exemple des expressions arithmétiques

Pour simplifier on se contentera des sommes et produits :

$a ::=$		Expression
	c	Constante entière
	$a + a$	Somme
	$a * a$	Produit

On peut représenter la syntaxe abstraite en Ocaml par une variante :

```
type exp =
  | Const of int
  | Somme of exp * exp
  | Produit of exp * exp;;
```

Par exemple, l'expression concrète $2 * (3 + 4)$ est représentée par :

```
let e = Produit (Const 2, Somme (Const 3, Const 4));;
```

Exercice 9 *Écrire une fonction* `eval : exp -> int` *qui prend une expression arithmétique et retourne sa valeur.*

Ajout des variables

On ajoute des variables aux expressions arithemétiques

$a ::=$	$\ldots$	
	x	Variable

```
type exp =
    ...
  | Var of string
let ex = Somme (Const 2, Produit (Var "x", Const 3));;
```

Exercice 10 *Écrire une fonction* variables : exp −> string Set.t *qui calcule l'ensemble des variables d'une expression arthmétique.*

On peut évaluer les expressions avec variables dans un environnement (liste de paires variables-entiers) qui donne des valeurs aux variables.

Exercice 11 *Écrire une fonction* eval : (string ∗ int) list −> exp −> int *qui prend un environnement, une expression et retourne le résultat de cette expression dans l'environnement.*

Les expressions du noyau de ML

Les constantes

La syntaxe est définie par rapport à un ensemble de constante entières $c \in \mathcal{C}$ et un ensemble de primitives données avec leur arité $f^n \in \mathcal{F}^n$. On notes **t** un tuple d'éléments de type t.

Un programme est une expression composée de l'une des formes suivantes :

$a ::=$		Expression
	c	Const
	$f\ \overline{a}$	Primitive
	x	Variable
	$a\ a$	Application
	$\texttt{fun}\ x \rightarrow a$	Fonction
	$\texttt{let}\ \ x = a_1\ \texttt{in}\ a_2$	Liaison

En Ocaml, on définit :

```
type prim = {name : string; arity : int }
let succ = {name ="succ"; arity = 1}
let plus = {name = "plus"; arity = 2}

type exp =
  | Const of int
  | Prim of prim * exp list
  | Var of var
  | App of exp * exp
  | Fonction of var * exp
  | Liaison of var * exp * exp
and var = string;;
```

Exemple de programme :

```
let e1 =
  Liaison
    ("f", Fonction ("y", Prim (plus, [ Var "y"; Const 3 ])),
     App (Var "f", Const 2));;
```

Intuitivement l'évaluation de l'expression e1 doit valoir 3 ? Quels sont les autres formes de valeurs possibles ?

Les valeurs de ML

Ce sont les constantes entières, mais aussi les fonctions (pas ou partiellement appliquées). L'évaluation du corps d'une fonction est en effet gelé jusqu'à ce que tous les arguments soient fournis.

Cependant une fonction peut comporter des variables libres (liées dans l'environnement d'évaluation). Traiter cette fonction comme une valeur permet de déplacer cette fonction et de l'utiliser dans un autre contexte d'évaluation. Afin de préserver la liaison (statique) des variables, une fonction est évaluée en une *fermeture*, i.e. en couplant la fonction avec l'environnement courrant d'évaluation.

Exercice 12 *On considère le programme Ocaml*

```
Liaison
 ("f", Liaison ( "x", Const 1,
                 Fonction ("y", Prim (plus, [ Var "x"; Var "y"]))) 
  Liaison ( "x", Const 2, App (Var "f", Var "x")));;
```

Vérifier que cette expression n'est pas équivalente à

```
Liaison
  ("x", Const 2,
   App (Fonction ("y", Prim (plus, [Var "x"; Var "y"])), Var "x"));;
```

obtenue en remplaçant Var "f" par Fonction ("y", Prim (plus, [Var "x"; Var "y"])).

Formellement :

$$
\begin{array}{lll}
v ::= & & \text{Valeurs} \\
 & c & \text{Constante entière} \\
 & \langle \texttt{fun}\ x \to a, \rho \rangle & \text{Fermeture}
\end{array}
$$

où ρ est un environement d'évuation, *i.e.* de la forme $x_1 \mapsto v_1, ... x_k \mapsto v_k$. En Ocaml,

```
type valeur =
  | Vconst of int
  | Vfonction of var * exp * env
and env = (var * valeur) list ;;
```

Les erreurs

Le résultat de l'évaluation peut être erroné.

$$
\begin{array}{lll}
r ::= & & \text{Résultats} \\
 & v & \text{Valeur} \\
 & Err & \text{Erreur}
\end{array}
$$

Formellement un résultat est un élément d'un type variante. Cependant en Ocaml, on traitera les erreurs comme des exceptions et on retournera une valeur dans le cas normal.

```
type error =
  | Libre of string
  | Delta of prim * valeur list
  | Beta of exp
exception Error of error
let error e = raise (Error e);;
```

L'évaluation d'un programme est paramétrée par l'évaluation des primitives. On se donne donc pour chaque primitive d'arité n une fonction d'évaluation de même arité, partiellement définie.

```
let delta_succ   [Vconst x] = Vconst (x + 1)
let delta_plus   [Vconst x; Vconst y] = Vconst (x + y)
let delta =
  [ "succ", delta_succ;  "plus", delta_plus ; ] ;;
```

Règles d'évaluations

On décrit la sémantique opérationnelle par une relation $\rho \vdash a \Rightarrow r$ qui se lit *dans l'environnement* ρ*, le programme* a *s'évalue en le résultat* v.

Cette relation est définie par des règles d'inférences de la forme

$$\frac{P_1 \qquad \dots \qquad P_k}{C}$$

qui signifie que la conclusion C est vrai si les prémisse P_1, ... P_k sont vraies simultanément.

Const
$$\rho \vdash n \Rightarrow n$$

Var
$$\rho \vdash x \Rightarrow \rho(x)$$

Fun
$$\rho \vdash \texttt{fun}\ x \rightarrow a \Rightarrow \langle \texttt{fun}\ x \rightarrow a, \rho \rangle$$

App
$$\frac{\rho \vdash a_1 \Rightarrow \langle \texttt{fun}\ x \rightarrow a, \rho_0 \rangle \rho \vdash a_2 \Rightarrow v \qquad \rho_0, x \mapsto v \vdash a \Rightarrow w}{\rho \vdash a_1\ a_2 \Rightarrow w}$$

Prim
$$\frac{\rho \vdash a_i \Rightarrow v_i \qquad \delta(f, \overline{v}) = w}{\rho \vdash f\ \overline{a} \Rightarrow w}$$

Règles d'erreurs

En fait, il faut compléter les règles de bonne évaluation par des règles d'erreurs.

Var-Libre
$$\frac{x \notin dom\ \rho}{\rho \vdash x \Rightarrow Err}$$

App-Err-Left
$$\frac{\rho \vdash a_1 \Rightarrow r \qquad r \not\equiv \langle \texttt{fun}\ x \rightarrow a, \rho_0 \rangle}{\rho \vdash a_1\ a_2 \Rightarrow Err}$$

App-Err-Right
$$\frac{\rho \vdash a_1 \Rightarrow \langle \texttt{fun}\ x \rightarrow a, \rho_0 \rangle \qquad \rho \vdash a_2 \Rightarrow Err}{\rho \vdash a_1\ a_2 \Rightarrow Err}$$

Prim-Right
$$\frac{\rho \vdash a_i \Rightarrow Err}{\rho \vdash f\ \overline{a} \Rightarrow Err}$$

Prim-Left
$$\frac{\rho \vdash a_i \Rightarrow v_i \qquad \delta(f, \overline{v}) \text{ undefined}}{\rho \vdash f\ \overline{a} \Rightarrow Err}$$

En fait les règles d'erreurs sont plus nombreuses que les règles de bonnes évaluations. Elles décrivent à la fois comment les erreurs se déclanchent (par exemple *Var-Libre*) et comment elles se propagent (par exemple *App-Err-Right*.

Les règles d'erreur forcent également un ordre d'évaluation : ici l'ordre gauche-droite est imposé par la combinaison de *App-Err-Left* et *App-Err-Right*, puisque pour appliqué *App-Err-Right* il faut que la partie gauche s'évalue correctement.

Implémentation

Une fois l'ordre d'évaluation imposé (par les règles d'erreurs), la relation d'évaluation devient une fonction. On peut alors lire les règles presque comme une définition récursive par pattern matching (sur la forme du programme).

- Deux conclusions de formes différentes sont deux cas différents dans la fonction eval.
- Deux conclusions de la même forme (cas de l'application) corresponde à un même cas, les sous-expressions sont évaluées puis selon la forme de résultat l'une ou l'autre règle s'applique.

```
let find x env =
  try List.assoc x env with Not_found -> error (Libre x);;

let rec eval env = function
  | Const c -> Vconst c
  | Var x as vx -> find x env
  | Fonction (x, a) as f -> Vfonction (x, a, env)
  | App (a1, a2) as e ->
      let f = eval env a1 in
      let v = eval env a2 in
      begin match f with
      | Vfonction (x, a, env0) -> eval ((x, v) :: env0) a
      | Vconst _ -> error (Beta e)
      end
  | Prim (f, args) as e ->
      let vargs = List.map (eval env) args in
      begin try (List.assoc f.name delta) vargs
      with x -> error (Delta (f, vargs))
      end
  | Liaison (x, e1, e2) as e ->
      eval env (App (Fonction (x, e2), e1)) ;;

let v1 = eval [] e1;;
```

L'évaluation d'un programme

Trois cas se présentent :

- Dans le cas idéal, l'évaluation retourne une valeur.
- L'évaluation produit une erreur.
- L'évaluation peut aussi "ne pas terminer".

  Formellement, la seule chose que l'on peut dire est que pour un environnement ρ et un programme certain a il n'existe pas de réponse r telle que $a\rho \vdash a \Rightarrow r$. On note alors $\rho \vdash a \uparrow$.

  Toutefois, la divergence ne correspond à la non terminaison au sens habituel que si les règles d'erreurs sont correctement définie (et comportent sufissement de cas). En effet, supprimer une règle d'erreur revient à chacun une erreur en une divergence.

Ajout des paires

De nombreuses constructions peuvent être ajoutées en suivant le même modèle. (il faut toutefois l'enrichir).

Pour modéliser l'ajout des paires, on ajoute une notion de valeurs construites qui généralise les entiers à des constructeurs pouvant prendre des arguments.

$$
\begin{array}{rlll}
a ::= & & & \text{Expression} \\
 & \ldots & & \text{comme avant, sauf} \\
 & c^k\ \overline{a}^k & \text{Const} & \\
c^0 ::= & 0, 1, 2, \ldots & & \text{Entiers} \\
c^2 ::= & (\cdot, \cdot) & & \text{Paires}
\end{array}
$$

En Ocaml

```
type constr = Int of int | Constr of string
type const = {constr : constr; carity : int}
    ...
  | Const of constructeur * exp list
```

Il faut aussi étendre les valeur de façon similaires :

$$
\begin{array}{rll}
v ::= & & \text{Valeurs} \\
 & \ldots & \text{comme avant, sauf} \\
 & c^k\ \overline{v}^k & \text{Const} \\
c^0 ::= & 0, 1, 2, \ldots & \text{Entiers} \\
c^2 ::= & (\_, \_) & \text{Paires}
\end{array}
$$

Puis ajouter les règles d'évaluation

$$
\textsc{Paire}\ \frac{\rho \vdash a_1 \Rightarrow v_1 \qquad \rho \vdash a_2 \Rightarrow v_2}{\rho \vdash (a_1, a_2) \Rightarrow (v_1, v_2)}
$$

Exercice 13 *Ajouter toutes les règles d'erreurs pour imposer une évaluation gauche-droite.*

Exercice 14 *Modifier l'évaluateur pour prendre en compte les paires.*

Ajout d'une conditionnelle

$$
\begin{array}{rll}
a ::= & & \text{Expression} \\
 & \ldots & \text{comme avant, plus} \\
 & \texttt{ifz}\ a_0\ \texttt{then}\ a_1\ \texttt{else}\ a_2 & \text{Conditionnelle}
\end{array}
$$

$$
\textsc{Ifz-True}\ \frac{\rho \vdash a_0 \Rightarrow 0 \qquad \rho \vdash a_1 \Rightarrow v}{\rho \vdash \texttt{ifz}\ a_0\ \texttt{then}\ a_1\ \texttt{else}\ a_2 \Rightarrow v}
\qquad
\textsc{Ifz-False}\ \frac{\rho \vdash a_0 \Rightarrow n \qquad n \neq 0 \qquad \rho \vdash a_2 \Rightarrow v}{\rho \vdash \texttt{ifz}\ a_0\ \texttt{then}\ a_1\ \texttt{else}\ a_2 \Rightarrow v}
$$

Exercice 15 (Ajout d'une conditionnelle) *Compléter l'évaluteur du noyau de ML avec la contruction de test à zéro* `ifz` a_0 `then` a_1 `else` a_2.

Ajout de l'affectation

Dans le noyau précédent, une variable a une valeur donnée une fois pour toute. Plusieurs évaluations de la même expression retourneront toujours la même valeur. C'est souvent un avantage (plus grande sécurité de la programmation), mais parfois un inconvénient. Par exemple comment peut-on implémenter un compteur qui retourne un nouvel entier à chaque appel ?

Pour cela, il faut se donner, comme en Ocaml, des objects mutables, le cas le plus simple étant la cellule de référence.

$a ::=$		Expression
	$\ldots$	comme avant, sauf
	$\texttt{ref}\ a$	Référence
	$!a$	Référence
	$a := a$	Référence

En Ocaml,

```
type exp =
    ...
  | Ref of exp
  | Deref of exp
  | Assign of exp * exp
```

Pour décrire la sémantique avec des cellules mutables, on suit l'intuition : une cellule est une location (adresse) mémoire à laquelle on peut écrire ou lire.

$v ::=$		Valeurs
	$\ldots$	comme avant, sauf
	$\texttt{loc}\ m$	Location mémoire

La mémoire est une fonction qui associe des valeurs à des locations-mémoire. La relation sémantique $\sigma/\rho \vdash a \Rightarrow v/\sigma'$ prend alors deux paramètres de plus, la mémoire en entrée σ et la mémoire modifiée après l'évaluation de l'expression σ'.

Règles pour les constructions d'affectation

$$\textsc{Ref}\quad \frac{\sigma/\rho \vdash a \Rightarrow v/\sigma' \qquad m \notin dom\,(\sigma')}{\sigma/\rho \vdash \texttt{ref}\ a \Rightarrow \texttt{loc}\ m/\sigma', m \mapsto v}$$

$$\textsc{Deref}\quad \frac{\sigma/\rho \vdash a \Rightarrow \texttt{loc}\ m/\sigma' \qquad \sigma(m) = v}{\sigma/\rho \vdash !a \Rightarrow v/\sigma'}$$

$$\textsc{Assign}\quad \frac{\sigma/\rho \vdash a_1 \Rightarrow \texttt{loc}\ m/\sigma_1 \qquad \sigma_1/\rho \vdash a_2 \Rightarrow v/\sigma_2 \qquad m \in dom\,(\sigma_2)}{\sigma/\rho \vdash a_1 := a_2 \Rightarrow v/(\sigma_2 \setminus m, m \mapsto v)}$$

Plus les règles d'erreurs...

Par ailleurs, il faut modifier les anciennes règles : bien qu'elle ne modifie pas la mémoire directement, elles peuvent le faire indirectement, par exemple en évaluant des sous-expressions.

Exemple de réécriture :

$$\textsc{App}\quad \frac{\begin{array}{c}\sigma/\rho \vdash a_1 \Rightarrow \langle \texttt{fun}\ x \to a, \rho_0\rangle/\sigma_1 \\ \sigma_1/\rho \vdash a_2 \Rightarrow v/\sigma_2 \qquad \sigma_2/\rho_0, x \mapsto v \vdash a \Rightarrow w/\sigma_3\end{array}}{\sigma/\rho \vdash a_1\ a_2 \Rightarrow w/\sigma_3}$$

Cette ré-écriture fixe également l'ordre d'évaluation.

En ocaml

Pour la mise en œuvre en Ocaml, on pourrait suivre les règles mot à mot, mais on va plutôt profiter des références du langage hôte, ce qui revient à délocaliser la mémoire σ. Aussi on ajoute simplement au valeurs

```
type valeur = ... | Loc of valeur ref;;
let loc v = Loc (ref v);;
```

Puis on modifie la fonction d'évaluation :

```
type error = ...
  | Loc of exp

let rec eval = ...
  | Ref a -> loc (eval env a)
  | Deref a as e ->
      let v = eval env a in
      begin match v with Loc l -> !l | _ -> error (Loc e) end
  | Assign (a1, a2) ->
      let v1 = eval env a1 in
      begin match v with
        Loc l -> l := eval env a2
      | _ -> error (Loc e)
      end ;;
```

Récursion

Le noyau ML défini ci-dessus ne permet pas d'écrire directement des programmes récursifs.

On peut le faire de 3 façons :

- Directement, en ajoutant une construction pour cela.
- Indirectement, en utilisant des fonctions.
- Indirectement, en utilisant des références.

La récursion directement

On peut ajouter une nouvelle construction `funrec` $f\ x \to a$ qui s'évalue vers un nouveau type de valeur $\langle \texttt{funrec}\ f\ x \to a, \rho \rangle$

Fun

$$\rho \vdash \texttt{funrec}\ f\ x \to a \Rightarrow \langle \texttt{funrec}\ f\ x \to a, \rho \rangle$$

App

$$\frac{v_1 \equiv \langle \texttt{funrec}\ f\ x \to a, \rho_0 \rangle \qquad \begin{array}{c} \rho \vdash a_1 \Rightarrow v_1 \\ \rho \vdash a_2 \Rightarrow v \end{array} \qquad \rho_0, f \mapsto v_1, x \mapsto v \vdash a \Rightarrow w}{\rho \vdash a_1\ a_2 \Rightarrow w}$$

L'opérateur de point fixe

On peut définir l'opérateur :

$$fix \equiv \texttt{fun}\ b \rightarrow (\texttt{fun}\ f\ x \rightarrow b\ (f\ f)x)\ (\texttt{fun}\ f\ x \rightarrow b\ (f\ f)x)$$

alors

$$\texttt{let}\quad \texttt{rec}\ f = \texttt{fun}\ x \rightarrow a_1\ \texttt{in}\ a_2 \quad \equiv \quad \texttt{let}\quad f = fix\ (\texttt{fun}\ f\ x \rightarrow a_1)\ \texttt{in}\ a_2$$

Vérification : comme ce programme n'est typable qu'avec des types récursifs.

$ ocaml -rectypes

```
let fix b = (fun f x -> b (f f) x) (fun f x -> b (f f) x);;
let fact = fix (fun fact n -> if n > 1 then n * fact (n-1) else 1);;
fact 5 = 120;;
```

7

Ocaml
Typage des programmes.

Cours	Exercices
Typage	
1. Erreurs d'évaluation 2. Typage 3. Polymorphisme	1. Évaluation bloquée 2. Indécidabilité 3. Variables libres (1, 2, 3) 4. Substitutions (1, 2) 5. Composition de substitutions 6. Inférence, primitives

Erreurs d'évaluation

L'exécution d'un programme peut ne pas produire le résultat attendu par le programmeur. Hormis une erreur d'implémentation, c'est dans tous les cas une erreur du programmeur. On peut classe les erreurs en différentes catégories.

1. Une erreur bête, une typographie, une étourderie, etc.
2. une erreur de raisonnement, d'algorithmique (le programmeur a fait une fautre dans la conception de son programme).
3. une erreur sémantique : le programmeur peut avoir mal compris la sémantique d'une construction du langage.

Le classement ci-dessus est par fréquence (par exemple, le troisième type d'erreur est extrêmement rare, sauf chez les débutant...)

Indépendemment de l'orginie de l'erreur, on peut observer la manisfestation de l'erreur.

– L'évaluation est bloquée (produit une erreur)
– Le résultat retourne une valeur, mais celle-ci est incorrecte.
– Le programme ne termine pas.

En fait, on voudrait éliminer le premier cas, de telle façon que la sémantique soit bien définie pour tous les programmes.

Exercice 16 *Donner deux exemples où l'évaluation est bloquée dans l'évaluation du noyau de ML.*

Pour cela il faut restreindre l'ensemble des programmes à un sous-ensemble des programmes qui s'évaluent correctement.

Cependant cet ensemble n'est pas décidable : on ne peut pas écrire un algorithme qui caractérise les programmes qui s'évalue correctement.

Exercice 17 *Montrer que si on pouvait décider des programmes qui s'exécutent correctement, on pourrait décider de l'arrêt des programmes.*

On cherche donc une approximation par défaut décidable de l'ensemble des programmes corrects.

Évidemment il n'existe pas de meilleure approximation, *i.e.* de plus grand sous-ensemble décidable des programmes corrects. Sinon, cet ensemble contiendrait que et tous les programmes corrects et serait décidable.

Une façon de définir un tel sous-ensemble est d'utiliser le typage.

Typage

Le typage est une appoximation statique (décidable), modulaires des expressions qui permet d'en déduire la forme des valeurs qui peut être le résultat de leur évaluation.

Typiquement, on identifie toutes les valeurs de la même forme en leur donnant le même type. Par exemple,

- Tous les entiers sont du type `int`,
- Toutes les chaînes sont du type `string`,
  Les booléens sont du type `bool`,
- Les fonctions sont d'un type fonctionnel $\tau \to \tau$. Toutefois, il existe plusieurs types fonctionnles, par exemple $\texttt{int} \to \texttt{int}$ (la fonction successeur) $\texttt{string} \to \texttt{string} \to \texttt{string}$ (la fonction concat), $\texttt{int} \to \texttt{string}$ (la conversion d'un entier en une chaîne), etc.

Plus précisément, le type d'une fonction $\tau \to \tau'$ décrit le type τ de son argument et le type τ' du résultat de l'application de la fonction à une valeur du type τ.

La clé et la difficulté réside dans le types des fonctions et de l'application. En effet, c'est là que se passe le calcul, et c'est donc là qu'il faut ne pas perdre d'information dans l'approximation du calcul.

Un type peut aussi être une variable de type, ce qui signifie que le type est quelconque. Mais attention deux occurrences de la même variable représentent le même type.

Par exemple, l'identité $\texttt{fun}\ x \to x$ est une fonction qui retourne son argument. Elle prend une valeur de type τ et retourne une valeur du même type τ. Elle est de type $\tau \to \tau$ pour tout type τ et on peut donc dire qu'elle est de type $\alpha \to \alpha$.

Lorsqu'une fonction est d'un type polymorphe, par exemple $\alpha \to \alpha$, on peut en déduire immédiatement d'autres types, en remplaçant simultanément toutes les occurrences d'une variable par un même type τ.

Aussi l'identité a les types $\texttt{int} \to \texttt{int}$, $\texttt{string} \to \texttt{string}$, $(\beta \to \beta) \to (\beta \to \beta)$, mais elle n'a pas le type $\beta \to \gamma$.

Les types de ML

$\tau ::=$		Valeurs
	c	Constantes de type
	α	Variable de type
	$\tau \to \tau$	Type flèche
$c ::=$	$\texttt{int}, \texttt{string}, \ldots$	

Les constantes de types sont $\texttt{int}, \texttt{string}, \ldots$, mais on se limitera pour simplifié au seul type `int`.

En Ocaml, la solution la plus natuelle est :

```
type tvar = int
type texp =
    Tint
  | Tvar of tvar
  | Tarrow of texp * texp;;
```

(Voir pour une autre solution)

On définit l'ensemble $vl(\tau)$ des variables de types de τ de façon naturelle.

Exercice 18 *Écrire une fonction* libre_texpr : texpr -> tvar list *qui une liste contenant les variables (de type) libres de son argument (éventuellement avec des doublons).*

Exercice 19 *Écrire une fonction* libre : tvar -> texp -> bool *où* libre v t *retourne* true *si la variable* v *apparaît dans le type* t *(*i.e. *si* $\mathtt{v} \in vl(\mathtt{t})$ *et* false *sinon.*

Règles de typage

La relation de typage $A \vdash a : \tau$ exprime que dans l'environnement de typage A le programme a a le type τ.

Un environnement de typage est une fonction partielle des variables de programmes vers les types. Concrètement,

$A ::=$		Envirommenent de typage
	$\emptyset$	Environnement vide
	$x : \tau, A$	Déclaration du type d'une variable

On étend la notion de variables (de type) libres aux contextes.

Exercice 20 *Le type des environnements de typage est*

```
type tenv = (var * texp) list ;;
```

Définir une fonction libre_tenv : tenv -> tvar list *et qui retourne les variables libres d'un environnement de typage.*

La relation de typage est définie par des règles d'inférence.

$$\textsc{Const}\quad A \vdash n : \mathtt{int} \qquad \textsc{Var}\quad \frac{x \in dom\,(A)}{A \vdash x : A(x)} \qquad \textsc{Fun}\quad \frac{A, x : \tau_1 \vdash a : \tau_2}{A \vdash \mathtt{fun}\ x \to a : \tau_1 \to \tau_2}$$

$$\textsc{App}\quad \frac{A \vdash a_1 : \tau_2 \to \tau_1 \qquad A \vdash a_2 : \tau_2}{A \vdash a_1\ a_2 : \tau_1} \qquad \textsc{Let}\quad \frac{A \vdash a_1 : \tau_1 \qquad A, x : \tau_1 \vdash a_2 : \tau_2}{A \vdash \mathtt{let}\ \ x = a_1\ \mathtt{in}\, a_2 : \tau_2}$$

Preuve de typage

Les règles précédentes, permettent de prouver qu'un programme a un type dans un environnement donné. Pour cela, on présente un dérivation de typage. Par exemple,

$$\textsc{Let}\ \dfrac{\textsc{Fun}\ \dfrac{\textsc{Var}\ \dfrac{}{y : \texttt{int} \vdash y : \texttt{int}}}{A \vdash \texttt{fun}\ y \to y : \texttt{int} \to \texttt{int}} \qquad \textsc{App}\ \dfrac{\textsc{Var}\ \dfrac{}{f : \texttt{int} \to \texttt{int} \vdash f : \texttt{int}} \qquad \textsc{Const}\ \dfrac{}{f : \texttt{int} \to \texttt{int} \vdash 1 : \texttt{int}}}{f : \texttt{int} \to \texttt{int} \vdash f\ 1 : \texttt{int}}}{A \vdash \texttt{let}\quad f = \texttt{fun}\ y \to y\ \texttt{in}\ f 1}$$

Mais, il existe plusieurs typage possibles pour un même programme. Par exemple, on peut dérver

$$\dfrac{x : \texttt{int} \vdash x : \texttt{int}}{\emptyset \vdash \texttt{fun}\ x \to x : \texttt{int} \to \texttt{int}} \qquad\qquad \dfrac{x : \alpha \vdash x : \alpha}{\emptyset \vdash \texttt{fun}\ x \to x : \alpha \to \alpha}$$

Substitution

Parmi les deux typages précédents, l'un peut se déduire de l'autre par remplacement de la variable α par le type `int`.

On appelle substitution d'un type α par un type τ_0 une fonction des type dans les types qui remplace dans un type τ toutes les occurences de α dans τ par τ_0.

On généralise à la substitution simultanée d'un ensemble de types.

Exercice 21 *En ocaml, on peut représenter une substitution par une liste de paires :*

```
type subst = (tvar * texp) list ;;
```

Écrire une fonction `subst_texp : subst -> texp -> texp` *qui prend une substitution* `s` *et un type* `t` *et retourne le résultat de l'application de la substitution* `s` *au type* `t`*.*

Exercice 22 *Généraliser la substitution en une fonction sur les environnements de typage :* `subst_tenv : subst -> tenv -> tenv`*.*

Types principaux

Parmi tous les typages possibles d'un programme a, il en existe un dit principal $A \vdash a : \tau$. Cela signifie que tout autre typage $A' \vdash a : \tau'$ du programme a peut se déduire du précédent par instantiation (substitution de ses variables de type par des types).

Inversement, si $A \vdash a : \tau$, alors $\theta(A \vdash a : \tau)$ *i.e.* $\theta(A) \vdash a : \theta(\tau)$).

Cela permet de remplacer la relation de typage par une fonction qui étant donné un environement de typage A et un programme a retourne le type principal τ et une substitution θ tels que $\theta(A) \vdash a : \tau$ soit un typage principal de a.

Ce calcul a besoin de deviner les contraintes minimal à imposé sur les variables pour rendre certains types égaux : il utilise un mécanisme d'unification.

Pour l'instant on suppose que l'on a à notre disposition une fonction

```
unif : texp -> texp -> subst
```

où `unify t1 t2` calcule la plus petite substitution `s` qui égalise `t1` et `t2`.

Unification

On se donne une liste l de paires types et on cherche une substitution s qui rende tous les éléments de u égaux.

Par exemple, si est réduit à la paire ($\alpha \to \texttt{int} =^? \texttt{int} \to \alpha'$) alors la substitution ($\alpha \mapsto \texttt{int}, \alpha' \mapsto \texttt{int}$) est une solution.

S'il existe une solution à un problème d'unification, alors il existe une solution principale.

L'unification utilise la composition de substitutions : $\theta_2 \circ \theta_1$ est la composition au sens mathématique, qui à x associe $\theta_2(\theta_1(x))$.

Elle envoit

1. si $x \in dom\,(\theta_1)$, alors $(\theta_2 \circ \theta_1)(x) = \theta_2(\theta_1(x))$
2. si $x \in dom\,(\theta_2) \setminus dom\,(\theta_1)$, alors $(\theta_2 \circ \theta_1)(x) = \theta_2(x)$
3. sinon, $(\theta_2 \circ \theta_1)(x) = x$

Exercice 23 *Écrire successivement, les fonctions :*

```
domain : subst -> tvar list
```

a) qui retourne le domain d'une substitution.

```
restrict_out : subst -> tvar list -> subst
```

b) où `restrict_out s l` *retourne la restriction de* `s` *aux valeurs en dehors de* `l`*.*

```
compose : subst -> subst -> subst
```

c) qui retourne la composition de deux substitutions.

On peut maintenant implémenter les fonctions

```
unify : texp -> texp -> subst
unify_list : ( texp * texp) list -> subst.
```

récursivement.

```
exception Unify of texp * texp;;
let rec unify t1 t2 =
  match t1, t2 with
  | Tvar x, Tvar y when x = y -> []
  | Tvar x, t ->
      if libre x t then raise (Unify (t1, t2)) else [ x, t ]
  | t, Tvar x ->
      if libre x t then raise (Unify (t1, t2)) else [ x, t ]
  | Tint, Tint -> [ ]
  | Tarrow (t11, t12), Tarrow (t21, t22) ->
      unify_list [ t11, t21; t12, t22 ]
  | t1, t2 -> raise (Unify (t1,t2))

and unify_list l =
  match l with
    [] -> []
  | ( t1,t2):: l ->
      let s1 = unify_list l in
      let s2 = unify (subst_texp s1 t1) ( subst_texp s1 t2) in
      compose s2 s1
;;

unify ( Tarrow (Tint, Tvar 1)) (Tarrow (Tint, Tvar 2));;
# - : (tvar * texp) list = [1, Tvar 2]
```

Algorithme d'inférence

L'algorithme d'inférence de types s'implante en inférant toujours le type le plus général.

La fonction infer : tenv -> texp -> (texp * sustititution) retourne le type principal et la substition minimal à appliquer à l'environnement pour que le programme soit typable.

Il faut alors reporter cette substitution à toutes les autres prémisses de la dérivation.

```
exception Libre of var
let tvar = let last = ref 0 in fun () -> incr last; Tvar !last ;;
let find_type_var x env =
  try List.assoc x env with Not_found -> raise (Libre x)

let rec infer env a =
  match a with
  | Const n -> Tint, [ ]

  | Var x -> find_type_var x env, [ ]

  | App (a1, a2) ->
     let t1, s1 = infer env a1 in
     let t2, s2 = infer (subst_tenv s1 env) a2 in
     let t3 = tvar() in
     let s3 = unif (subst_texp s2 t1) (Tarrow (t2, t3)) in
     subst_texp s3 t3, compose s3 (compose s2 s1)

  | Fonction (x, a) ->
     let t0 = tvar() in
     let t1, s1 = infer ((x, t0):: env) a in
     Tarrow (subst_texp s1 t0, t1), s1

  | Liaison (x, a1, a2) ->
     let t1, s1 = infer env a1 in
     let t2, s2 = infer ((x, t1):: subst_tenv s1 env) a2 in
     t2, compose s2 s1

  | Prim (f, al) -> failwith "Laissé en exercice ..."
;;
let a = Liaison ("f", Fonction ("x", Var "x"), App (Var "f", Var "f"))
in infer [] a;;
# - : texp * subst = Tint, [4, Tint; 3, Tint]
```

Exercice 24 *Implémenter le cas des primitive dans l'algorithme d'inférence. Voir aussi l'algorithme complet.*

Versions impératives

En pratique, on utilise une version impérative de l'algorithme d'unification et d'inférence, ce qui simplifie nettement les algorithmes et les rend plus efficaces.

Généralisation

On étend les types avec des schémas de types de $\forall(\overline{\alpha})\,\tau$ et on redéfinit les environnements de typage pour autoriser des schémas de types.

$A ::=$	Envirommenent de typage
$\quad \emptyset$	Environnement vide
$\quad x : \tau, A$	Liaison de type monomorphe
$\quad x : \sigma, A$	Liaison de type polymorphe

On modifie la règles *Let* et on ajoute une nouvelle forme de la règle *Var* :

$$\textsc{Let}\quad \frac{A \vdash a_1 : \tau_1 \qquad A, x : \forall(vl(\tau_1) \setminus vl(A))\,\tau_1 \vdash a_2 : \tau_2}{A \vdash \texttt{let}\ \ x = a_1\ \texttt{in}\, a_2 : \tau_2}$$

$$\textsc{Var}\quad \frac{x : \forall(\alpha_1, ...\alpha_n)\,\tau}{A \vdash x : \tau[\overline{\alpha}_1 \leftarrow \tau_1, ...\alpha_n \leftarrow \tau_n]}$$