# L'approche qualité

# avec Objective-Caml

Par **Damien Guichard**

Dans ce tutoriel nous allons aborder les concepts du paradigme fonctionnel et les situer dans le cadre plus large d'une démarche méthodologique qui vise à atteindre la qualité totale au plus tôt. Dans son but cette démarche s'oppose à la tendance naturelle qui repousse les difficultés au plus tard, au mieux lors du débogage, au pire lors du test qualité, ou pire encore, lors de l'utilisation. Nous illustrons par de nombreux exemples comment le paradigme fonctionnel et l'approche qualité coopèrent à améliorer la confiance du programmeur.

Pour accéder à la documentation officielle du langage *Objective-Caml* cliquez sur le chameau ci-dessus.
Pour toute interrogation suscitée par *L'approche qualité avec Objective-Caml* veuillez exposer ce qui vous fait obstacle dans le fil de discussion réservé à cet effet.

## Partie I: Présentation

1. L'approche qualité
2. Programmation et composants
3. Programmation et langage
4. Installation d'Objective Caml

## Partie II: La programmation fonctionnelle

5. Expressions immédiates
6. Fonctions et application
7. Déclarations de variables
8. Déclarations et récursion
9. Déclarations de fonctions
10. Récursion et accumulation
11. Déclarations locales
12. Fonctions et assertions
13. Les types énumérés
14. Le filtrage
15. Les listes
16. Les fonctions sur les listes

MCours.com

17. Le polymorphisme paramétrique
18. Les couples
19. Les huit reines
20. Le tri fusion
21. Les modules
22. Le traitement des erreurs
23. Les enregistrements
24. Les performances

## Partie III: Application à la gestion d'inventaires

25. Domaine des inventaires
26. Opérations élémentaires sur les inventaires
27. Opérations avancées sur les inventaires
28. Cas d'usage du module Inventory

## Partie IV: Application à la théorie des jeux

29. Le jeu des tétrominos
30. Le tableau de jeu
31. La génération des coups
32. Le calcul de la valeur d'un tableau

## Partie V: La programmation impérative

33. Les champs mutables
34. Les références
35. Les boucles
36. Les tableaux
37. Les entrées-sorties
38. Les exceptions

## Partie VI: Application aux grands nombres

39. Domaine des entiers naturels
40. Opérations sur les tableaux
41. L'addition
42. La comparaison
43. La soustraction
44. La multiplication longue
45. La multiplication *Knuth-Karatsuba*
46. La division *Burnikel-Ziegler*
47. Calcul de π au goutte-à-goutte

## Partie VII: Application à l'analyse syntaxique

48. Techniques d'analyse syntaxique
49. Les requêtes syntaxiques
50. Un analyseur pour *pascal*
51. Les erreurs de syntaxe
52. Implémentation de *Lex.lex_parser*

## Partie VIII. Les types algébriques

53. Ordre lexicographique
54. Les types produit
55. Les types somme

56. Les types inductifs
57. Les catamorphismes
58. Les paramorphismes

## Remerciements


- à **millie** et à **gorgonite** pour leur soutien
- à **InOCamlWeTrust** pour ses suggestions
- à **Florent Aspic** pour sa promptitude et son attention dans la correction orthographique
- enfin, remerciement spécial à **Nicolas Le Griel** pour sa lecture méticuleuse et ses remarques appréciables


## Partie I: Présentation

1. L'approche qualité
2. Programmation et composants
3. Programmation et langage
4. Installation d'Objective Caml

### 1. L'approche qualité

Le **génie logiciel** (ou *software engineering*) est la discipline qui consiste à dégager un certain nombre de principes de conception et d'implémentation dont l'application systématique vise à réduire les efforts (coût humain), les risques de sûreté (coût en insécurité), les risques de fiabilité (coût des indisponibilités) et les risques économiques (coûts financiers) liés au cycle de vie du produit logiciel, depuis son développement, puis sa validation, puis son déploiement, sa mise en service, sa maintenance, sa mise à jour, jusqu'à sa disparition ou son remplacement.

Le **génie logiciel** est donc par définition essentiellement préventif, il met tout en oeuvre pour minimiser toutes les marges possibles dans les surcoûts liés aux imprévus qui accompagnent les projets logiciels de façon aussi récurrente que pour d'autres réalisations humaines complexes telles que les grands travaux de génie civil. Le génie logiciel ne s'occupe pas des questions humaines et d'organisation sociale, il ne se préoccupe que de la qualité du produit logiciel. Il n'en demeure pas moins que l'application rigoureusement des principes du génie logiciel contribue à atténuer grandement le stress lié à la conception et à la programmation (par l'augmentation de la lisibilité, de la fiabilité et même de l'esthétique du code), augmente la satisfaction et la confiance des utilisateurs (par la robustesse du produit) et plus généralement participe au progrès informatique en rendant possible les logiciels qui doivent exploiter les ressources toujours plus grandes des nouveaux matériels informatiques, sans cesse plus performants, ainsi que satisfaire les nouveaux usages sans cesse plus paramétrables, plus flexibles, plus mobiles et plus communicants.

Enfin, une présentation du génie logiciel ne serait pas complête sans la formulation de sa règle d'or, que le développeur devra toujours garder à l'esprit, et ce d'autant plus si le produit logiciel n'est encore qu'au début de son cycle de développement:

- **les gains en qualité ne peuvent jamais attendre**
- **les gains en fonctionnalités peuvent toujours attendre**

Cette règle de développement est connue sous le nom de *quality first* ou encore *release early, release often*.
Elle dit que la qualité du code doit être maximale dès le début du développement et ne jamais baisser, quitte à différer l'implémentation de fonctionnalités supplémentaires, quelle que soit leur désirabilité.
Elle dit qu'au final la tortue court plus vite que le lièvre.

## 2. Programmation et composants

Au fur et à mesure qu'ils résolvent des problèmes de plus en plus complexes, les programmes informatiques deviennent eux-mêmes de plus en plus complexes. Cette croissance de la complexité doit rester maîtrisée: un programme ne doit pas devenir plus complexe que le problème qu'il résout. Une solution consiste à *diviser pour régner*, la conception d'un logiciel se réduit alors à un problème de découpage en **composants** (*components*) plus élémentaires. Chaque composant offre une fonctionnalité partielle et leur assemblage fournit la totalité de la fonctionnalité recherchée. Il s'ensuit que la première qualité d'un composant est sa composabilité, c'est-à-dire la facilité avec laquelle on peut l'assembler avec ses semblables pour produire un nouveau composant.

Toute la discipline du génie logiciel consiste alors à débattre des mérites comparatifs des différents candidats opérande/opérateur susceptibles de réaliser le couple composant/assembleur.
Dans le cadre de ce cours on a choisi:

- la **fonction** (*function*) comme composant
- la **composition de fonction** comme opérateur de composition

Ce choix est motivé par les facilités suivantes:

- l'application de **fonction** permet le raisonnement équationnel, comme en mathématiques (d'où une bonne lisibilité et une excellente prouvabilité)
- la **composition de fonction** est le moyen de composition qui assure le maximum de découplage entre les composants, c'est-à-dire que les composants, tout en étant assemblables, restent totalement lisibles et compréhensibles indépendamment les uns des autres

es deux qualités seront des atouts précieux pour écrire le plus rapidement possible des programmes qu'on espère de la plus grande qualité possible.

## 3. Programmation et langage

Le langage d'implémentation sera **Objective Caml** (ou en abrégé: **OCaml**).

Le choix de OCaml est motivé par les avantages suivants:

- OCaml est déjà utilisé dans les universités, l'enseignement supérieur et les concours aux grandes écoles, ainsi que dans de nombreux domaines scientifiques et industriels (conception de circuits intégrés, traitement de signal, biologie moléculaire, séquençage ADN, interpréteurs, compilateurs, cryptologie, ERP, XML, OpenGL). Microsoft, Dassault-Aviation, Intel, XenSource sont membres du consortium OCaml.
- OCaml peut être soit un interpréteur (interactif) soit un compilateur (efficace).
- OCaml est portable et disponible sur de nombreux systèmes.
- OCaml pratique l'inférence de type (ou typage automatique).

Ce dernier point est décisif, l'inférence de type est une aide précieuse au développement, elle élimine une grande partie des erreurs de conception au plus tôt, dès la saisie du code. Les erreurs de typage rapportées par OCaml sont moins lisibles qu'avec d'autres langages mais en contrepartie les phases de test et de déboggage sont souvent beaucoup plus restreintes. Il n'est pas rare qu'un code qui compile donne le résultat attendu dès le premier essai. En pratique le gain de productivité est considérable. À chaque nouvelle construction introduite nous introduirons la **règle d'inférence de type** qui l'accompagne.

## 4. Installation d'Objective Caml

**Pour Microsoft Windows 98 et supérieur (versions 32 bits uniquement)**

Installez la dernière version de Objective Caml pour Windows.

Le chemin de l'interpréteur pour l'interface *OCamlWinPlus* est *"C:\Program Files\Objective Caml\bin"*.
Ensuite suivez ce tutoriel pour installer l'éditeur *Crimson* et *MinGW*:

- **http://www.france-ioi.org/cours_caml/win_ide.php**
- **http://www.france-ioi.org/cours_caml/win_ide_avance.php**

L'éditeur *Crimson* offre une édition plus confortable avec:

- la numérotation des lignes
- la coloration syntaxique personnalisable
- la compilation et l'exécution dans l'éditeur, sans ligne de commande
- le décalage gauche/droite des blocs de code
- des macros définissables par l'utilisateur

L'environnement *MinGW* permet la compilation des sources OCaml en code natif sous Windows, ainsi on profite des avantages de l'inférence de types, de l'interpréteur interactif et du ramasse-miettes sans payer le coût en mémoire et en performance qui pénalise les plateformes à compilation juste-à-temps (*Just-In-Time*).

Soyez conscient que l'interface *OCamlWinPlus* est assez capricieuse et que d'une façon générale la plateforme Windows, tout en étant largement utilisable, n'est pas forcément la plus confortable pour le développement OCaml.

Les versions pour *Windows 64bits* sont pour l'instant disponibles sous forme de code source uniquement.

#### Pour Linux x86, Mac OSX (x86 et PPC)

Voyez la page des distributions binaires précompilés officielles.

Ces plateformes sont les seules à bénéficier d'environnements de développement dédiés à OCaml.

#### Pour les autres Unix

Voyez la page de la distribution source officielle.

Il vous faudra lancer le script d'installation qui compilera les sources.

#### Pour BeOS

Il s'agit d'un portage officieux (recompilation à partir du code source).
Un projet d'interfaçage complet avec l'API de BeOS est en cours.
Les binaires de la version 3.08.4 sont à la page http://www.bebits.com/app/3746

#### Pour Syllable OS

Il s'agit d'un autre portage officieux (recompilation du code source).
Les binaires de la version 3.10.2 sont à la page OCaml pour Syllable.

# Partie II: La programmation fonctionnelle


5. Expressions immédiates
6. Fonctions et application
7. Déclarations de variables
8. Déclarations et récursion
9. Déclarations de fonctions
10. Récursion et accumulation
11. Déclarations locales
12. Fonctions et assertions
13. Les types énumérés

14. Le filtrage
15. Les listes
16. Les fonctions sur les listes
17. Le polymorphisme paramétrique
18. Les couples
19. Les huit reines
20. Le tri fusion
21. Les modules
22. Le traitement des erreurs
23. Les enregistrements
24. Les performances

## 5. Expressions immédiates

La programmation en OCaml consiste essentiellement à évaluer des expressions. Lorsqu'une expression valide suivie de ;; est entrée :

1. OCaml synthétise son type.
2. Si la synthèse de type échoue un message d'erreur de type est affiché
3. Sinon OCaml affiche son type et sa valeur

Exemple :

```
# 1 + 3;;
- : int = 4
```

Ce qui signifie: l'expression était entière et valait 4.
Pour tirer le meilleur parti de OCaml il est de la plus grande importance de saisir la stratégie utilisée pour la synthèse de type.
Le type des opérandes doit toujours être conforme au type attendu par les opérateurs.
Ainsi l'expression suivante est mal typée et ne peut pas être évaluée :

```
# 1.0 + 3;;
Characters 0-3:
  1.0 + 3;;
  ^^^
This expression has type float but is here used with type int
```

Parce que + est l'addition entière et attend deux arguments entiers.
Dans ce cas il faut utiliser +. qui est l'addition réelle :

```
# 1. +. 3.;;
- : float = 4.
```

**Remarque:** contrairement à d'autres langages OCaml infère le type des opérandes à l'aide du type des opérateurs.

OCaml distingue les minuscules et les majuscules.
Les commentaires sont ouverts par (* et sont fermés par *) et ils peuvent être imbriqués.

Les types atomiques sont les nombres entiers (**int**), les booléens (**bool**), les chaînes de caractères (**string**), les caractères (**char**) et les nombres réels à virgule flottante (**float**). Voici la liste de ces types accompagnés de leurs opérateurs les plus courants. Ce tableau est suivi de quelques exemples d'utilisation :

| Type: | **int** |
|---|---|
| Instances: | 1 4 -3 0 |

| | |
|---|---|
| Préfixés: | min max abs - lnot float_of_int char_of_int string_of_int |
| Infixés: | + - * / mod land lor lsl lsr = <> < <= >= > |

| | |
|---|---|
| Type: | **bool** |
| Instances: | true false |
| Préfixés: | not |
| Infixés: | && || or = <> |

| | |
|---|---|
| Type: | **string** |
| Instances: | "Hello World!" |
| Préfixés: | String.length print_string int_of_string |
| Infixés: | ^ .[i] = <> < <= >= > |

| | |
|---|---|
| Type: | **char** |
| Instances: | 'a' |
| Préfixés: | int_of_char |
| Infixés: | = <> < <= >= > |

| | |
|---|---|
| Type: | **float** |
| Instances: | 3.14 .5 0. |
| Préfixés: | min max sqrt sin cos tan log exp asin acos atan int_of_float |
| Infixés: | +. -. *. /. ** = <> < <= >= > |

| Exemples: | Résultat: |
|---|---|
| # 1+1;; | - : int = 2 |
| # 2*3 = 13 mod 7;; | - : bool = true |
| # abs (-3);; | - : int = 3 |
| # min 34 (-67);; | - : int = -67 |
| # not (1<2);; | - : bool = false |
| # not (true && false);; | - : bool = true |
| # "good" ^ "bye!";; | - : string = "goodbye!" |
| # "beef".[3];; | - : char = 'f' |
| # 2.0**3.0;; | - : float = 8. |
| # sqrt 4.;; | - : float = 2. |
| # exp 1.;; | - : float = 2.7182818284590451 |
| # cos 0.;; | - : float = 1. |

```
(* La priorité des opérateurs est respectée *)
# 2+3*4;;
- : int = 14
```

```
(* Les parenthèses permettent d'utiliser un opérateur infixe en position préfixe *)
# (+) 2 2;;
- : int = 4
```

```
(* L'opérateur * en position préfixe doit être entouré d'espaces *)
(* sinon il est interprété comme l'ouverture d'un commentaire *)
# ( * ) 2 2;;
- : int = 4
```

Le typage remonte l'arbre d'expression, il s'applique d'abord aux constantes, puis à l'échelle des expressions élémentaires et enfin il se propage à travers toutes les expressions englobantes.

**Règles de typage (expressions simples)**

La règle de typage d'une expression *abs n*, *-n*, *lnot n*, *float_of_int n*, *char_of_int n*, ou *string_of_int n* distingue deux cas possibles :

- ou bien le type de *n* est **int** alors l'expression est de type (respectivement) **int**, **int**, **int**, **float**, **char**, **string**
- ou bien l'expression est mal typée

La règle de typage d'une expression *a + b*, *a - b*, *a * b*, *a* **mod** *b*, *a* **land** *b*, *a* **lor** *b*, *a* **lsl** *b*, *a* **lsr** *b*, distingue deux cas possibles :

- ou bien le type de *a* est **int** et le type de *b* est **int** alors l'expression est de type **int**
- ou bien l'expression est mal typée

La règle de typage d'une expression *a = b*, *a <> b*, *a < b* et *a <= b*, *a >= b*, ou *a > b* distingue deux cas possibles :

- ou bien *a* et *b* ont un même type *E* alors l'expression est de type **bool**
- ou bien l'expression est mal typée

La règle de typage d'une expression *min a b* ou *max a b* distingue deux cas possibles :

- ou bien *a* et *b* ont un même type *E* alors l'expression est de type *E*
- ou bien l'expression est mal typée

La règle de typage d'une expression *a && b*, *a || b* ou *a or b* distingue deux cas possibles :

- ou bien le type de *a* est **bool** et le type de *b* est **bool** alors l'expression est de type **bool**
- ou bien l'expression est mal typée

La règle de typage d'une expression *sqrt x*, *sin x*, *cos x*, *tan x*, *log x*, *exp x*, *asin x*, *acos x* ou *atan x* distingue deux cas possibles :

- ou bien le type de *x* est **float** alors l'expression est de type **float**
- ou bien l'expression est mal typée

## 6. Fonctions et application

En OCaml les notions d'expression et de valeur ont un sens assez large puisqu'elles englobent aussi les fonctions qui sont des valeurs à part entière.
Une fonction d'une seule variable se déclare à l'aide du mot-clé **fun** et d'une flèche, par exemple la fonction de n qui renvoie n + 1 se déclare comme ceci :

```
# fun n -> n + 1;;
- : int -> int = <fun>
```

L'interpréteur a bien évalué cette expression comme une valeur et lui a attribué le type *int* → *int*, c'est-à-dire une fonction d'un entier qui renvoie un entier.
Cette attribution de type se fait grâce au soutien des règles de typage, au niveau le plus élevé, celui de la fonction, le système de typage a réuni suffisamment d'informations de typage pour en déduire le type complet d'une fonction.
La flèche est un constructeur infixe pour les types fonctionnels, elle prend à gauche le type d'un argument et à droite le type d'une application à un argument. Un type fonctionnel est le correspondant OCaml d'une *signature* dans d'autres langages. Dans ces autres langages une signature n'est pas un type parce que les fonctions ne sont pas des valeurs à part entière.

L'application de fonction est une opération tellement élémentaire qu'en OCaml on l'écrit par simple juxtaposition, par exemple le successeur de l'entier 2 est l'entier 3 :

```
# (fun n -> n + 1) 2;;
- : int = 3
```

Une fonction à plusieurs argument se construit en imbriquant plusieurs fonctions à un seul argument, par exemple cette fonction réalise l'addition de deux entiers *a* et *b* :

```
fun a -> (fun b -> a + b);;
```

Son type est *int* → *(int* → *int*), car la flèche est associative à droite, si on lui passe en entier comme argument alors le résultat est de type *int* → *int*. Comme la flèche est associative à droite et que l'application de fonction est associative à gauche, pour additionner 1 et 2 on pourra écrire :

```
(fun a -> fun b -> a + b) 1 2;;
```

La valeur 1 est alors associée à la variable *a* puis la valeur 2 est associée à la variable *b* dans l'expression 1 + *b* ce qui donne 1 + 2 = 3.
On peut écrire en plus condensé :

```
(fun a b -> a + b) 1 2;;
```

Cette abréviation n'altère en rien le mécanisme d'évaluation qui reste exactement le même, en particulier, comme le suggérait le type *int* → *(int* → *int*), on peut construire une fonction d'incrémentation en appliquant 1 à cette fonction :

```
(fun a b -> a + b) 1;;
```

**Règle de typage (construction d'une fonction)**
La règle de typage d'une expression **fun** *v* → *expr* distingue trois cas possibles :

- ou bien toutes les occurrences de *v* dans *expr* ont un même type *T* alors l'expression est de type *T* → *E* où *E* est le type de *expr*
- ou bien il n'y a aucune occurrence de *v* dans *expr* alors l'expression est de type *'a* → *E* où *E* est le type de *expr*, ce cas ne nous intéresse pas pour l'instant, nous définirons le type *'a* au moment opportun
- ou bien l'expression est mal typée

**Règle de typage (application d'une fonction)**
La règle de typage d'une application *f x* distingue deux cas possibles :

- ou bien l'expression *f* est du type $X \rightarrow Y$ et l'expression *x* est du type *X* alors l'application *f x* est de type *Y*
- ou bien l'application *f x* est mal typée

## 7. Déclarations de variables

Les déclarations se font avec le mot-clé **let** qui associe une valeur à une variable, la constante *pi* pourra être définie ainsi :

```
let pi = 3.1415926535898;;
```

L'addition entière pourra être définie ainsi :

```
let add = fun a b -> a + b;;
```

**Attention:** les variables ainsi définies sont liées, elles ne sont pas modifiables, leur valeur est fixée, elles ne peuvent pas en changer en utilisant une assignation.

Les mots-clés **if** ... **then** ... **else** ... permettent un choix parmi deux expressions possibles selon la valeur d'une condition :

```
let abs = fun z -> if z >= 0 then z else -z;;
let minimum = fun a b -> if a < b then a else b;;
let maximum = fun a b -> if a > b then a else b;;
```

Le **if** ... **then** ... **else** ... n'est pas une instruction mais une expression qui peut prendre deux valeurs possibles.

**Règle de typage (expression conditionnelle)**
La règle de typage d'une expression **if** *cond* **then** *expr1* **else** *expr2* distingue deux cas possibles :

- ou bien le type de *cond* est **bool** et les expressions *expr1* et *expr2* ont un même type *E* alors l'expression est de type *E*
- ou bien l'expression est mal typée

## 8. Déclarations et récursion

Le mot-clé **rec** permet la récursion c'est-à-dire l'utilisation d'une variable pendant sa définition avec **let** :

```
let rec factorial =
  fun n -> if n=0 then 1 else n * factorial (n - 1);;

let rec power =
  fun x n -> if n=0 then 1. else x *. power x (n-1);;

let rec binomial =
  fun n p -> if (p=0) or (p=n) then 1 else binomial (n-1) p + binomial (n-1) (p-1);;

let rec fibonacci =
  fun n -> if n < 2 then 1 else fibonacci (n - 1) + fibonacci (n - 2);;
```

**Remarque:** pour donner une idée intuitive de cette dernière fonction *fibonacci* disons que si une grenouille pouvait sauter au maximum deux marches d'escalier alors *fibonacci n* serait le nombre de façons différentes pour cette grenouille de monter *n* marches à l'aide d'une combinaison de sauts d'une ou deux marches.

## 9. Déclarations de fonctions

Les déclarations suivantes sont, par abus de langage, appelées déclarations de fonctions :

```
let rec factorial n = if n=0 then 1 else n * factorial (n - 1);;

let rec power x n = if n=0 then 1. else x *. power x (n-1);;

let rec binomial n p = if (p=0) or (p=n)  then 1 else binomial (n-1) p + binomial (n-1) (p-1);;

let rec fibonacci n = if n < 2 then 1 else fibonacci (n - 1) + fibonacci (n - 2);;
```

Leur sémantique est exactement la même que leurs précédentes versions respectives. *factorial*, *power*, *binomial* et *fibonacci* restent des *variables* liées à des *valeurs fonctionnelles*.

## 10. Récursion et accumulation

Il existe une façon alternative de définir des fonctions récursives: au lieu d'effectuer le calcul au plus tard on l'effectue au plus tôt, pour cela on rajoute un paramètre dit *accumulateur* qui enregistre le résultat partiel du calcul en cours :

```
let rec factorial acc n =
  if n=0 then acc else factorial (acc*n) (n-1);;
```

L'expression *factorial 1* correspond alors à la fonction factorielle recherchée.

**Remarque:** on a bien renversé le sens du calcul, cette version itérative calcule n×(n-1)×(n-2)×...×2 alors que la version sans accumulation calculait 2×...×(n-2)×(n-1)×n, bien sûr cela n'a pas d'intérêt ici car la multiplication est commutative mais cet effet peut se révéler très utile lors d'applications plus concrètes comme nous aurons l'occasion d'en donner des exemples.

Nombre de fonctions récursives (mais pas toutes) peuvent être réécrites à l'aide d'un ou plusieurs accumulateurs :

```
let rec power acc x n =
  if n=0 then acc else power (acc *. x) x (n-1);;
```

L'expression *power 1.0* correspond à la fonction puissance ordinaire.

Pour la fonction *fibonacci* on aura besoin de deux accumulateurs :

```
let rec fibonacci acc0 acc1 n =
  if n < 2 then acc1 else fibonacci acc1 (acc0 + acc1) (n - 1);;
```

L'expression *fibonacci 1 1* correspond à la fonction *fibonacci* ordinaire.

Cette définition de *fibonacci* est beaucoup plus efficace que la version purement récursive. Pour faire une comparaison imaginons qu'avec la version à accumulateurs la grenouille se contente de monter *n* marches, alors avec la version récursive la grenouille expérimenterait toutes les façons possibles de monter ces mêmes *n* marches.

Dans toutes ces implémentations avec accumulateurs la branche **then** renvoie le résultat final tandis que la branche **else** n'est qu'une ré-application de la fonction avec de nouveaux arguments. Comme nous le verrons plus loin cette propriété dite de *récursion terminale* a aussi son importance dans les performances des calculs intensifs.

## 11. Déclarations locales

Avouons-le, les versions à accumulateur(s) sont moins confortables à l'usage, à cause de leur(s) paramètre(s) supplémentaire(s). Il serait plus commode de camoufler les paramètres-accumulateurs pour n'utiliser que les paramètres significatifs. Pour ce faire nous allons utiliser les mots-clés **let** *v* = *expr1* **in** *expr2*. Cette construction autorise une définition locale, elle agit comme un **let** ordinaire sauf que la variable *v* n'est définie que dans *expr2* où elle vaudra *expr1*. La variable *v* n'est pas définie dans le reste du programme. La construction **let rec** *v* = *expr1* **in** *expr2* autorise la récursion sur une définition locale de la variable *v*, c'est-à-dire que *v* pourra être présente dans *expr1*.

Nous avons besoin d'un **let** pour créer le raccourci ainsi que d'un **let rec** qui définira la fonction récursive locale à accumulateur :

```
let factorial n =
  let rec loop acc n =
    if n=0 then acc else loop (acc*n) (n-1)
  in loop 1 n;;

let power x n =
  let rec loop acc n =
    if n=0 then acc else loop (acc *. x) (n-1)
  in loop 1. n;;

let fibonacci n =
  let rec loop acc0 acc1 n =
    if n < 2 then acc1 else loop acc1 (acc0 + acc1) (n - 1)
  in loop 1 1 n;;
```

Il existe une autre raison, plus profonde que celle de la commodité, pour laquelle un changement dans l'expression d'une fonction ne doit pas modifier la façon d'appliquer cette fonction. Cette raison tient à l'approche de *componentisation* systématique sur laquelle nous nous sommes engagés dès notre présentation de la notion de *complexité* dans les programmes. Les fonctions sont un de nos meilleurs atouts, si nous voulons préserver leur *composabilité* il nous faut préserver leur type car il est le connecteur qui permet de les emboîter les unes avec les autres.

**Règle de typage (déclaration locale)**
La règle de typage d'une expression **let** *v* = *expr1* **in** *expr2* distingue trois cas possibles :

- ou bien toutes les occurrences de *v* dans *expr2* ont le même type que *expr1* alors l'expression est de type *E* où *E* est le type de *expr2*
- ou bien il n'y a aucune occurence de *v* dans *expr2* alors l'expression est de type *E* où *E* est le type de *expr2*
- ou bien l'expression est mal typée

## 12. Fonctions et assertions

Parfois le domaine de définition d'une fonction est strictement inclus dans son type, on utilisera alors le mot-clé **assert** pour garantir qu'une fonction n'est toujours appliquée que sur son domaine de définition :

```
let factorial n =
  assert(n >= 0);
  let rec loop acc n =
    if n=0 then acc else loop (acc*n) (n-1)
  in loop 1 n;;

let power x n =
  assert(n >= 0);
  let rec loop acc n =
    if n=0 then acc else loop (acc *. x) (n-1)
  in loop 1. n;;

let fibonacci n =
  assert(n >= 0);
  let rec loop acc0 acc1 n =
```

```
      if n < 2 then acc1 else loop acc1 (acc0 + acc1) (n - 1)
    in loop 1 1 n;;

  let rec binomial n p =
    assert(0 <= p && p <= n);
    if (p=0) or (p=n)  then 1 else binomial (n-1) p + binomial (n-1) (p-1);;
```

Lorsqu'une fonction sera appliquée hors de son domaine de définition OCaml terminera l'évaluation par une erreur *Assert_failure* telle que celle-ci :

```
# factorial (-1);;
Exception: Assert_failure ("", 5, 1).
```

Une assertion ne fait que rendre explicite une indication d'usage que vous auriez autrement mis en commentaire. Expliciter ce qui autrefois était implicite c'est un thème récurrent du principe *quality first*.

Le respect du principe *quality first* prescrit d'écrire systématiquement les assertions nécessaires, avant même d'écrire le corps de la fonction. Vous vous remercierez de l'avoir fait pendant le déboguage. Anticiper les phases suivantes du développement c'est un autre thème récurrent du principe *quality first*. N'effacez jamais une assertion, au pire mettez-là en commentaire.

Les assertions font également parti de notre panoplie d'outils pour préserver la composabilité, elle doivent être considérées comme des types, la contrainte qu'elles ajoutent a du sens, elle capture une restriction de domaine que notre système de type n'est pas assez expressif pour imposer à la compilation.

**Règle de typage (assertion)**
La règle de typage d'une expression **assert** (*cond*) distingue deux cas possibles :

- ou bien le type de *cond* est **bool** alors l'expression est de type **unit** (nous reparlerons de ce type **unit** au moment opportun)
- ou bien l'expression est mal typée

## 13. Les types énumérés

Un *type énuméré* est un ensemble fini qui correspond au domaine de définition d'une fonction. De cette façon on n'aura pas à ajouter d'*assertion*, car le type de la fonction correspondra à son domaine de définition.

```
type day =
  | Monday
  | Tuesday
  | Wednesday
  | Thursday
  | Friday
  | Saturday
  | Sunday
  ;;
```

Chaque jour ainsi énuméré devient un constructeur pour le type *day*.

**Remarque:** le nom de chacun des membres (on dit *variants*) d'un type énuméré doit commencer par une majuscule, c'est une obligation, au contraire les autres identificateurs habituels (les types, fonctions, variables et étiquettes) doivent toujours commencer par une lettre minuscule.

**Règle de typage (constructeur)**
La règle de typage d'une expression *Constructeur* ne contient qu'un seul cas possible :

- le type d'un *Constructeur* est le *type* dans lequel il a été énuméré

## 14. Le filtrage

Le filtrage (*pattern matching*) permet une analyse de cas en distinguant chacun des motifs (*patterns*) possibles. Une variable dans un motif prend la valeur à laquelle elle est associée dans ce motif. Les motifs sont ordonnés, ils sont confrontés tour-à-tour à la valeur filtrée, le premier motif filtrant cette valeur sélectionne la valeur renvoyée par l'expression filtrante. Les variables dans les motifs filtrent toutes les valeurs.

```
let rec fibonacci n =
  assert(n >= 0);
  match n with
  | 0 -> 1
  | 1 -> 1
  | m -> fibonacci(m-1) + fibonacci(m-2) ;;
```

La sémantique de l'expression filtrante **match** dans cette fonction est la suivante :

- si *n* = 0 alors l'expression vaut 1
- si *n* = 1 alors l'expression vaut 1
- sinon l'expression vaut **let** *m* = *n* **in** *fibonacci(m-1) + fibonacci(m-2)*

Comme les motifs sont ordonnés un motif plus général pourra cacher un motif plus spécifique, dans ce cas OCaml délivrera un avertissement à prendre au sérieux comme dans l'exemple suivant où les motifs ont été clairement mal ordonnés :

```
# let rec factorial n =
    match n with
    | m -> m * factorial (m - 1)
    | 0 -> 1;;
Characters 80-81:
Warning U: this match case is unused.
    | 0 -> 1;;
      ^
val factorial : int -> int = <fun>
```

De même qu'il existe des fonctions anonymes (introduites par le mot-clé **fun**) il existe également des variables anonymes.
La variable anonyme est notée "_", la fonction *fibonacci* peut donc être réécrite :

```
let rec fibonacci n =
  assert(n >= 0);
  match n with
  | 0 -> 1
  | 1 -> 1
  | _ -> fibonacci(n-1) + fibonacci(n-2) ;;
```

Le filtrage multiple, c'est-à-dire sur plusieurs valeurs, est possible à l'aide de la virgule :

```
let rec binomial n p =
  assert(0 <= p && p <= n);
  match n,p with
  | _,0 -> 1
  | _,_ -> if p = n then 1
           else binomial (n-1) p + binomial (n-1) (p-1) ;;
```

Les schémas autorisés dans un motif sont :

- une nouvelle variable
- un constructeur
- une constante immédiate
- une nouvelle variable anonyme

Les expressions et les applications de fonction ne sont pas autorisées dans les motifs car un motif n'est pas une valeur mais un schéma de valeurs.

OCaml vérifie l'exhaustivité du filtrage, si un cas n'est pas filtré, un message d'alerte vous invite à intégrer ce cas ou à reconsidérer votre fonction :

```
# let string_of_day d =
    match d with
    | 1 -> "Monday"
    | 2 -> "Tuesday"
    | 3 -> "Wednesday"
    | 4 -> "Thursday"
    | 5 -> "Friday"
    | 6 -> "Saturday"
    | 7 -> "Sunday";;
Characters 23-97:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
0
```

Vous devez prendre cet avertissement au sérieux, dans le cas ci-dessus le filtrage a peu de sens car il y a une infinité d'entiers pour seulement 7 jours de la semaine. L'avertissement concernant la fonction *string_of_day* est le symptôme d'une erreur sur le domaine de définition, dans notre cas un type énuméré ferait davantage l'affaire :

```
let string_of_day d =
  match d with
  | Monday    -> "Monday"
  | Tuesday   -> "Tuesday"
  | Wednesday -> "Wednesday"
  | Thursday  -> "Thursday"
  | Friday    -> "Friday"
  | Saturday  -> "Saturday"
  | Sunday    -> "Sunday";;

let tomorrow d =
  match d with
  | Monday    -> Tuesday
  | Tuesday   -> Wednesday
  | Wednesday -> Thursday
  | Thursday  -> Friday
  | Friday    -> Saturday
  | Saturday  -> Sunday
  | Sunday    -> Monday;;
```

**Remarque:** OCaml rejette la définition multiple d'une même variable dans un même motif.

```
# let equal a b =
    match a,b with
    | x,x -> true
    | _,_ -> false;;
Characters 43-44:
      | x,x -> true
          ^
This variable is bound several times in this matching
```

Ici, ce qui semble une contrainte d'égalité est en fait une tentative de définir la même variable avec deux valeurs différentes.

**Règle de typage (filtrage)**
La règle de typage d'une expression **match** *expr* **with** $pat_1 \rightarrow expr_1$ | ... | $pat_n \rightarrow expr_n$ comportant *n* clauses distingue deux cas possibles :

- ou bien l'expression *expr* et les motifs $pat_1$ à $pat_n$ ont un même type *T* et les expressions $expr_1$ à $expr_n$ ont un même type *E* alors l'expression est de type *E*
- ou bien l'expression est mal typée

## 15. Les listes

La liste chaînée est le type *conteneur*-extensible le plus simple, tellement simple qu'en OCaml ce type est prédéfini.
Il existe deux façons distinctes d'exprimer une liste chaînée :

- ou bien par énumération, en encadrant les éléments par des crochets, séparés par des points-virgules

```
# [2;3;4];;
- : int list = [2; 3; 4]
```

La liste vide est l'énumération vide *[]*.

- ou bien par construction à l'aide du *constructeur* infixe ::

```
# 2::3::4::[];;
- : int list = [2; 3; 4]
```

Dans le cas du constructeur :: l'argument à gauche est appelé la tête (*head*) et celui à droite la queue (*tail*) de la liste.

Les fonctions qui manipulent des listes utilisent abondamment le filtrage :

- une liste énumérée pour le (ou les) cas de base
- une liste construite pour les cas composites

Par exemple, la fonction *sum_list* renvoie la somme des entiers d'une liste, la somme des éléments d'une liste vide étant nulle, et la somme des éléments d'une liste non vide étant égale à sa tête *n* plus la somme des éléments de sa queue *t* :

```
let rec sum_list l =
  match l with
  | []   -> 0
  | n::t -> n + sum_list t;;
```

Il s'agit là d'un raisonnement par cas qui sera souvent utilisé sur les listes :

- ou bien on est dans le cas trivial de la liste vide
- ou bien on déconstruit la liste et on applique le traitement adéquat sur sa tête et sa queue, souvent on aura une récursion sur la queue

Plus rarement il faudra traiter le cas de la liste à un seul élément ou alors il faudra considérer les deux premiers éléments en tête de la liste. Nous aurons bien assez tôt l'occasion de voir maints exemples de fonctions sur les listes.

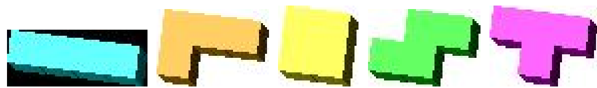
**Règle de typage (liste)**
La règle de typage d'une expression *a::l* ne contient qu'un seul cas possible :

- le type de l'expression *a* est *T*
- le type de l'expression *l* est *T list*
- le type de l'expression *a::l* est *T list*

## 16. Les fonctions sur les listes

On se propose de donner deux exemples plus avancés avec les listes.
En premier on déterminera toutes les vues sur un **tétromino**.
En second on donnera la liste des coordonnées des sommets du cube unité en dimensions quelconques.

Un *polyomino* est une congruence orthogonale de *n* carrés. Le jeu *Tetris* tire son nom du fait qu'il se joue avec les congruences orthogonales de 4 carrés, ce sont les 5 **tétrominos** surnommées I.L.O.S.T. :



On représentera un polyomino par une *edge list*, une liste des arêtes (*edge*) orientées dans le sens des aiguilles d'une montre où le type *edge* est défini par :

```
type edge =
  | Up
  | Down
  | Left
  | Right
  ;;
```

Voici les 5 tétrominos I.L.O.S.T. :

```
let tetro_I = [Up;Up;Up;Up;Right;Down;Down;Down;Down;Left];;
let tetro_L = [Up;Up;Up;Right;Down;Down;Right;Down;Left;Left];;
let tetro_O = [Up;Up;Right;Right;Down;Down;Left;Left];;
let tetro_S = [Up;Right;Up;Right;Right;Down;Left;Down;Left;Left];;
let tetro_T = [Up;Left;Up;Right;Right;Right;Down;Left;Down;Left];;
```

On voudra pouvoir les tourner (d'un quart de tour dans le sens horaire) :

```
let rec rotate p =
  match p with
  | [] -> []
  | Up::t    -> Right::rotate t
  | Down::t  -> Left::rotate t
  | Left::t  -> Up::rotate t
  | Right::t -> Down::rotate t
  ;;
```

On voudra également pouvoir les miroiter (échange gauche-droite) :

```
let mirror p =
  let rec loop l a =
    match l with
    | [] -> a
    | Up::t    -> loop t (Down::a)
    | Down::t  -> loop t (Up::a)
    | Left::t  -> loop t (Left::a)
    | Right::t -> loop t (Right::a)
  in loop p [];;
```

**Remarque:** pour pouvoir retourner la liste on a utilisé une fonction avec un accumulateur.

Nous allons maintenant représenter le vecteur unité par la liste [[0];[1]] :

```
# [[0];[1]];;
- : int list list = [[0]; [1]]
```

Et le carré unité par la liste [[0;0]; [0;1]; [1;0]; [1;1]]. Pour généraliser en dimension *n* on s'aidera de la fonction *mult_unit* qui multiplie un cube *c* par l'intervalle unité lui ajoutant ainsi une dimension. Il suffit alors d'utiliser cette multiplication pour calculer la puissance *n*-ième de l'intervalle unité, c'est ce que fait la fonction *power_cube* :

```
let rec power_cube n =
  assert(n >= 1);
  let rec mult_unit c =
    match c with
    | []   -> []
    | h::t -> (0::h)::(1::h)::mult_unit t
  in
    if n=1 then [[0];[1]]
    else mult_unit (power_cube (n-1));;
```

## 17. Le polymorphisme paramétrique

Nous avons déjà vu qu'une fonction n'a qu'un seul paramètre (sur son *domaine*) et ne renvoie qu'un résultat (sur son *co-domaine*).

Une conséquence logique c'est qu'une fonction n'a que deux schémas de type possibles :

- ou bien le co-domaine est identique au domaine, alors le type de la fonction est de la forme $\alpha \rightarrow \alpha$
- ou bien le co-domaine est différent du domaine, alors le type de la fonction est de la forme $\alpha \rightarrow \beta$

Ou encore, dit autrement, l'opérateur infixe $\rightarrow$ est le constructeur du *type fonctionnel* $\alpha \rightarrow \beta$.

En OCaml les schémas de type $\alpha$ et $\beta$ sont notés respectivement 'a et 'b :

```
# let f x = x in f;;
- : 'a -> 'a = <fun>

# let rec f x = f x in f;;
- : 'a -> 'b = <fun>
```

Un type paramétré par un schéma de type est dit *polymorphe*.
Un tel type n'est pas une curiosité, nous en avons déjà rencontré un :

```
# [];;
- : 'a list = []
```

Le type *list* est un type paramétré, une liste est homogène, tous ses éléments sont forcément du même type *'a*. Ce paramètre de type 'a peut être fixé librement d'où le nom de *polymorphisme paramétrique* donné à cette capacité.
Ce polymorphisme paramétrique s'applique aussi bien aux types qu'aux fonctions :

- un type polymorphe permet de séparer le type conteneur du type contenu
- une fonction polymorphe permet de séparer le type de la fonction du type de ses paramètres

Les deux vont de paire, la définition d'un type polymorphe s'accompagnant des fonction polymorphes qui agissent sur lui.
Par exemple des fonctions sur une liste qui ne présagent pas du contenu de cette liste si bien qu'elles sont utiles sur tous les types de liste possibles.
C'est en premier lieu le cas du constructeur de liste, qui n'est pas une fonction puisqu'on ne peut pas l'appliquer, mais dont on va faire apparaître le type de la façon suivante :

```
# fun h t -> h::t;;
- : 'a -> 'a list -> 'a list = <fun>
```

La distinction entre fonctions et constructeurs se justifie par le filtrage, une fonction est faite pour être appliquée, un constructeur est fait pour être instancié et filtré.

L'opérateur *infixe* @ de concaténation de liste est également polymorphe :

```
# (@);;
- : 'a list -> 'a list -> 'a list = <fun>
```

Il est temps maintenant de profiter de la *pleine fonctionnalité*. Nous allons encore écrire des fonctions jouets, mais dorénavant elles seront plus amusantes que jamais. Alors écrivons 7 fonctions *polymorphes* sur les listes.

La première s'appelle *filter*, elle est du type :

```
('a -> bool) -> 'a list -> 'a list
```

Son 1ier paramètre est donc un *prédicat* (une fonction qui teste un élément), son 2nd paramètre est une liste et la fonction renvoie également une liste. Et le type nous dit encore plus que ceci. En effet regardons les paramètres de type, ils nous disent que le prédicat peut tester tous les élements de la liste donnée en paramètre et que si on appliquait un prédicat à *filter*, c'est-à-dire si on ne lui fournissait que le premier de ses deux arguments (on appelle ceci une *application partielle*), alors *filter* serait un *endomorphisme* (une fonction de type $\alpha \rightarrow \alpha$), elle aurait donc le même type que la fonction identité.

À partir de là, sur la seule foi de son type, il est possible d'imaginer ce que fait la fonction *filter*: elle renvoie la liste donnée en paramètre à laquelle ont été ôtés tous les éléments qui ne satisfont pas le prédicat *('a -> bool)*. Nous allons maintenant l'implémenter, voilà comment nous procédons, en raisonnant par cas:

- ou bien la liste est vide alors la liste filtrée est vide
- ou bien la liste a une tête et une queue alors on filtre la queue et on lui ajoute la tête (si elle satisfait le prédicat) ou pas (si elle ne satisfait pas le prédicat)

Ce qui nous donne :

```
let rec filter p l =
  match l with
  | [] -> []
  | h::t -> if p h then h::filter p t
            else filter p t;;
```

Passons à notre deuxième fonction polymorphe, elle s'appelle *exists* et elle est du type :

```
('a -> bool) -> 'a list -> bool
```

Raisonnons sur son type, c'est encore une fonction qui attend un prédicat et une liste, encore une fois le prédicat teste les éléments de cette liste (puisque lui et la liste ont le même paramètre de type *'a*), mais cette fois-ci la fonction *exists* est elle-même un prédicat (sur la liste donnée en argument) car si on lui applique un prédicat (par application partielle) alors son type devient $\alpha \rightarrow$ *bool*. La fonction *exists* fait l'union logique de l'application du prédicat sur tous les éléments de la liste. Avec ce type elle pourrait faire la conjonction au lieu de l'union (c'est d'ailleurs le rôle de la fonction *for_all* que nous verrons plus tard). Le type d'une fonction ne la caractérise donc pas totalement mais il en dit déjà assez long et il serait désavantageux d'ignorer cette information.

Raisonnons par cas :

- ou bien la liste est vide, elle ne contient alors aucun élément qui puisse satisfaire le prédicat, nous renvoyons **false**
- ou bien la liste a une tête et une queue alors nous renvoyons la conjonction logique du prédicat sur la tête et de la récursion sur la queue

Ce qui nous donne :

```
let rec exists p l =
  match l with
```

```
      | [] -> false
      | h::t -> p h or exists p t;;
```

Passons à notre troisième fonction polymorphe, elle s'appelle *mapi* et elle est du type :

```
(int -> 'a -> 'b) -> int -> 'a list -> 'b list
```

Imaginons que les **int** soient absents de ce type alors le type serait celui-ci :

```
('a -> 'b) -> 'a list -> 'b list
```

Et c'est tout de suite plus clair: nous comprenons que la fonction *mapi* applique la fonction *('a → 'b)* sur tous les éléments de la liste *('a list)* et renvoie une liste *('b list)*. Alors que viennent faire les **int** ? C'est simple, il s'agit d'un argument supplémentaire qui croit avec la position de l'élément *'a* dans la liste *('a list)*.

Raisonnons à nouveau par cas :

- ou bien la liste est vide alors la fonction *f* ne s'applique jamais et la liste renvoyée est vide
- ou bien la liste a une tête et une queue alors nous appliquons *(f n h)* et nous ajoutons le résultat à la transformation de la queue

Ce qui nous donne :

```
let rec mapi f n l =
  match l with
    | [] -> []
    | h::t -> f n h::mapi f (n+1) t;;
```

Notre quatrième fonction polymorphe utilisera notre fonction *exists*, elle s'appelle *exists_commutative* et elle est du type :

```
('a -> 'a -> bool) -> 'a list -> bool
```

Ce type ressemble beaucoup à celui de *exists* sauf que cette fois-ci le prédicat teste deux éléments de la liste donnée en paramètre. Cette fonction nous dira donc s'il existe deux élements dans la liste qui vérifient le prédicat donné en paramètre.

Raisonnons par cas :

- ou bien la liste est vide alors elle ne contient pas deux éléments, nous renvoyons **false**
- ou bien la liste a une tête et une queue alors nous renvoyons la conjonction logique du prédicat (duquel on a fixé le premier argument avec la tête) appliqué à toute la queue et de la récursion sur la queue

```
let rec exists_commutative p l =
  match l with
    | []   -> false
    | h::t -> (exists (p h) t) or exists_commutative p t;;
```

Notez que le prédicat doit être commutatif car pour chaque couple d'élément il n'est appliqué qu'une seule fois, si le prédicat était non commutatif il faudrait également l'appliquer en échangeant l'ordre des deux éléments.

Notre cinquième fonction polymorphe est la fonction *flatten* qui renvoie la concaténation de plusieurs listes :

MCours.com

```
let rec flatten l =
  match l with
   | [] -> []
   | h::t -> h @ flatten t;;
```

Notre sixième fonction polymorphe est *distribute* qui utilise *mapi* pour renvoyer la liste des listes *l* dans lesquelles on a inséré l'élément *a* à toutes les positions possibles :

```
let rec distribute a l =
  match l with
   | []   -> [[a]]
   | h::t -> (a::l) :: (mapi (fun _ x -> h::x) 0 (distribute a t));;
```

Enfin notre septième et dernière fonction est *permute* qui utilise *flatten*, *mapi* et *distribute* pour renvoyer la liste de toutes les permutations d'une liste :

```
let rec permute l =
  match l with
   | []   -> [[]]
   | h::t -> flatten (mapi (fun _ -> distribute h) 0 (permute t));;
```

**Remarque:**

Les opérations sur les listes sont toujours *non destructives*, la structure des listes passées en arguments est toujours préservée, les listes retournées en résultat sont fraîchement allouées dans le *tas* (la mémoire globale) par le constructeur (::), on dit que la liste est une structure de données *immutable*. D'une façon générale la programmation fonctionnelle privilégie plutôt les données immutables alors que la programmation impérative privilégie plutôt les données *mutables*. Les données immutables bénéficient de la propriété dite de *substituabilité* qui facilite un raisonnement équationnel assez similaire à celui utilisé en mathématiques. Cette facilité de raisonner sur le texte du programme et non sur son exécution améliore grandement la lisibilité du code. De plus la rigueur du raisonnement par récurrence fait que le texte du programme se rapproche naturellement du raisonnement qui établit la preuve de sa correction.

**Remarques:**

- toute fonction de type *'a → 'a* est isomorphe à la fonction identité
- toute fonction de type *'a → 'b → 'a* est un opérateur associatif à gauche
- toute fonction de type *'a → 'b → 'b* est un opérateur associatif à droite

**Règle de typage (liaison d'une variable de type)**
Cette règle explicite l'instanciation d'un type *polymorphe* par un autre type *monomorphe* ou *polymorphe* :

- la variable *'a* peut être liée à n'importe quel type *T* ne contenant pas d'occurrence libre de la variable *'a*
- la variable *'a* peut être liée à deux types $T_1$ et $T_2$ si et seulement si le type $T_1$ est identique au type $T_2$

## 18. Les couples

Les couples (*pairs*) correspondent à un produit cartésien de valeurs, il combinent deux valeurs de types quelconques sous la forme d'une seule paire. Les deux éléments d'un couple sont séparés par une virgule. Les couples sont utilisés notamment :

- comme des containers
- pour retourner deux valeurs
- pour le filtrage multiple (nous avons déjà parlé du filtrage sur plusieurs valeurs)

La virgule est un constructeur mais nous allons exposer son type comme nous l'avons fait pour le constructeur

de listes:

```
# fun a b -> (a,b);;
- : 'a -> 'b -> 'a * 'b = <fun>
```

Deux remarques:

- le résultat contient deux paramètres de type *'a* et *'b*, un couple peut donc être hétérogène
- les parenthèses ne sont pas indispensables mais elles sont souvent utiles surtout si on ne connaît pas bien la priorité des opérateurs OCaml

On retiendra que l'opérateur infixe * est le constructeur du *type produit* $\alpha * \beta$ .

Le **let**, le **let rec** et les déclarations de fonctions peuvent filtrer les couples.

En tant que containeurs les couples peuvent représenter les nombres complexes :

```
let add_z (a1,b1) (a2,b2) = (a1+.a2,b1+.b2);;
let mul_z (a1,b1) (a2,b2) = (a1*.a2-.b1*.b2,a1*.b2+.a2*.b1);;
```

Un couple peut également être utilisé pour retourner deux valeurs, c'est par exemple le cas de la fonction predefinie *modf* qui renvoie un couple de **float** :

```
# modf;;
- : float -> float * float = <fun>
```

Les deux composantes du couple renvoyé peuvent être récupérées par un **let**-filtrant, par exemple cette expression extrait la partie decimale de *pi* :

```
# let pi = 3.14159 in
  let (d,i) = modf pi in d;;
- : float = 0.141589999999999988
```

Parfois un couple cumule les deux utilités, une fonction renvoie deux valeurs mais ces valeurs peuvent être interprétées comme une seule valeur composite, par exemple ces valeurs peuvent être les deux bornes d'un intervalle à une dimension.

**Règle de typage (construction d'un couple)**
La règle de typage d'une expression *a*,*b* ne contient qu'un seul cas possible :

- le type d'un couple *a*,*b* est *A* * *B* où *A* est le type de l'expression *a* et *B* est le type de l'expression *b*

## 19. Les huit reines

Le problème des 8 reines est un problème classique, l'exercice consiste à générer toutes les façons de positionner 8 reines sur un échiquier sans qu'aucune ne puisse en prendre une autre. Nous allons résoudre ce problème plus facilement qu'il n'y paraît.

Supposons qu'une reine soit sur la case *(xa,ya)* et une autre sur la case *(xb,yb)*, alors le prédicat *check_queens* nous dit si ces deux reines sont en position de prise :

```
let check_queens (xa,ya) (xb,yb) =
  (xa = xb) or (ya = yb) or
  (xa - xb) = (ya - yb)  or
  (ya - yb) = (xb - xa);;
```

Cela résoud-il notre problème, apparemment non. Mais réfléchissez un instant aux fonctions polymorphes que nous venons de développer, elles ont le pouvoir d'étendre la sémantique d'une fonction aussi loin qu'il est nécessaire pour la résolution de notre problème.

En effet, considérez que :

- il n'y a forcément qu'une reine par colonne
- il n'y a forcément qu'une reine par ligne
- nous appelons reine *n* la reine placée sur la *n*-ième ligne, alors *permute [1;2;3;4;5;6;7;8]* place les reines 1 à 8 sur toutes les permutations de colonnes possibles
- nous pouvons alors utiliser *mapi* pour créer les couples de coordonnées *(colonne,ligne)*
- pour détecter une liste de reines qui ne serait pas une solution nous pouvons appliquer le test *exists_commutative check_queens* car le prédicat *check_queens* est commutatif
- nous pouvons utiliser *filter* pour ne garder que les listes de reines qui sont des solutions du problème

Ce qui nous donne :

```
let safe_boards queens =
  filter
    (fun l -> exists_commutative check_queens l = false)
    (mapi
      (fun _ l -> mapi (fun a b -> a,b) 1 l)
      1
      (permute queens)
    );;

safe_boards [1;2;3;4;5;6;7;8];;
```

Et comme la permutation nous garanti que chaque reine est sur un couple *(colonne,ligne)* unique nous pouvons simplifier le prédicat *check_queens* comme ceci :

```
let check_queens (xa,ya) (xb,yb) =
  (xa - xb) = (ya - yb) or
  (ya - yb) = (xb - xa) ;;
```

**Remarque:** les opérateurs logiques **or** et **&&** sont *semi-stricts*, si la valeur de leur premier argument suffit à établir le résultat alors ils n'évaluent pas leur second argument. De ce fait la fonction *check_queens* ci-dessus est aussi efficace que la fonction *check_queens* ci-dessous.

```
let check_queens (xa,ya) (xb,yb) =
  if (xa - xb) = (ya - yb) then true
  else (ya - yb) = (xb - xa) ;;
```

## 20. Le tri fusion

Encore un exemple, pour illustrer la récursion et la démarche *diviser pour régner*, et encore une fonction polymorphe sur les listes. Celle-ci trie une liste par ordre croissant. Elle tire son polymorphisme de celui de la fonction (>) :

```
# (>);;
- : 'a -> 'a -> bool = <fun>
```

En OCaml toutes les fonctions de comparaison sont des prédicats polymorphes, en effet, si *'a* est **int** ou **float** alors la comparaison a son sens mathématique habituel, avec les types **char** et **string** les données sont comparées suivant l'ordre alphabétique, sinon les données sont comparées selon une méthode structurelle (voir la documentation en ligne du module *Pervasives* pour plus de détails).

Le tri-fusion polymorphe nécessite trois fonctions auxiliaires.

La fonction *merge_list* fusionne deux listes *a* et *b* déja triées et renvoie leur union, c'est-à-dire une liste triée qui contient à la fois les éléments de *a* et ceux de *b* :

```
let rec merge_list a b =
  match a,b with
  | [],_ -> b
  | _,[] -> a
  | ha::ta,hb::tb ->
      if ha < hb then ha::(merge_list ta b)
      else hb::(merge_list a tb);;
```

Le premier élément d'une liste est l'élément de rang zéro.
La fonction *odd_list* renvoie la liste des éléments de rang impair de la liste *l* :

```
let rec odd_list l =
  match l with
  | []  -> []
  | [a] -> []
  | a::b::l -> b::odd_list l;;
```

La fonction *even_list* renvoie la liste des éléments de rang pair de la liste **l** :

```
let rec even_list l =
  match l with
  | []   -> []
  | a::l -> a::odd_list l;;
```

Enfin, la fonction *merge_sort* est la fonction du tri-fusion d'une liste *l*.
Elle procède en trois étapes :

- elle scinde la liste *l* en deux listes
- elle trie ces deux listes
- elle fusionne les deux listes triées en une seule

Cette décomposition récursive du processus de tri s'arrête nécessairement puisque l'on sait trier une liste vide et une liste qui contient un seul élément. Sinon la liste contient au moins deux éléments et alors on applique le procédé décrit ci-dessus. Voici le code de cette fonction :

```
let rec merge_sort l =
  match l with
  | [] -> []
  | [a] -> [a]
  | _ ->
      let p = even_list l in
      let q = odd_list  l in
      merge_list (merge_sort p) (merge_sort q);;
```

**Remarque:** on aurait pu effectuer la scission d'une liste en une seule étape, à l'aide de couples, comme ceci

```
let rec split_list l =
  match l with
  | [] -> [],[]
  | [a] -> [a],[]
  | a::b::l -> let p,q = split_list l in (a::p,b::q);;

let rec merge_sort l =
  match l with
  | [] -> []
  | [a] -> [a]
  | _ ->
      let p,q = split_list l
      in merge_list (merge_sort p) (merge_sort q);;
```

**Remarque:** il existe également un prédicat *compare a b*, qui effectue une comparaison plus complète que

(<), qui est lui aussi polymorphe, et qui renvoie un entier dont le signe dépend de $a < b$ et $a > b$

```
# compare;;
- : 'a -> 'a -> int = <fun>
```

Ce prédicat pourrait être implémenté ainsi pour les entiers :

```
# let compare_int a b = a - b;;
val compare_int : int -> int -> int = <fun>
```

Bien sûr cette version est *monomorphe*, elle n'accepte que les entiers, mais elle donne une bonne idée du comportement du *compare* polymorphe.

## 21. Les modules

Bien entendu OCaml permet de gérer les gros projets et les projets en équipe, pour cela il fournit un mécanisme de modularité qui offre un espace de nommage (et même beaucoup plus mais cela n'est pas le sujet de ce paragraphe). Ce mécanisme de modularité est d'autant plus indispensable que certaines fonctions sont "trop" polymorphes. Par exemple la fonction *compare* (celle du module *Pervasives*) est définie d'une façon trop générique, pour des types composites elle ne renverra pas forcément le résultat escompté, il faudra alors définir une fonction *compare* plus spécialisée qui renvoie le résultat attendu pour ce nouveau type. Il devient alors logique, pour ne pas écraser le *compare* habituel, de réunir la définition du nouveau type composite et la définition de sa fonction *compare* dans un même module.

Le mot-clé **module** permet de limiter la portée du **let**. Ainsi, en introduisant un **module** *List* on pourra limiter la portée des fonctions *filter*, *exists*, *flatten* sur les listes, ceci permettrait par exemple de réutiliser ces noms dans un hypothétique **module** *Tree*. Choisir des noms identiques pour des fonctionnalités identiques est une bonne pratique qui participe à augmenter la lisibilité des programmes.

La bonne nouvelle c'est que ce module *List* existe déjà, c'est le fichier *Objective Caml/libs/list.ml*, et il contient les fonctions *List.filter*, *List.exists*, et *List.flatten* telles que nous les avons défines, avec des implémentations récursives terminales quand cela est possible. Pour une spécification plus complète du module *List* veuillez consulter sa documentation en ligne.

Parfois la récursion terminale n'est pas directement possible, c'est le cas de la fonction *List.append* qui est la version *préfixe* de l'opérateur de concaténation de listes (@) que nous avons déjà utilisé. Il est possible d'écrire une version récursive terminale de la concaténation, en ajoutant un accumulateur, mais comme nous l'avons vu ceci peut avoir pour effet de renvoyer un résultat qui est le miroir du résultat voulu, dans le cas de la concaténation de liste ceci nous donne la fonction *List.rev_append* :

```
# List.rev_append;;
- : 'a list -> 'a list -> 'a list = <fun>
```

Cette fonction a le même type que *List.append*, elle est récursive terminale et elle concatène bien ses deux arguments mais son résultat est renversé, nous verrons néanmoins que cette fonction n'est pas inutile car nous n'aurons pas forcément besoin de renverser son résultat à l'aide de *List.rev*.

Vous l'avez deviné, *List.rev l* est la fonction qui renverse la liste *l*, elle est récursive terminale :

```
let rev l = rev_append l [];;
```

Voilà bien un premier exemple qui montre que *List.rev_append* n'est pas l'aberration qu'on pourrait croire.

Le module *List* contient également les fonctions *List.map*, *List.for_all* et *List.sort*.

La fonction *List.map* ressemble à notre fonction *mapi* à la différence près que *List.map* ne tient pas compte de

la position des éléments dans la liste :

```
# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

La fonction *List.map* applique la fonction *('a → 'b)* sur tous les éléments de la liste *('a list)* et renvoie une liste *('b list)*.
Cette fonction n'est pas récursive terminale, sa version récursive terminale (mais renversée) est *List.rev_map*.

La fonction *List.for_all* ressemble fort à la fonction *List.exists* et d'ailleurs on pourrait l'implémenter ainsi :

```
let for_all p l = List.exists (fun x -> p x = false) l = false;;
```

Elle dit simplement si les éléments d'une liste *l* vérifient tous le prédicat *p*.

Enfin la fonction *List.sort* ressemble fort à notre fonction *merge_sort* à la différence qu'en plus elle attend pour premier argument une fonction similaire à *compare* et c'est cette fonction qui lui sert de critère pour le tri :

```
# List.sort;;
- : ('a -> 'a -> int) -> 'a list -> 'a list = <fun>
```

Il reste le cas de nos fonctions *mapi*, *exists_commutative* et *permute*, et comme elles ont déjà prouvé leur utilité nous les conserverons dans un nouveau module que l'on nommera *ListXP* :

```
module ListXP = struct
  let rec mapi f n l = ...
  let rec exists_commutative p l = ...
  let rec permute l = ...
end;;
```

La portée de leurs déclarations est alors délimitée par les mots-clé **struct** et **end**.
Nous en profitons pour améliorer leurs implémentations en fournissant une version récursive terminale *rev_mapi* et en utilisant les fonctions du module *List* :

```
module ListXP = struct

  let rec mapi f n l =
    match l with
    | []   -> []
    | h::t -> f n h::mapi f (n+1) t;;

  let rev_mapi f n l =
    let rec loop l n acc =
      match l with
      | []   -> acc
      | h::t -> loop t (n+1) (f n h::acc)
    in loop l n [];;

  let rec exists_commutative p l =
    match l with
    | []   -> false
    | h::t -> (List.exists (p h) t) or exists_commutative p t;;

  let rec distribute a l =
    match l with
    | []   -> [[a]]
    | h::t -> (a::l) :: (List.map (fun x -> h::x) (distribute a t));;

  let rec permute l =
    match l with
    | []   -> [[]]
    | h::t -> List.flatten (List.map (distribute h) (permute t));;

end;;
```

Notons qu'un module est une expression et a donc un type (qu'on appelle aussi *signature*), voici la signature de notre module *ListXP* telle que l'interpréteur OCaml l'a inférée :

```
module ListXP :
  sig
    val mapi : (int -> 'a -> 'b) -> int -> 'a list -> 'b list
    val rev_mapi : (int -> 'a -> 'b) -> int -> 'a list -> 'b list
    val exists_commutative : ('a -> 'a -> bool) -> 'a list -> bool
    val distribute : 'a -> 'a list -> 'a list list
    val permute : 'a list -> 'a list list
  end
```

**Remarque:** le nom d'un module doit obligatoirement commencer par une majuscule.

## 22. Le traitement des erreurs

Parfois une fonction ne peut pas renvoyer de résultat parce que ses arguments sont incompatibles avec le calcul à effectuer.
Il existe 3 façons simples de régler ces cas, par ordre de préférence:

- la fonction *assert*, déjà vue
- la fonction *invalid_arg* appliquée à une chaîne de message d'erreur
- la fonction *failwith* appliquée à une chaîne de message d'erreur

Voici un cas où l'on privilégiera la fonction *assert* parce qu'il n'y pas de filtrage:

```
# abs min_int;;
- : int = -1073741824
# let safe_abs n =
    assert(n > min_int);
    abs n;;
val safe_abs : int -> int = <fun>
```

Voici un cas où l'on privilégiera au contraire la fonction *invalid_arg* parce que le cas à éliminer est déjà pris en compte par un filtrage exhaustif :

```
# let maximum l =
    let rec loop a l =
      | []   -> a
      match l with
      | b::t -> if a > b then loop a t else loop b t
    in match l with
    | []   -> invalid_arg "maximum";;
    | a::t -> loop a t
val maximum : 'a list -> 'a = <fun>
```

On réservera l'usage de la fonction *failwith* aux cas où les arguments sont bien dans le domaine de définition.

**Remarque:** les fonctions *failwith* et *invalid_arg* sont de type **string** → *'a*, un type à l'évidence totalement dépourvu de sémantique, avec pour seul avantage qu'il trouve sa place n'importe où.
La bonne nouvelle c'est qu'un déclenchement d'erreur pourra se substituer à n'importe quelle expression.
La mauvaise nouvelle c'est que n'importe quelle expression suffisamment obscure pourra cacher un déclenchement d'erreur.

## 23. Les enregistrements

Les enregistrements (*records*) correspondent eux aussi à un produit cartésien de valeurs et ont la même utilité que les *couples* ou *n-uplets*. La principale différence c'est qu'avec un *n-uplet* un élément est identifié par sa position alors qu'avec un enregistrement un élément est identifié par son étiquette (*label*).

Une autre différence c'est le typage, OCaml doit pouvoir typer la totalité d'un *couple* alors que pour un *enregistement* il lui suffit de typer chacune de ses *composantes*. Nous étudierons plus tard en quoi le typage limite parfois les *couples* et avantage les *enregistrements*.

Typiquement la déconstruction d'un couple se fait plutôt par filtrage alors que la déconstruction d'un enregistrement se fait plutôt par ses *accesseurs*. Le filtrage est plus naturel pour le calcul symbolique, les accesseurs sont plus naturels pour les calculs numériques.

En voici un exemple, nous déclarons d'abord les types *vector* et *point*, en 3 dimensions :

```
type vector = {x: float; y: float; z: float};;
type point  = vector;;
```

Le type *point* n'est qu'un *alias* (un synonyme) pour le type *vector*, parce qu'un point peut être désigné par le vecteur qui le sépare de l'origine. Nous pouvons maintenant implémenter les opérations vectorielles classiques, l'addition, la soustraction, le produit scalaire et le carré scalaire :

```
let add a b =
  {x = a.x +. b.x; y = a.y +. b.y; z = a.z +. b.z};;

let sub a b =
  {x = a.x -. b.x; y = a.y -. b.y; z = a.z -. b.z};;

let scalar_dot a b =
  a.x *. b.x +.
  a.y *. b.y +.
  a.z *. b.z;;

let scalar_sqr v =
  v.x *. v.x +.
  v.y *. v.y +.
  v.z *. v.z;;
```

Nous ajoutons également un constructeur qui fabrique un vecteur à partir de deux points :

```
let make (a: point) (b: point) =
  {x = b.x -. a.x; y = b.y -. a.y; z = b.z -. a.z};;
```

On remarquera la notation (*expr: type)* qui demande explicitement au compilateur d'essayer d'inférer le type mentionné. Ceci n'a rien à voir avec le transtypage dans les langages pourvus d'un typage moins fort que celui d'OCaml, si le compilateur ne parvient pas à inférer le type demandé alors une erreur de typage sera reportée. Ici on demande au compilateur d'inférer le type *point* plutôt que le type *vector*. Cette façon d'orienter le typage s'appelle une *annotation de type*.

Nous voilà prêts pour affronter la problématique de base du lancé de rayons. Un *rayon* est une demi-droite paramétrée *s + tv* , elle a pour source le point *s* et pour direction le vecteur *v*.
Une *sphère* a pour centre un point *c* et pour rayon un réel *r*. L'intersection d'un rayon avec une sphère est soit vide, soit un interval *(t1,t2*) de paramètres du rayon *s + tv*.

```
# let ray_sphere (s: point) v (c: point) r =
    (* Carré du rayon de la sphère: *)
    let r2 = r *. r   in
    (* Vecteur de la source au centre de la sphère: *)
    let sc = make s c in
    (* Carré de la distance de la source au centre de la sphère: *)
    let sc2 = scalar_sqr sc   in
    let v2  = scalar_sqr v    in
    let p   = scalar_dot sc v in
    let p2  = p *. p    in
    (* Carré de la distance du rayon au centre de la sphère: *)
    let d2 = sc2 -. p2 /. v2 in
    let delta = r2 -. d2 in
    if delta <= 0. then
      failwith "empty"
    else
```

```
        let sq = sqrt (delta /. v2) in
        let t1 = p /. v2 -. sq in
        let t2 = p /. v2 +. sq in
        if t2 < 0. then
          failwith "empty"
        else
          (max 0. t1, t2);;
  val ray_sphere : point -> vector -> point -> float -> float * float = <fun>
```

**Règle de typage (synonymie de type)**
Cette règle précise le sens de la condition "même type" en présence d'*aliasing* de type :

- un type est le même type qu'un type qui est son *alias*

**Règle de typage (typage explicite)**
La règle de typage d'une expression explicitement typée *(expr: E)* distingue deux cas possibles :

- ou bien le type de *expr* est le même type que *E* alors l'expression est de type *E*
- ou bien l'expression est mal typée

## 24. Les performances

Bien que les langages fonctionnels soient plus proches de la spécification que les langages impératifs ils n'en sont pas moins exempts des considérations de performances. Il y trois types de ressources à économiser :

- la *pile* est une ressource mémoire locale et limitée
- le *tas* désigne la mémoire globale
- les *processeurs* désignent les ressources d'exécution

Du fait de leur système de typage bien plus avancé les langages fonctionnels peuvent mieux capitaliser sur la modularité pour réutiliser les meilleurs algorithmes et les meilleures structures de données les plus adaptés aux spécificités du domaine. Du coup le programmeur fonctionnel est généralement moins sous pression quant aux gains de performances liés à l'économie des ressources d'exécution.

Il faudra tout de même prendre garde à partager les calculs autant que possible, par exemple

```
let square f x = f x *. f x;;
```

Devra s'écrire :

```
let square f x = let y = f x in y *. y;;
```

Sinon le calcul, potentiellement long, de *f x* se fera deux fois au lieu d'une seule.

La contrepartie d'une meilleure maîtrise des ressources d'exécution c'est que le programmeur fonctionnel doit faire davantage attention à économiser les ressources mémoire, les langages fonctionnels sont des langages de haut niveau, ils ont une tendance naturelle à consommer plus de *pile* et de *tas* que les langages de bas niveau.

L'astuce principale pour économiser la *pile* consiste à privilégier la *récursion terminale*, par exemple si l'on veut concaténer deux listes et que seuls comptent les éléments (pas leur ordonnancement) alors on utilisera *List.rev_append* plutôt que *List.append* parce que *List.rev_append* est récursive terminale et pas *List.append*. Il en va de même pour *List.rev_map* comparée à *List.map*.

Dans d'autres cas il faudra utiliser l'*accumulation* si cela est nécessaire, notamment si l'on envisage de traiter des chaînes de longueur potentiellement élevée. Souvent il sera plus aisé de faire l'implémentation en deux étapes, d'abord écrire une version récursive, la tester, puis ensuite écrire une nouvelle version avec

accumulation.

Une astuce de nature similaire s'applique au *tas*, elle consiste à remplacer les listes normales par des listes *paresseuses*, ces listes sont détruites au fur et à mesure qu'elles sont construites de sorte que seule leur tête est présente dans le tas au lieu de toute leur longueur. Nous n'aborderons pas les listes paresseuses dans ce chapitre.

Une autre astuce appréciable pour économiser le *tas* consiste à éviter la duplication des listes et des couples quand elle n'est pas strictement indispensable.

Pour les listes cela peut se faire en cumulant deux fonctionnalités en une seule, par exemple si on effectue des grandes quantités de ce traitement :

```
List.rev_map f (List.filter p l);;
```

Alors on pourra avantageusement le remplacer par :

```
let rev_map_filter p f l =
  let rec loop l acc =
    match l with
    | [] -> acc
    | h::t ->
        if p h then loop t (f h::acc)
        else loop t acc
  in loop l [];;
```

Ce qui permet d'économiser la production de listes par *filter* qui seraient aussitôt consommées par *rev_map*.

Une situation similaire concerne les couples, dans les cas où soit on veut pouvoir à la fois les détruire et les conserver.
Supposons que nous voulions ordonner tous les couples d'une liste, que le second élément du couple soit toujours supérieur au premier, voici une façon de faire :

```
let ordered_pair (a,b) = if a < b then (a,b) else (b,a);;
let list_ordered_pairs l = List.map ordered_pair l;;
```

Le cas dérangeant c'est le cas $a < b$ car alors nous détruisons le couple pour en construire un autre en tous points identique, en conséquence le *tas* est davantage sollicité qu'il ne le devrait. Afin d'éviter ces cas il existe le mot-clé **as** qui permet d'apparier une variable (ici *p*) à la totalité d'un motif (ici *a,b*) :

```
let ordered_pair (a,b as p) = if a < b then p else (b,a);;
```

Avec cette version seuls les "nouveaux" couples sont alloués, les anciens sont réutilisés.

# Partie III: Application à la gestion d'inventaires

Pour conclure ce tour d'horizon des techniques de programmation élémentaires, des listes polymorphes, de la modularité et du traitement des erreurs, nous allons étudier un projet complet et réaliste qui fait usage de toutes ces connaissances.

25. Domaine des inventaires
26. Opérations élémentaires sur les inventaires
27. Opérations avancées sur les inventaires
28. Cas d'usage du module Inventory

## Partie IV: Application à la théorie des jeux

29. Le jeu des tétrominos
30. Le tableau de jeu
31. La génération des coups
32. Le calcul de la valeur d'un tableau

## Partie V: La programmation impérative

33. Les champs mutables
34. Les références
35. Les boucles
36. Les tableaux
37. Les entrées-sorties
38. *(réservé)*

### 33. Les champs mutables

Jusqu'à présent nous avons manipulé beaucoup de listes et nous avons laissé de côté les structures plus générales comme les *arbres* et les *graphes*. Rattrapons cet oubli.

Commençons par les *arbres*.

Un programmeur habitué à des langages à typage dynamique pourrait imaginer que la structure *list* suffira à représenter les *arbres n-aires* (*n-ary trees*) par exemple de la façon suivante :

```
# [[1];[[2;3];[4;5]]];;
Characters 6-11:
  [[1];[[2;3];[4;5]]];;
        ^^^^^
This expression has type 'a list but is here used with type int
```

Et pourtant cette tentative est immédiatement rejetée par OCaml, pourquoi ?
Parce que le type *'a list* est obligatoirement *homogène*, or ici le 1$^{ier}$ élément de notre liste est de type *int list* alors que le 2$^{nd}$ élément est de type *int list list*. Notre liste est donc *hétérogène*, elle est mal typée.

Pensons plutôt notre arbre *récursivement*. Nous aimerions que chaque noeud puisse contenir un élément de type *'a* et une liste de sous-arbres du même type que lui-même. Nous avons donc besoin de deux composantes, ce qui nous laisse le choix de l'implémentation, soit un *couple*, soit un *enregistrement*. Nous pouvons alors choisir entre l'une des ces deux déclarations de types.

Ou bien à l'aide d'un enregistrement :

```
# type 'a tree = {item: 'a; sub: 'a tree list};;
type 'a tree = { item : 'a; sub : 'a tree list; }
```

Ou bien à l'aide d'un couple:

```
# type 'a tree = 'a * ('a tree list);;
Characters 4-34:
  type 'a tree = 'a * ('a tree list);;
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
The type abbreviation tree is cyclic
```

En fait non, nous ne pouvons pas utiliser les *couples*. La raison peut paraître obscure pourtant elle devient limpide si nous évaluons ces deux expressions :

```
# 1,[];;
- : int * 'a list = (1, [])
# 1,[2,[];3,[]];;
- : int * (int * 'a list) list = (1, [(2, []); (3, [])])
```

Le type que nous voudrions définir dépend de la structure de données. Il croît avec elle, nous n'avons pas de moyen de définir le *domaine* d'une façon générale. Au contraire, avec le type *enregistrement*, *'a tree* est le domaine. Il est déjà défini et donc le type *'a tree list* est disponible pour le champ *sub*. On dit que le type enregistrement est un *type récursif*, au contraire le type n-uplet n'est pas un *type récursif*.

Les fonctions sur les arbres *rev_tree*, *map_tree*, *exists_tree*, *flatten_tree*, suivent de façon assez immédiate leurs correspondantes sur les listes :

```
let rec rev_tree t =
  {item = t.item; sub = List.rev_map rev_tree t.sub};;

let rec map_tree f t =
  {item = f t.item; sub = List.map (map_tree f) t.sub};;

let rec exists_tree p t =
  (p t.item) or List.exists (exists_tree p) t.sub;;

let rec flatten_tree t =
  t.item::List.flatten (List.map flatten_tree t.sub);;
```

Passons maintenant aux *graphes dirigés avec racine*.

Un *graphe dirigé avec racine* (*rooted directed graph*) est très semblable à un *arbre n-aire* avec la possibilité supplémentaire de partager des noeuds, de surcroît des *cycles* peuvent exister, de ce fait les algorithmes de parcours valables sur un arbre ne sont plus valables sur un *graphe dirigé* car un cycle pourrait piéger le parcours dans une boucle infinie.

Afin d'éviter les boucles infines, on doit arrêter la récursion quand on rencontre un sommet déjà visité. Il nous faut donc trouver un moyen de marquer les sommets déjà visités pendant la traversée. Il nous faut également nettoyer toutes les marques après la traversée. Voici comment on procède: on déclare un *graphe dirigé avec racine* de la même façon qu'un *arbre n-aire* tout en lui ajoutant un champ *time* qui est déclaré **mutable** :

```
type 'a rooted_digraph =
  {point: 'a; arrows: 'a rooted_digraph list; mutable time: int};;
```

Essentiellement il s'agit de relier des sommets (*point*) par des flèches (*arrows*) et d'enregistrer l'horaire (*time*) de la dernière visite.

**mutable** signifie que la valeur du champ *time* pourra changer. Nous verrons qu'heureusement ce changement fait l'objet de règles de bonne conduite. Mais d'abord nous allons déclarer l'horloge qui fournira notre temps de référence :

```
let clock = {point = 1; arrows = []; time = 0};;
```

Désormais l'heure sera la valeur du champ *clock.time*.

Notons que, puisque *clock* est le sommet d'un *rooted_digraph*, il est logique que chaque sommet d'un *rooted_digraph* puisse être considéré comme une horloge, chacune de ces horloges pourra soit être à l'heure soit être en retard par rapport à *clock* notre horloge de référence.

Nous définissons maintenant trois fonctions horaires :

- *clock_tick t* fait progresser l'horloge de référence de *t* unités
- *clock_slow h* nous dit si l'horloge *h* a pris du retard
- *clock_sync h* remet l'horloge *h* à l'heure de référence

```
let clock_tick t =
  assert(t > 0);
  clock.time <- clock.time + t;;

let clock_slow h =
  h.time < clock.time;;

let clock_sync h =
  h.time <- clock.time;;
```

Les deux fonctions *clock_tick* et *clock_sync* garantissent :

- que toute horloge est strictement croissante
- que toute horloge est synchronisable avec l'heure de référence
- que l'heure de référence est également une heure maximale, aucune horloge ne pourra la dépasser

Ces deux fonctions sont précisément celles qui justifient la *mutabilité* du champ *time*, le qualificatif **mutable** signifie qu'on peut assigner ce champ à l'aide de l'opérateur ←, cette opération d'assignation renvoie un élément du type **unit**. En fait le type **unit** ne contient qu'un seul élément, à savoir **()**, par conséquent l'opérateur d'assignation renvoie toujours la valeur **()**. Plus généralement, toute expression de type **unit**, c'est-à-dire de valeur **()** est considérée comme une instruction valide. En particulier l'opérateur infixe de séquencement des instructions est le point-virgule, il est associatif à droite et de type *unit* → *'a* → *'a*, de sorte que la valeur d'une séquence d'instruction est la valeur de l'expression qui la suit immédiatement.

**Preuve:**

- la flèche → est associative à droite, par conséquent le point-virgule est de type *unit* → (*'a* → *'a*)
- *'a* → *'a* est forcément le type de l'identité

La fonction d'un *rooted_digraph* la plus élémentaire est *length_graph* qui donne le nombre de sommets présents dans le graphe :

```
let length_graph g =
  let rec loop acc g =
    if clock_slow g then begin
      clock_sync g;
      List.fold_left loop (acc + 1) g.arrows
    end else
      acc
  in begin
    clock_tick 1;
    loop 0 g
  end;;
```

Les mots-clés **begin** … **end** encadrent une séquence d'instructions séparées par un point-virgule.
Une instruction est une expression du type **unit**, sauf pour la dernière d'entre elles qui est d'un type quelconque, c'est le type de cette dernière instruction qui devient le type de toute la séquence, car les séquences sont typées elles aussi. La valeur **()** est la seule et unique valeur de type **unit**. En fait l'expression **begin** $instr_1 ; \ldots ; instr_n$ **end** a exactement la même sémantique que l'expression **let ()** = $instr_1$ **and** … **and ()** = $instr_{n-1}$ **in** $instr_n$.

**Remarque:** nous ne décrirons pas la fonction *List.fold_left*, pour plus d'information sur cette fonction veuillez vous reporter à la documentation en ligne du *module List*.

Les fonctions *exists_graph* et *flatten_graph* sont bâties sur le même modèle que *length_graph* :

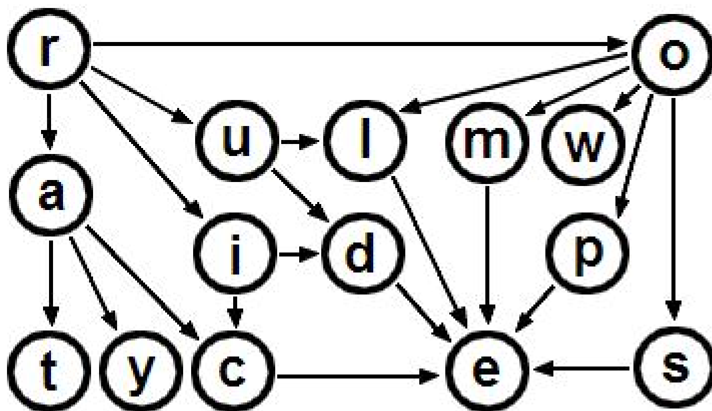
```
let exists_graph p g =
  let rec loop g =
    if clock_slow g then begin
      clock_sync g;
      (p g.point) or List.exists loop g.arrows
    end else
      false
  in begin
    clock_tick 1;
    loop g
  end;;

let flatten_graph g =
  let rec loop acc g =
    if clock_slow g then begin
      clock_sync g;
      List.fold_left loop (g.point::acc) g.arrows
    end else
      acc
  in begin
    clock_tick 1;
    List.rev (loop [] g)
  end;;
```

Dans le cas courant d'un graphe dirigé *acyclique*, la fonction *flatten_graph* effectue ce que l'on appelle un *tri topologique*.

Mais au fait, comment construit-on un *rooted_digraph* ?
On construit un graphe à l'aide de **let rec** ... **and** ... **and** ... **in** *root* où *root* désigne le sommet racine du graphe.

Exemple avec un automate qui reconnaît les mots anglais *race*, *rat*, *ray*, *rice*, *ride*, *role*, *rome*, *rope*, *rose*, *row*, *rude*, *rule* :




La fonction *r_graph* construit le graphe de cet automate et place les lettres sur les sommets correspondants :

```
let r_graph () =
  let
  rec a = {point= 'a'; arrows= [t;y;c];     time=0}
  and c = {point= 'c'; arrows= [e];         time=0}
  and d = {point= 'd'; arrows= [e];         time=0}
  and e = {point= 'e'; arrows= [];          time=0}
  and i = {point= 'i'; arrows= [c;d];       time=0}
  and l = {point= 'l'; arrows= [e];         time=0}
  and m = {point= 'm'; arrows= [e];         time=0}
  and o = {point= 'o'; arrows= [l;m;p;s;w]; time=0}
  and p = {point= 'p'; arrows= [e];         time=0}
  and r = {point= 'r'; arrows= [a;i;o;u];   time=0}
  and s = {point= 's'; arrows= [e];         time=0}
```

```
   and t = {point= 't'; arrows= [];         time=0}
   and u = {point= 'u'; arrows= [d;l];      time=0}
   and w = {point= 'w'; arrows= [];         time=0}
   and y = {point= 'y'; arrows= [];         time=0}
   in  r;;
```

La mauvaise nouvelle c'est que l'interpréteur *OCaml* est bien maladroit quand il s'agit d'afficher un tel graphe, il parcourt la structure comme un arbre en affichant plusieurs fois les sommets partagés. Pire, si notre graphe contenait une seule boucle elle serait affichée comme une structure infinie, jusqu'à épuisement de la capacité de la console.
C'est pourquoi nous n'avons pas défini le graphe comme une valeur mais comme une fonction qui attend un élément **unit** avant de renvoyer le graphe, de cette façon nous coupons court à toute tentative d'affichage tout en définissant un raccourci bien commode pour tester nos fonctions sur ce graphe :

```
# length_graph (r_graph ());;
- : int = 15
# exists_graph ((=) 'n') (r_graph ());;
- : bool = false
# exists_graph ((=) 'o') (r_graph ());;
- : bool = true
# flatten_graph (r_graph ());;
- : char list =
['r'; 'a'; 't'; 'y'; 'c'; 'e'; 'i'; 'd'; 'o'; 'l'; 'm'; 'p'; 's'; 'w'; 'u']
```

La bonne nouvelle c'est qu'il n'est pas bien difficile d'écrire une procédure de décision pour les automates :

```
# let rec recognized l auto =
  match l with
  | [] -> false
  | c::[] -> auto.point=c && auto.arrows=[]
  | c::l  -> auto.point=c && List.exists (recognized l) auto.arrows
  ;;
val recognized : 'a list -> 'a rooted_digraph -> bool = <fun>
# recognized ['r';'o';'p';'e'] (r_graph ());;
- : bool = true
# recognized ['r';'a';'t';'e'] (r_graph ());;
- : bool = false
```

## 34. Les références

Comme nous l'avons mentionné au paragraphe *déclarations de variables*, il n'est pas possible en OCaml de modifier la valeur d'une variable. D'autre part nous venons de voir qu'il est possible de modifier un champ **mutable** d'un enregistrement. On peut dès lors imaginer un type *conteneur* pour un champ mutable. C'est précisément ce qu'est le type prédéfini *'a ref*, un type enregistrement polymorphe contenant un seul champ mutable nommé *contents* :

```
# type 'a ref = {
    mutable contents : 'a;
  };;
type 'a ref = { mutable contents : 'a; }
```

Ce type possède un *constructeur* prédéfini :

```
# let ref v = {contents = v};;
val ref : 'a -> 'a ref = <fun>
```

Évidemment la fonction duale est un *accesseur*, également prédéfini, en position *préfixe* :

```
# let (!) r = r.contents;;
val ( ! ) : 'a ref -> 'a = <fun>
```

Mais la nouveauté réside bien entendu dans les *méthodes*, également prédéfinies, l'*assignation* (en position *infixe*) et ses deux variantes plus spécialisées, l'*incrémentation* et la *décrémentation* :

```
# let (:=) r v = r.contents <- v;;
val ( := ) : 'a ref -> 'a -> unit = <fun>

# let incr r = r.contents <- succ r.contents;;
val incr : int ref -> unit = <fun>

# let decr r = r.contents <- pred r.contents;;
val decr : int ref -> unit = <fun>
```

Avec ces outils il n'est toujours pas possible de modifier la valeur d'une variable mais il devient possible de modifier le contenu de la valeur d'une variable, par exemple on pourra déclarer une variable *r1* de type **int ref** :

```
# let r1 = ref 1;;
val r1 : int ref = {contents = 1}
```

Ce qui permettra, à valeur constante, de modifier son contenu, par exemple comme ceci :

```
# let () = (r1 := 4) in !r1;;
- : int = 4
```

Comme **let ()** = $expr_1$ **in** $expr_2$ est équivalent à $expr_1$; $expr_2$ on peut écrire plus simplement:

```
# r1 := 4; !r1;;
- : int = 4
```

Après cette opération la variable *r1* désigne toujours le même enregistrement, cependant le champ *contents* de cet enregistrement a été modifié :

```
# r1;;
- : int ref = {contents = 4}
```

Pour illustrer les possibilités des *références* nous allons implémenter une fonction prédecesseur pour *l'appel de fonction*, en conséquence notre fonction *pred_call* appliquera la fonction *f* non pas à son argument courant mais à son argument au précédent appel, nous devons donc fournir une fonction *f* de type *'a* → *'b* et un argument initial *init* de type *'a*, le résultat sera une nouvelle fonction de type *'a* → *'b* :

```
# let pred_call f init =
  let memo = ref (f init) in
  fun n -> let result = !memo
           in  memo := f n; result;;
val pred_call : ('a -> 'b) -> 'a -> 'a -> 'b = <fun>
```

Ainsi une fonction *successeur* pour le moins étrange pourrait renvoyer le *succ* de son argument au précédent appel :

```
# let pred_succ = pred_call succ 0;;
val pred_succ : int -> int = <fun>
```

Voici un exemple de session :

```
# pred_succ 34;;
- : int = 1
# pred_succ 17;;
- : int = 35
# pred_succ 22;;
- : int = 18
```

Après cette courte présentation de l'utilité des champs **mutable** et des *références* dans un style qui reste somme toute assez fonctionnel, nous allons entrer de plein pied dans le monde de la programmation impérative.

## 35. Les boucles

Il n'existe que deux boucles prédéfinies en OCaml, la boucle **while** et la boucle **for**.

La boucle **while** *expression* **do** *instruction* **done** répète l'*instruction* tant que l'*expression* est égale à **true**.

Ainsi on se convaincra aisément que le code ci-dessous implémente la même fonction *factorial* que celle du paragraphe **II.12** :

```
let factorial n =
  assert(n >= 0);
  let i = ref n and result = ref 1 in
  while !i > 1 do
    result := !result * !i;
    decr i
  done;
  !result;;
```

**Règle d'inférence de type**
La règle d'inférence d'une expression **while** *expression* **do** *instruction* **done** distingue trois cas possibles :

- ou bien *expression* n'est pas de type **bool** alors l'expression est mal typée
- ou bien *instruction* n'est pas de type **unit** alors l'expression est mal typée
- ou bien l'expression est de type **unit**

Dans les cas où l'on connaît par avance le nombre de répétitions on peut économiser un **let** en utilisant une boucle **for** au lieu de la boucle **while**, on a alors deux sortes de boucle **for** à notre disposition, ou bien la boucle ascendante **for** ... **to** ... **do** ... **done** qui incrémente le compteur de boucle :

```
let factorial n =
  assert(n >= 0);
  let result = ref 1 in
  for i = 1 to n do
    result := !result * i
  done;
  !result;;
```

Ou bien la boucle descendante **for** ... **downto** ... **do** ... **done** qui décrémente le compteur de boucle :

```
let factorial n =
  assert(n >= 0);
  let result = ref 1 in
  for i = n downto 1 do
    result := !result * i
  done;
  !result;;
```

Ici, nous ne donnerons pas davantage d'exemples de boucles car ceci fera l'objet du **chapitre VI**.

**Règle d'inférence de type**
La règle d'inférence d'une expression **for** *var* = *start* **(to|downto)** *limit* **do** *instruction* **done** distingue trois cas possibles :

- ou bien *start* n'est pas de type **int** ou encore *limit* n'est pas de type **int** alors l'expression est mal typée
- ou bien *instruction* n'est pas de type **unit** alors l'expression est mal typée
- ou bien l'expression est de type **unit**

**Remarque:** l'expression *limit* est évaluée une fois et une seule pour toute la durée de la boucle **for**.

## 36. Les tableaux

Les *tableaux* (**array**) permettent l'accès à leur *n*-ième élément en temps constant. En outre les éléments d'un tableaux sont toujours assignables. Les fonctions sur les tableaux sont réunies dans le module *Array*, et en particulier:

- *Array.length a* renvoie le nombre d'éléments dans le tableau a, les indices dans un tableau a seront toujours compris entre *0* et *Array.length a - 1*
- *Array.make n x* crée un nouveau tableau de longueur *n* et dont tous les éléments valent *x*
- *Array.get a n* renvoie le *n*-ième élément du tableau *a*, cette fonction possède une forme infixe *a.(n)*
- *Array.set a n x* assigne la valeur *x* au *n*-ième élément du tableau *a*, cette instruction possède une forme infixe *a.(n) ← x*

Ces quelques fonctions sont amplement suffisantes pour implémenter l'équivalent du tri-fusion sur les tableaux.
Pour une spécification plus complète du module *Array* veuillez consulter sa documentation en ligne.

Écrivons d'abord une fonction qui échange deux éléments d'un tableaux *a* :

```
let swap a i j =
  let tmp = a.(i) in
  a.(i) <- a.(j);
  a.(j) <- tmp;;
```

L'algorithme *QiSort* est une sorte d'équivalent du tri-fusion pour un tableau, il est bien décrit sur cette page et son implémentation en OCaml est assez immédiate :

```
let array_sort a =
  let rec qi_sort lower upper =
    let left = ref lower and right = ref upper in
    while !left < !right do
      while !left < !right && a.(!left) < a.(!right) do decr right done;
      if !left < !right then begin
        swap a !left !right;
        incr left;
        while !left < !right && a.(!left) < a.(!right) do incr left done;
        if !left < !right then begin
          swap a !left !right;
          decr right;
        end;
      end;
    done;
    decr left;
    incr right;
    if lower  < !left then qi_sort lower  !left;
    if !right < upper then qi_sort !right upper;
  in qi_sort 0 (Array.length a - 1);;
```

La notation *OCaml* pour les tableaux en extension est comparable à celle des listes en extension, à la différence de la barre verticale qui suit le crochet ouvrant et qui précède le crochet fermant :

```
# let a=[|4;2;5;8;6;9;3;7;0;1|] in array_sort a; a;;
- : int array = [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|]
```

Dans la **partie VI** on développera un **module** complet qui mettra en oeuvre toutes nos nouvelles connaissances sur les références, les tableaux et les boucles, tout cela sans rien mettre de côté de l'*approche qualité* qui a fait ses preuves jusqu'ici.

À signaler: du fait de la proximité conceptuelle des chaînes de caractères (**string**) avec les tableaux de **char**, le module String fournit une interface assez comparable à celle du module Array.
C'est ainsi que les 4 fonctions pour les tableaux ont leurs équivalents exacts pour les chaînes :

- *String.length s* renvoie le nombre de caractères dans la chaîne *s*, les indices dans une chaîne *s* seront toujours compris entre *0* et *String.length s - 1*
- *String.make n c* crée une nouvelle chaîne de longueur *n* et dont tous les caractères valent *c*
- *String.get s n* renvoie le *n*-ième caractère de la chaîne *s*, cette fonction possède une forme infixe *s.[n]*
- *String.set s n c* assigne la valeur *c* au *n*-ième caractère de la chaîne *s*, cette instruction possède une forme infixe *s.[n] ← c*

## 37. Les entrées-sorties

Le module *Pervasives* exporte des fonctions pour l'affichage des valeurs élémentaires :

```
# print_int;;
- : int -> unit = <fun>
# print_float;;
- : float -> unit = <fun>
# print_char;;
- : char -> unit = <fun>
# print_string;;
- : string -> unit = <fun>
# print_newline;;
- : unit -> unit = <fun>
```

Parmi ces fonctions seule *print_newline* force l'affichage, les autres ne font que remplir un cache d'affichage qui sera vidé lors d'un appel *print_newline ()*. On peut aussi vider explicitement le cache, et ainsi forcer l'affichage sans passage à la ligne, à l'aide de *flush stdout*.

Le module *Pervasives* exporte aussi le type *in_channel* qui désigne un fichier accessible en lecture ainsi que plusieurs fonctions d'accès parmi lesquelles on pourra retenir au moins les trois suivantes :

- *open_in : string -> in_channel* qui ouvre en lecture le fichier texte situé au chemin d'accès spécifié
- *input_line : in_channel -> string* qui lit une ligne de texte dans un fichier ouvert en lecture
- *close_in : in_channel -> unit* qui ferme un fichier texte ouvert à l'aide de *open_in*

Le module *Printf* quant à lui exporte des fonctions pour l'affichage de chaînes formatées.
Par exemple la fonction *Printf.printf* accepte des paramètres similaires à son homologue en *C* :

```
# Printf.printf "Delay: %04d\n" 89;;
Delay: 0089
- : unit = ()
```

# Partie VI. Application aux grands nombres

L'objectif est d'implémenter *Big_nat*, un module raisonnablement performant pour l'arithmétique sur les

entiers naturels.
Pour situer le niveau de performance, nous nous fixons comme objectif de fournir une multiplication et une division de complexité inférieure à celle d'une implémentation naïve. C'est-à-dire d'une complexité inférieure à la complexité quadratique.


39. Domaine des entiers naturels
40. Opérations sur les tableaux
41. L'addition
42. La comparaison
43. La soustraction
44. La multiplication longue
45. La multiplication *Knuth-Karatsuba*
46. La division *Burnikel-Ziegler*
47. Calcul de π au goutte-à-goutte


# Partie VII. Application à l'analyse syntaxique


48. Techniques d'analyse syntaxique
49. Les requêtes syntaxiques
50. Un analyseur pour *pascal*
51. Les erreurs de syntaxe
52. Implémentation de *Lex.lex_parser*


## 48. Techniques d'analyse syntaxique

On peut distinguer trois familles méthodologiques quand il s'agit de faire l'analyse d'un document texte structuré :

- les outils spécialisés *(Lex/Yacc/Bison*) utilisent un méta-langage pour spécifier la structure du texte, à partir de cette spécification ils génèrent une source qui réalise l'analyse du texte selon cette structure
- les *parser combinators* offrent une collection d'opérations élémentaires d'analyse ainsi que des opérateurs qui permettent de les combiner pour produire des analyseurs plus complexes
- la méthode traditionnelle, telle qu'enseignée par **Niklaus Wirth**, consiste à tout écrire à la main, sans dépendance ni à un outil ni à une bibliothèque

Le module que nous allons réaliser dans cette partie est une modernisation de la méthode traditionnelle.
Cette méthode suppose d'écrire à la main un analyseur syntaxique de type *descendant*.
Ce cours ne vous apprendra pas à écrire un *analyseur descendant*, si vous ne savez pas le faire veuillez vous référer à la littérature classique sur le sujet, par exemple le très classique Compiler Construction de **Niklaus Wirth** ou bien le cours de **Sébastien Doeraene**.

## 49. Les requêtes syntaxiques

Le type le plus important exporté par le module *Lex* est sans aucun doute le type *Lex.lex_parser*.
Hormis quelques considérations sur l'analyse syntaxique descendante en général, nous allons surtout nous attacher aux spécificités particulières à l'utilisation d'un objet *Lex.lex_parser*.
Comme vous allez le voir, il s'agit d'une révision de l'analyse syntaxique descendante à la **Niklaus Wirth**, dont le mérite principal est que la syntaxe y apparaît plus clairement grâce à une formulation plus ramassée.

Pour commencer il n'y a pas d'analyseur lexical au sens habituel.
Par exemple il n'y a pas de mots réservés mais seulement des mots clés contextuels.
On ne peut pas définir ses propres unités lexicales, elles sont toutes prédéfinies.

Un élément lexical appartient à l'une de ces 7 catégories :

- un *word* est un mot-clé introduit par le langage, par exemple **begin**

MCours.com

- un *name* est un identificateur introduit par le programmeur, par exemple **push**
- un *mono* est un charactère simple, par exemple **+**
- un *poly* est un charactère multiple, par exemple **>=**
- un *string* est une valeur chaîne de charactères immédiate, par exemple **"hello!"**
- un *char* est une valeur charactère immédiate, par exemple **'a'**
- un *int* est une valeur entière décimale immédiate non-signée, par exemple **31415**

Après les spécificités lexicales passons maintenant aux spécificités syntaxiques.

Classiquement, en plus de construire un arbre de syntaxe abstraite, un analyseur syntaxique descendant est responsable de 3 autres sortes d'opérations élémentaires :

- reconnaître une unité lexicale (souvent à l'aide d'une égalité)
- avancer d'une unité lexicale
- signaler une erreur syntaxique

Avec ces 4 tâches différentes il n'est pas étonnant que le code d'un analyseur descendant apparaîsse souvent comme surchargé.
Ce malheureux cumul des mandats est contraire à l'injonction dite de *separation of concerns* et encourage le recours à des outils spécialisés.
Au contraire de ces outils spécialisés, avec leur notation méta-linguistique, le module *Lex* vous maintient dans le confort du langage OCaml que vous connaissez déjà.

La conception du module *Lex* est basée sur deux critiques de la méthode traditionnelle :

- les opérations comme reconnaître une unité lexicale ou bien avancer d'une unité lexicale n'ont rien à faire dans un analyseur syntaxique, elles auraient mieux leur place dans un analyseur lexical
- les opérations *reconnaître-avancer-erreur* sont de trop bas niveaux, il faudrait pouvoir les fusionner dans des opérations de plus haut niveau

Pour remédier à ces deux déficiences le module *Lex* propose :

- de redescendre certaines opérations (comme l'égalité) au niveau de l'analyse lexicale
- la notion de requête qui combine *reconnaître-avancer-erreur* en une opération unique

Désormais, pour effectuer une analyse syntaxique descendante, il vous faudra penser en termes de requêtes *demand*, *granted* ou *promised* qui sont liées aux opérations classiques par l'explication suivante :

- si une requête *demand* reconnaît alors elle avance sinon elle signale une erreur
- si une requête *granted* reconnaît alors elle avance sinon elle n'avance pas mais ne signale pas d'erreur
- une requête promised *reconnaît* (ou pas); n'avance jamais; ne signale jamais d'erreur

Cette explication ne se justifie que par l'expérience de l'analyse descendante, elle n'est pas très intuitive en elle même.
C'est pourquoi, de façon similaire aux diagrammes syntaxiques, ce formalisme pour les grammaires **LL(1)** possède sa propre représentation graphique, les ordigrammes syntaxiques, qui va je l'espère vous aider à l'appréhender et à l'utiliser dans vos propres projets.

Les ordigrammes syntaxiques complètent harmonieusement les diagrammes syntaxiques :

- les diagrammes syntaxiques disent à quoi ressemble une phrase, leur sémantique est "déclarative"
- les ordigrammes syntaxiques disent comment reconnaître une phrase, leur sémantique est "impérative"

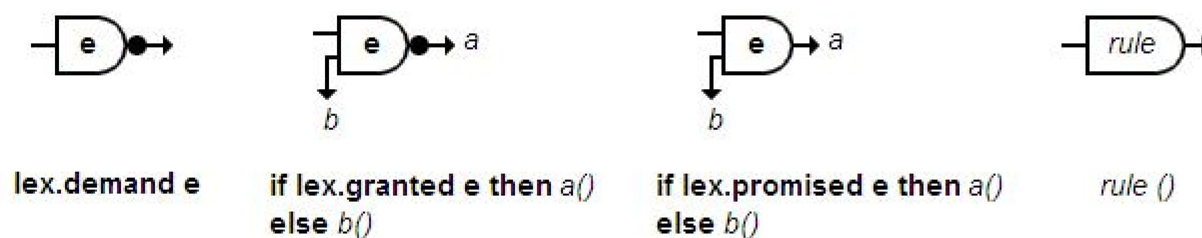
Pour les lire il suffira de savoir que :

- les ordigrammes syntaxiques sont composés d'écluses (ou portes) reliées par des canaux dans lesquels le flux lexical peut s'écouler
- le flux lexical entre dans une porte par la gauche
- si le flux lexical charrie le motif inscrit sur la porte alors il ressort par la droite, si ce faisant il traverse une puce noire alors l'élément inscrit sur la porte est retiré du flux lexical (la lecture dans le flux lexical avance d'une unité)
- sinon le flux lexical est interdit de passage et il poursuit sa course en bas à gauche de la porte
- sur une porte les unités lexicales sont marquées en **gras** tandis que les règles syntaxiques sont marquées en *italique*

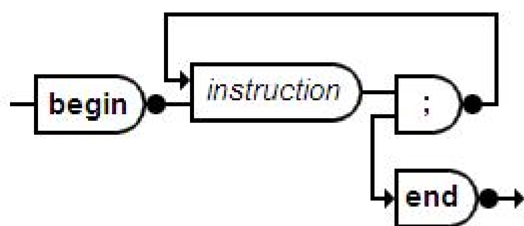
- une porte avec deux sorties, une 1ière à droite et la 2nd à gauche, ne peut pas être gardée par une *règle*, elle ne peut être gardée que par une unité lexicale

Il en résulte que la grammaire utilise la première unité du flux lexical comme le discriminant d'une phrase, on dit aussi que la grammaire est sous la forme **LL(1)**.

Les requêtes du module *Lex* sont reliées aux ordigrammes syntaxiques par une correspondance bi-univoque, chaque schéma possédant son équivalent textuel et vice-versa :




Par exemple l'ordigramme syntaxique d'une séquence :




Se traduit textuellement en *Caml* par la fonction *sequence* :

```
let sequence () =
  lex.demand_word "begin";
  instruction ();
  while lex.granted_mono ';' do
    instruction ()
  done;
  lex.demand_word "end"
```

## 50. Un analyseur pour *pascal*

Dans ce chapitre, afin de se donner une idée plus concrète des facilités et des limites du module *Lex*, on se propose d'implémenter un analyseur syntaxique descendant pour un langage *Turbo Pascal* modulaire à peine simplifié (sans extension *POO* cependant, nous sommes là pour apprendre l'analyse syntaxique, pas pour programmer en *Object Pascal*).

Pour le besoin du cours cette source *TP5.ml* est présentée ici en plusieurs morceaux entrecoupés de commentaires. Afin de faciliter vos expérimentations les sources *Lex.ml* et *TP5.ml* complètes ainsi que l'exécutable pour *Linux* et *Win32* ont été réunis dans cette archive.

Pour commencer :

- on ouvre le module *Lex* à l'aide de *open Lex*
- on créer un analyseur *lex* de type *Lex.lex_parser* qui lit le flux *stdin*, pour lire un fichier il suffira de rediriger l'entrée
- on créer *simple_name ()*, un analyseur élémentaire qui se contente d'oublier l'identificateur que l'on vient de demander à l'aide de *lex.demand_name ()*

```
open Lex;;

let lex = Lex.make stdin in

let rec simple_name () =
  ignore(lex.demand_not reserved)
```

Après ce **let rec** initial les définitions suivantes seront simplement introduites par **and**. Elles sont donc mutuellement récursives ce qui nous permettra d'utiliser par anticipation des règles de syntaxe pas encore introduites.

Un *simple_name* est un nom simple, c'est-à-dire non qualifié, comme par exemple dans **unit** *test*.
Un *simple_name* est un identificateur, il ne peut pas être un mot réservé.
Nous avons exprimé cette contrainte à l'aide de la requête syntaxique *demand_not reserved*.

*reserved* est le nom de notre prédicat qui dit si un mot est réservé ou non.
On peut réserver la liste complète des mots-clé du *pascal* en les plaçant dans un tableau soumis à un test d'occurence.
Le module *Lex* exporte justement une fonction *sorted_array_mem*, cette fonction réalise un test d'occurence sur un tableau, nous n'avons pas à chercher plus loin.

```
and reserved =
  Lex.sorted_array_mem
  [|
  "and";"array";"asm";
  "begin";"case";"const";
  "div";"do";"downto";"else";"end";
  "file";"for";"function";"goto";
  "if";"implementation";"in";"interface";
  "label";"mod";"nil";"not";"of";"or";"out";
  "packed";"procedure";"program";
  "record";"repeat";
  "set";"shl";"shr";"string";
  "then";"to";"type";
  "unit";"until";"uses";
  "var";"while";"with";"xor";
  |]
```

Précisons tout de même que *sorted_array_mem* fait une recherche *dichotomique* : elle est efficace mais elle n'est correcte que sur un tableau trié.
Si par misère notre tableau de mots-clé n'était pas déjà trié, nous pourrions le trier à l'aide de *Array.sort*.

Bien entendu nous pourrions coder une reconnaissance encore plus efficace des mots réservés, par exemple à l'aide d'un *arbre Patricia*.
Cette méthode n'aurait rien à envier aux meilleures technologies à base d'automates, toutefois pour avancer plus vite nous laissons ce raffinement à titre d'exercice à la discrétion du lecteur. Et nous progressons promptement dans l'épanchement de notre soif de syntaxe.

Un nom qualifié est un nom suivi par un suffixe comme par exemple *table* dans *table^[i].key* ou dans *table.at(i)*.
Remarquez que *key* est un nom et pourrait à son tour être qualifié.

Nous en savons assez pour pouvoir qualifier un nom :

```
and qualify_name () =
  ignore(lex.granted_mono '^');
  match lex with
  | l when lex.granted_mono '.' ->
        simple_name (); qualify_name ()
  | l when lex.granted_mono '[' ->
        expression ();
        lex.demand_mono ']';
        qualify_name ()
  | l -> arguments ()
```

Il nous reste alors à traiter le dernier cas, où un nom est qualifié par une liste d'arguments réels, comme *insert* dans *insert(key,item)*.

```
and arguments () =
  if lex.granted_mono '(' then begin
    expression ();
    while lex.granted_mono ',' do
      expression ()
    done;
    lex.demand_mono ')'; qualify_name ()
  end
```

Voilà qui nous amène aux expressions, cela peut paraître intimidant.
Cependant en y regardant de plus près nous avons déjà bien entamé la définition d'une expression.
Plus précisément il ne nous manque que les cas suivants :

- l'expression est une constante entière ou réelle (partie entière plus partie décimale)
- l'expression est une constante caractère
- l'expression est une constante chaîne de caractères
- l'expression est parenthésée
- l'expression est préfixé par un opérateur unaire (ou bien **-** ou bien **not**)
- l'expression est un nom éventuellement qualifié (c'est le cas déjà traité)
- l'expression est une opération (ou l'opérateur figure parmi **+ - * <= >= <> < > = and or xor div mod**)

Il n'y a plus qu'à traduire :

```
and expression () =
  ( match lex with
  | l when lex.promised_int ()     ->
        ignore (lex.demand_int ());
        if lex.granted_mono '.' then ignore (lex.demand_int ())
  | l when lex.promised_mono '\"' -> ignore (lex.demand_char ())
  | l when lex.promised_mono '\'' -> ignore (lex.demand_string ())
  | l when lex.granted_mono  '('  -> expression (); lex.demand_mono ')'
  | l when lex.granted_mono  '-'  -> expression ()
  | l when lex.granted_word "not" -> expression ()
  | l -> simple_name (); qualify_name ()
  ) ;
  ( match lex with
  | l when lex.granted_mono '+'   -> expression ()
  | l when lex.granted_mono '-'   -> expression ()
  | l when lex.granted_mono '*'   -> expression ()
  | l when lex.granted_poly "<="  -> expression ()
  | l when lex.granted_poly ">="  -> expression ()
  | l when lex.granted_poly "<>"  -> expression ()
  | l when lex.granted_mono '<'   -> expression ()
  | l when lex.granted_mono '>'   -> expression ()
  | l when lex.granted_mono '='   -> expression ()
  | l when lex.granted_word "and" -> expression ()
  | l when lex.granted_word "or"  -> expression ()
  | l when lex.granted_word "xor" -> expression ()
  | l when lex.granted_word "div" -> expression ()
  | l when lex.granted_word "mod" -> expression ()
  | l -> ()
  )
```

Nous en avons terminé avec les expressions, nous passons aux instructions.

La *séquence* permet de regrouper une série d'instructions séparées par un point-virgule :

```
and sequence () =
  lex.demand_word "begin";
  instruction ();
  while lex.granted_mono ';' do
```

```
        instruction ()
      done;
      lex.demand_word "end"
```

Finalement une instruction est :

- ou bien une séquence
- ou bien une instruction structurée (à choisir parmi **if while repeat for with**)
- ou bien un appel de procédure
- ou bien une assignation

Il n'y a plus qu'à traduire :

```
    and instruction () =
      match lex with
      | l when lex.promised_word "begin" ->
          sequence ();
      | l when lex.granted_word "if" ->
          expression ();
          lex.demand_word "then";
          instruction ();
          if lex.granted_word "else" then instruction ()
      | l when lex.granted_word "while" ->
          expression ();
          lex.demand_word "do";
          instruction ();
      | l when lex.granted_word "repeat" ->
          instruction ();
          while lex.granted_mono ';' do
            instruction ()
          done;
          lex.demand_word "until"; expression ()
      | l when lex.granted_word "for" ->
          simple_name (); qualify_name ();
          lex.demand_poly ":=";
          expression ();
          if lex.granted_word "to" then ()
          else lex.demand_word "downto";
          expression ();
          lex.demand_word "do";
          instruction ();
      | l when lex.granted_word "with" ->
          simple_name (); qualify_name ();
          lex.demand_word "do";
          instruction ();
      | l ->
          simple_name (); qualify_name ();
          if lex.granted_poly ":=" then expression ()
          else arguments()
```

Comme vous voyez ça avance très vite, nous en avons terminé avec les instructions.
Nous passons maintenant aux déclarations de types.

Une caractéristique de *pascal*, plutôt à l'abandon dans les langages modernes, ce sont ces types anonymes.
Bien sûr il y a les types prédéfinis comme **Integer**, **Longint**, **Real**, **Byte**, **Word**, **Char**, **Boolean**, **Pointer**.
Un programmeur *pascal* peut également définir un nouvel alias de type.
Mais en plus, sans avoir à le nommer, le programmeur *pascal* peut spécifier :

- un nouveau type chaîne **string***[max]*
- un nouveau type fichier **file of** *data_type*
- un nouveau type tableau **array** *[min..max]* **of** data_type
- un nouveau type enregistrement **record** *components* **end**
- un nouveau type **procedure** ou **function**
- un nouveau type pointeur (à l'aide du caractère ^)
- un nouveau type énuméré comme *(North, East, South, West)*

Il n'y a plus qu'à traduire :

```
and data_type () =
  match lex with
  | l when lex.granted_word "Integer" -> ()
  | l when lex.granted_word "Longint" -> ()
  | l when lex.granted_word "Real" -> ()
  | l when lex.granted_word "Byte" -> ()
  | l when lex.granted_word "Word" -> ()
  | l when lex.granted_word "Char" -> ()
  | l when lex.granted_word "Boolean" -> ()
  | l when lex.granted_word "Pointer" -> ()
  | l when lex.granted_word "string" ->
        if lex.granted_mono '[' then begin
          constant_value ();
          lex.demand_mono ']'
        end
  | l when lex.granted_word "file" ->
        lex.demand_word "of";
        data_type ()
  | l when lex.granted_word "array" ->
        lex.demand_mono '[';
        constant_value ();
        lex.demand_poly "..";
        constant_value ();
        lex.demand_mono ']';
        lex.demand_word "of";
        data_type ()
  | l when lex.granted_word "record" ->
        components ();
        lex.demand_word "end"
  | l when lex.granted_word "procedure" ->
        procedure_type ()
  | l when lex.granted_word "function" ->
        procedure_type (); lex.demand_mono ':'; data_type ()
  | l when lex.granted_mono '^' ->
        data_type ()
  | l when lex.granted_mono '(' ->
        simple_name ();
        while lex.granted_mono ',' do
          simple_name ()
        done;
        lex.demand_mono ')'
  | l -> simple_name ()
```

Sachant que *constant_value()*, *components ()* et *procedure_type ()* restent encore à définir.

Une valeur constante est soit un nom (défini à l'aide de **const**) soit une constante entière :

```
and constant_value () =
  if lex.promised_name () then simple_name ()
  else ignore (lex.demand_int ())
```

L'ensemble des composantes d'un enregistrement est une série de déclarations *noms*: *type* séparées par des virgules et points-virgules :

```
and components () =
  simple_name ();
  while lex.granted_mono ',' do
    simple_name ()
  done;
  lex.demand_mono ':';
  data_type ();
  if lex.granted_mono ';' then components ()
```

Un type procedure se résume à une (éventuelle) liste d'arguments formels *noms*: *type* séparés par des virgules et points-virgules. Ces arguments pourront être annotés **const**, **var**, **in**, **out** selon la façon dont on peut ou non les modifier.

```
and procedure_type () =
  if lex.granted_mono '(' then
  let rec loop () =
    ( match lex with
    | l when lex.granted_word "const" -> ()
    | l when lex.granted_word "var" -> ()
    | l when lex.granted_word "in" -> ()
    | l when lex.granted_word "out" -> ()
    | l -> ()
    ) ;
    simple_name ();
    while lex.granted_mono ',' do
      simple_name ()
    done;
    lex.demand_mono ':';
    data_type ();
    if lex.granted_mono ';' then loop ()
  in loop (); lex.demand_mono ')'
```

C'est fini pour les types.

Nous avons les expressions, nous avons les instructions, nous avons les types, il nous manque les deux derniers niveaux hiérarchiques supérieurs, à savoir :

- les déclarations (de constantes, de types, de variable globales, de procedures, de fonctions)
- les unités et les programmes

Une simple boucle nous donnera une vue d'ensemble des définitions :

- ou bien une déclaration débute par l'un des mots-clés **const type var procedure function**
- ou bien on quitte la boucle des déclarations

```
and declare () =
  comments ();
  match lex with
  | l when lex.granted_word "const" ->
      comments (); constant (); declare ()
  | l when lex.granted_word "type" ->
      comments (); definition (); declare ()
  | l when lex.granted_word "var" ->
      comments (); components (); declare ()
  | l when lex.granted_word "procedure" ->
      comments (); procedure (); declare ()
  | l when lex.granted_word "function" ->
      comments (); typed_procedure (); declare ()
  | l -> ()
```

Les déclarations sont souvent un endroit privilégié pour placer des commentaires.
Le module *Lex* ne gère pas tout seul les commentaires, les commentaires font donc partie de la grammaire.
Nous utiliserons les commentaires d'une ligne tels que *Delphi* les recommande :

```
and comments () =
  while lex.granted_poly "//" do
    lex.demand_line ()
  done
```

La déclaration de constantes réutilise *constant_value* :

```
and constant () =
  comments ();
  simple_name ();
  lex.demand_mono '=';
  constant_value ();
  if lex.granted_mono ';' then constant ()
```

La déclaration de types réutilise *data_type* :

```
and definition () =
  comments ();
  simple_name ();
  lex.demand_mono '=';
  data_type ();
  if lex.granted_mono ';' then definition ()
```

La déclaration de procédures réutilise *procedure_type* :

```
and procedure () =
  simple_name (); procedure_type ();
  lex.demand_mono ';';
  procedure_body ()
```

Une fonction est une *typed_procedure* :

```
and typed_procedure () =
  simple_name (); procedure_type ();
  lex.demand_mono ':'; data_type (); lex.demand_mono ';';
  procedure_body ()
```

Le corps d'une procédure est :

- ou bien une déclaration avancée **forward**
- ou bien une séquence **begin...end** éventuellement précédée d'une déclaration locale **var** (les *components* d'une procédure)

```
and procedure_body () =
  if not (lex.granted_word "forward") then begin
    if lex.granted_word "var" then components ();
    sequence ();
  end;
  lex.demand_mono ';'
```

Un programme débute par **program** puis un certain nombre de déclarations précèdent une séquence **begin...end**.

```
and program () =
  lex.demand_word "program";
  simple_name ();
  lex.demand_mono ';';
  comments ();
  declare ();
  sequence ()
```

Une unité débute par **unit**.
Puis suivent les sections **interface** et **implementation**.
Une unité est éventuellement clôturée par une séquence **begin...end**.

```
and modular () =
  lex.demand_word "unit";
```

```
      simple_name ();
      lex.demand_mono ';';
      comments ();
      lex.demand_word "interface";
      imports ();
      comments ();
      declare ();
      lex.demand_word "implementation";
      imports ();
      declare ();
      if lex.promised_word "begin" then sequence ()
      else lex.demand_word "end"
```

Enfin, une clause d'importation est un mot-clé **uses** suivi d'une liste de noms d'unités séparés par une virgule :

```
    and imports () =
      comments ();
      while lex.granted_word "uses" do
        comments ();
        simple_name ();
        while lex.granted_mono ',' do
          simple_name ()
        done;
        lex.demand_mono ';'
      done;
```

## 51. Les erreurs de syntaxe

Nous en arrivons au programme principal où il n'est plus question que de :

- discriminer entre **unit** et **program**
- demander le caractère '.' qui termine tout programme ou unité
- traiter les erreurs de syntaxe

```
    and turbo_pascal_5 () =
      try
        comments ();
        if lex.promised_word "unit" then modular ()
        else program ();
        if lex.promised_mono '.' then ()
        else lex.demand_mono '.'
      with
      | Lex.Demand_denied(position,demand) ->
          Lex.print_position position;
          Lex.print_denied   demand;
      | Lex.Word_reserved(position) ->
          Lex.print_position position;
          Lex.print_reserved ();
      | End_of_file ->
          print_endline "unexpected end of text"

    in turbo_pascal_5 ()
```

On voit qu'en fait il n'y a que trois erreurs possibles, chacune générant une exception :

- une requête *lex.demand_...* n'a pas pu être satisfaite et une exception *Lex.Demand_denied* est levée
- une requête *lex.demand_not reserved* n'a pas pu être satisfaite et une exception *Lex.Word_reserved* est levée
- le texte se termine prématurément et une exception *End_of_file* est levée

*Lex.print_position*, *Lex.print_denied* et *Lex.print_reserved* génèrent un rapport d'erreur qui permet à l'utilisateur de localiser l'unité lexicale fautive et de la situer dans son contexte en affichant :

- l'unité lexicale qui était attendue
- le numéro de la ligne fautive
- le texte de la ligne fautive (le passage incriminé est souligné)

Le traitement spécifique au point qui suit le **end** final mérite un petit commentaire.
Pour imposer la présence d'un point il suffit habituellement de la requête :

```
        lex.demand_mono '.';
```

Le problème c'est que si cette requête réussi alors elle va lire une unité lexicale supplémentaire.
Et comme c'est normalement la fin du texte une exception *End_of_file* va être générée.
Pour éviter ce désagrément l'idée est de tester la présence du point avec *lex.promised* et de ne faire appel à *lex.demand* qu'en cas d'erreur :

```
        if lex.promised_mono '.' then ()
        else lex.demand_mono '.'
```

C'est plus alambiqué qu'à l'habitude mais on obtient le comportement adéquat.

## 52. Implémentation de *Lex.lex_parser*

L'implémentation du module *Lex* est très représentative de la flexibilité de l'approche fonctionnelle.

À première vue, avec notre utilisation abondante de la notation pointée, on pourrait penser à quelque classe équipée d'un constructeur *make* dans le style classique de la programmation *orientée objet*.
Or il n'en est rien, nous n'avons utilisé que des constructions Caml déjà connues.

La notation pointée n'est due qu'à un type enregistrement dont les composantes sont de type fonctionnel :

```
    type lex_parser =
      {
      (* demands *)
      demand_line   : unit   -> unit;
      demand_word   : string -> unit;
      demand_mono   : char   -> unit;
      demand_poly   : string -> unit;
      demand_name   : unit   -> string;
      demand_not    : (string -> bool) -> string;
      demand_string : unit   -> string;
      demand_char   : unit   -> char;
      demand_int    : unit   -> int;
      (* promises *)
      promised_word : string -> bool;
      promised_mono : char   -> bool;
      promised_name : unit   -> bool;
      promised_int  : unit   -> bool;
      (* grants *)
      granted_word  : string -> bool;
      granted_mono  : char   -> bool;
      granted_poly  : string -> bool;
      }
```

Le constructeur *make* à son tour n'est qu'une fonction ordinaire qui :

- défini des variables locales
- défini des fonctions locales manipulant l'état local

- utilise ces fonctions locales pour construire un objet de type *Lex.lex_parser*

Le squelette de *make* ressemble à ceci (les passages omis ont été remplacés par ...) :

```
let make file =

  let
    (* local variables *)
    ...
  in

  (* demands *)
  let demand_line () = ...
  and demand_word s = ...
  and demand_mono c = ...
  and demand_poly s = ...
  and demand_name () = ...
  and demand_not reserved = ...
  and demand_string () = ...
  and demand_char () = ...
  and demand_int  () = ...
  (* promises *)
  and promised_word s = ...
  and promised_mono c = ...
  and promised_name () = ...
  and promised_int  () = ...
  (* grants *)
  and granted_word s = ...
  and granted_mono c = ...
  and granted_poly s = ...
  in

  demand_line ();

  (* build a lex_parser object *)
  {
  (* demands *)
  demand_line   = demand_line;
  demand_word   = demand_word;
  demand_mono   = demand_mono;
  demand_poly   = demand_poly;
  demand_name   = demand_name;
  demand_not    = demand_not;
  demand_string = demand_string;
  demand_char   = demand_char;
  demand_int    = demand_int;
  (* promises *)
  promised_word = promised_word;
  promised_mono = promised_mono;
  promised_name = promised_name;
  promised_int  = promised_int;
  (* grants *)
  granted_word  = granted_word;
  granted_mono  = granted_mono;
  granted_poly  = granted_poly;
  }
```

# Partie VIII. Les types algébriques

On appelle *type algébrique* un type qui réalise la *somme de produits* de types.
Ce vocabulaire sera défini en deux chapitres :

- le **chapitre 54** a pour objet de définir la nature d'un *type produit*
- le **chapitre 55** a pour objet de définir la nature d'un *type somme*

La connaissance des *types somme* et des *types produit* entraîne la connaissance des *types algébriques*.
Les *types inductifs* sont des *types algébriques récursifs* et feront l'objet du **chapitre 56**.

Enfin, les *schémas de récursion* les plus courants seront présentés dans les **chapitres 57 et 58**.


53. Ordre lexicographique
54. Les types produit
55. Les types somme
56. Les types inductifs
57. Les catamorphismes
58. Les paramorphismes


## 53. Ordre lexicographique

Implémenter un *ordre total* sur un type revient à implémenter une fonction similaire à la fonction *Pervasives.compare*.
Parfois on a une implémentation d'un *ordre total* sur un type et on veut l'étendre à une structure de donnée.
Par exemple on peut vouloir étendre cet ordre vers un ordre sur les listes ou sur les n-uplets.
Il existe une façon générale de réaliser cette extension, cette façon se nomme *ordre lexicographique* et fonctionne de façon similaire à l'ordre alphabétique.

Prenons exemple sur les caractères.
On peut bien sûr comparer deux caractères de type **char** :

```
# 'a' < 'o';;
- : bool = true
```

Et on peut aussi comparer deux chaînes de caractères de type **string** :

```
# "baba" < "bobo";;
- : bool = true
```

Cette façon d'étendre la comparaison du type **char** vers le type **string** c'est l'ordre alphabétique. On compare d'abord le $1^{ier}$ caractère, en cas d'égalité on compare le $2^{nd}$, puis éventuellement le $3^{ième}$, et ainsi de suite jusqu'à éventuellement atteindre la fin de la chaîne. C'est justement cette méthode qu'utilise la fonction *Pervasives.compare*. Illustration avec les listes :

```
# ['b';'a';'b';'a'] < ['b';'o';'b';'o'];;
- : bool = true
```

Puis avec les n-uplets :

```
# ('b','a','b','a') < ('b','o','b','o');;
- : bool = true
```

**Remarque:** dans le cas où l'on implémente une fonction *compare* spécifique (nous l'avons déjà fait par exemple pour les *inventaires*) la fonction qui suit nous renvoie l'*extension lexicographique* de cet ordre sur les listes.

```
# let lexicographical (cmp:'a -> 'a -> int) =
  let rec loop l1 l2 =
    match l1,l2 with
    | [],[] -> 0
    | [],_ -> -1
    | _,[] ->  1
    | a::t1,b::t2 ->
        let r = cmp a b in
        if r = 0 then loop t1 t2
        else r
```

```
    in loop;;
val lexicographical : ('a -> 'a -> int) -> 'a list -> 'a list -> int = <fun>
```

## 54. Les types produit

On appelle *type produit* un type qui réalise le *produit cartésien* de deux ou plusieurs types.

En OCaml il s'agit essentiellement :

- des *n-uplets* dont on a parlé dans le chapitre **18. Les couples**, le constructeur pour ces types est le caractère étoile, le constructeur pour ces valeurs est le caractère virgule
- des *enregistrements* dont on a parlé dans le chapitre **23. Les enregistrements**, les constructeurs pour ces types sont les noms d'un *type enregistrement*, le constructeur pour ces valeurs est la paire d'accolades

## 55. Les types somme

On appelle *type somme* un type qui réalise l'*union disjointe* de plusieurs types.
Il s'agit des *types énumérés* dont on a parlé au chapitre **13. Les types énumérés**, le constructeur pour ces types est le caractère barre-verticale, le constructeur pour ces valeurs est un *variant*.

À voix haute la barre-verticale se lit "ou bien".

Les *variants* ne sont pas limités aux *types énumérés* et nous allons progressivement exposer toute leur généralité.
Nous allons commencer par un simple problème de *jeu de cartes* qui suffira amplement à se convaincre de leur étonnante expressivité. Le problème que nous avons choisi est celui de la *main gagnante* au *poker*, après l'*abattage* (*showdown*) : laquelle parmi deux *mains* l'emporte sur l'autre ?

Le néophyte pourra rapidement s'initier au poker, il retiendra en particulier qu'une *main* au poker est constituée de 5 cartes et que le jeu consiste à produire la meilleure *combinaison* possible :

- avoir en main une *carte haute* (*HighCard*) c'est passable
- avoir en main une *paire* (*OnePair*) c'est un peu mieux
- avoir en main *deux paires* (*TwoPair*) c'est déjà pas mal
- avoir en main un *brelan* (*ThreeOfAKind*) c'est meilleur
- avoir en main une *quinte* (*Straight*) c'est déjà confortable
- avoir en main une *couleur* (*Flush*) c'est encore plus confortable
- avoir en main une *main pleine* (*FullHouse*) c'est bien plus avantageux
- avoir en main un *carré* (*FourOfAKind*) c'est encore plus avantageux
- avoir en main une *quinte-flush* (*StraightFlush*) c'est l'idéal

On résume ainsi la relation d'*ordre total* pour une *combinaison* :

- *HighCard* < *OnePair* < *TwoPair* < *ThreeOfAKind* < *Straight* < *Flush* < *FullHouse* < *FourOfAKind* < *StraightFlush*

Ainsi bien sûr que la relation d'*ordre total* pour une *carte seule* :

- *2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < jack < queen < king < ace*

Pour implémenter l'*ordre total* sur *une seule carte* il suffit de désigner les cartes numériques par leur valeur entière et les *figures* par cette définition multiple (à l'aide de l'appariement de motif sur un *quadruplet*) :

```
# let ace,king,queen,jack = 50,40,30,20;;
val ace : int = 50
val king : int = 40
val queen : int = 30
```

```
val jack : int = 20
```

La tentation est grande d'appliquer la même technique (appariement de motif sur un *n-uplet*) pour implémenter l'ordre total sur une *combinaison* :

```
let
  highcard,onepair,twopair,threeofakind,straight,
  flush,fullhouse,fourofakind,straightflush
  =
  1,2,3,4,5,6,7,8,9
  ;;
```

Et cela nous permet bien la comparaison de deux *combinaisons*. Cependant l'exemple des *jours de la semaine* dans le déjà lointain chapitre **14. Le filtrage** et la discussion qui en découlait nous incitent à penser qu'un *type énuméré* ferait mieux l'affaire. Nous préfèrerons donc déclarer le type *poker_hand* :

```
type poker_hand =
  | HighCard
  | OnePair
  | TwoPair
  | ThreeOfAKind
  | Straight
  | Flush
  | FullHouse
  | FourOfAKind
  | StraightFlush
  ;;
```

D'un côté, en éliminant les valeurs numériques arbitraires, nous gagnons en clarté, de plus le type *poker_hand* est désormais candidat au *filtrage*, une facilité si essentielle que l'habitude et le confort nous l'ont déjà rendue indispensable. De l'autre côté nous ne perdons absolument rien car, les fonctions standards de comparaisons étant *polymorphes*, les *types énumérés* sont aussi bien ordonnés que les **int**. Pour l'exemple :

```
# ThreeOfAKind < FullHouse;;
- : bool = true
# compare Straight StraightFlush;;
- : int = -1
```

Toutefois il subsiste encore deux questions sans réponse :

- que décider en cas de *combinaisons* de force égale, par exemple comment comparer un *brelan de dames* avec un *brelan de rois* ?
- comment comparer deux *mains* opposant la même combinaison, par exemple une *paire de dames* contre une *paire de dames* ?

À ces deux problèmes OCaml apporte une seule et même solution :

- ou bien un *variant* n'attend aucun paramètre
- ou bien un *variant* attend exactement 1 paramètre

Le paramètre d'un *variant* peut être de n'importe quel type et en particulier un *type produit*.
Un *type énuméré* est un *type somme* dégénéré, aucun de ses *variants* n'admet un paramètre.

Pour spécifier le paramètre d'un *variant* on utilise le mot-clé **of** suivi du type du paramètre.
Par exemple il est tout à fait judicieux de déclarer notre type *poker_hand* comme ceci :

```
type poker_hand =
  | HighCard of int * int * int * int * int
  | OnePair of int * int * int * int
  | TwoPair of int * int * int
  | ThreeOfAKind of int
  | Straight of int
  | Flush of int * int * int * int * int
  | FullHouse of int * int
```

```
        | FourOfAKind of int
        | StraightFlush of int
        ;;
```

Le type *poker_hand* réalise l'*union disjointe* de *n-uplets*, c'est donc un type *somme de produits* ou *type algébrique* (non récursif).

À *variant* égal la comparaison de deux valeurs de type *poker_hand* se fait selon l'ordre lexicographique sur les *n-uplets*.
Par exemple cet extrait de session compare dans l'ordre, deux *cartes hautes*, deux *paires de dames*, deux *mains pleines*, et pour fournir une *quinte-flush au 8* face à une *quinte-flush au 6* :

```
HighCard(king,queen,jack,10,7) > HighCard(king,queen,jack,10,5);;
- : bool = true
OnePair(queen,king,jack,10) > OnePair(queen,king,jack,7);;
- : bool = true
FullHouse(queen,jack) > FullHouse(queen,10);;
- : bool = true
StraightFlush(8) > StraightFlush(6);;
- : bool = true
```

## 56. Les types inductifs

On appelle *type inductif* un type algébrique *récursif*.

La *récursivité* permet la création de types *composites*, avec un *type inductif* on peut créer des *composants*, c'est-à-dire des valeur-objets *enfichables* qui sont toujours du même type et qui sont traités de façon uniforme quelle que soit leur complexité.

Premier exemple :

Une liste de type *'a list* est :

- ou bien la liste vide *Nil*
- ou bien une paire *Cons* formée d'une tête de type *'a* et d'une queue de type *'a list*

Le type *'a list* est donc un *type inductif* que l'on pourrait déclarer ainsi :

```
# type 'a list =
    | Nil
    | Cons of 'a * ('a list)
    ;;
type 'a list = Nil | Cons of 'a * 'a list
```

La *liste 1::2::3::[]* s'écrirait alors *Cons(1,Cons(2,Cons(3,Nil)))* et l'opérateur de concaténation s'implémenterait ainsi :

```
# let rec append l1 l2 =
  match l1 with
    | Nil -> l2
    | Cons(h,t) -> Cons(h,append t l2)
    ;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

Même si le type *'a list* n'était pas prédéfini en OCaml nous pourrions encore l'implémenter simplement grâce à un *type inductif*.

**Remarque:** la définition du type *list* ci-dessus a écrasé la définition du type *list* prédéfini, celui-ci n'est plus disponible pour l'utilisateur, il vous faut redémarrer l'interpréteur pour avoir à nouveau accès au type *list* prédéfini que bien sûr nous allons continuer d'utiliser dans la suite de ce cours.

Autre exemple :

Au **chapitre 33** nous n'avons pas réussi à implémenter les *arbres n-aires* à l'aide d'un couple (c'est-à-dire à l'aide d'un simple *type produit*), nous allons voir que ce qui faisait obstacle c'étaient la notion de *type inductif* qui nous manquait encore à ce moment. .

Un *arbre n-aire* est :

- un noeud (*node*) contenant un élément et une liste de sous-*arbres n-aires* (*récursivement*)

La déclaration de ce type suit immédiatement :

```
# type 'a tree =
    Node of 'a * ('a tree list);;
type 'a tree = Node of 'a * 'a tree list
```

Pas de surprise, nos fonctions sur ces *arbres n-aires* sont tout à fait analogues à leurs cousines du **chapitre 33** qui utilisaient un *type enregistrement* :

```
let rec rev_tree (Node(a,l)) =
  Node(a,List.rev_map rev_tree l);;

let rec map_tree f (Node(a,l)) =
  Node(f a,List.map (map_tree f) l);;

let rec exists_tree p (Node(a,l)) =
  p a or List.exists (exists_tree p) l;;
```

Une dernière utilisation des *types inductifs* est d'une importance particulière, ce sont les *arbres de syntaxe abstraite* qui modélisent les expressions, par exemple les expressions arithmétiques, ici avec les 4 opérateurs :

```
type arithmetic =
  | Int of int
  | Add of arithmetic * arithmetic
  | Sub of arithmetic * arithmetic
  | Mul of arithmetic * arithmetic
  | Div of arithmetic * arithmetic
  ;;
```

Ces *types inductifs* sont généralement accompagnés d'une fonction d'évaluation, soit *eval* pour notre exemple :

```
let rec eval expr =
  match expr with
  | Int n -> n
  | Add(a,b) -> eval a + eval b
  | Sub(a,b) -> eval a - eval b
  | Mul(a,b) -> eval a * eval b
  | Div(a,b) -> eval a / eval b
  ;;
```

## 57. Les catamorphismes

On appelle *catamorphisme* une fonction qui réalise un *parcours canonique* sur un *type inductif*.

Le *type inductif* le plus élémentaire est le type *list*, le module *List* fournit deux sortes de parcours sur les listes :

- La fonction *fold_right* effectue un parcours récursif, on pourrait l'implémenter comme ceci :

```
let rec fold_right f l init =
  match l with
  | [] -> init
  | h::t -> f h (fold_right f t init)
  ;;
```

- La fonction *fold_left* effectue un parcours récursif terminal, on pourrait l'implémenter comme ceci :

```
let rec fold_left f init l =
  match l with
  | [] -> init
  | h::t -> fold_left f (f init h) t
  ;;
```

Plutôt que ces implémentations un peu ardues, essayons de donner une définition d'un *catamorphisme de liste* :

- l'image de *[]* par un catamorphisme est la valeur initiale *init*
- la fonction *f* attend deux arguments
- soit *$$* un opérateur infixe égal à la fonction *f* et que l'on pourrait déclarer ainsi :

```
let ($$) = f;;
```

- alors l'image d'une liste $a_0::a_1::\ ...\ ::a_n::[]$ par un catamorphisme est la valeur $a_0\$\$a_1\$\$\ ...\ \$\$a_n\$\$init$

En résumé :

- le constructeur inital *[]* est remplacé par la valeur initiale *init*
- chaque constructeur *::* est remplacé par l'opérateur *$$*

Le module *Seq* qui suit illustre la généralité des catamorphismes *fold_left* et *fold_right*, cette implémentation d'une partie de la *signature* du module *List* est surprenante de concision :

```
module Seq = struct
  let rec fold_left f init l =
    match l with
    | [] -> init
    | h::t -> fold_left f (f init h) t
  let rec fold_right f l init =
    match l with
    | [] -> init
    | h::t -> f h (fold_right f t init)
  let length l = fold_left (fun a b -> a+1) 0 l
  let rev l = fold_left (fun a b -> b::a) [] l
  let append l1 l2 = fold_right (fun a b -> a::b) l1 l2
  let rev_append l1 l2 = fold_left (fun a b -> b::a) l2 l1
  let iter f l = fold_left (fun a b -> f b) () l
  let map f l = fold_right (fun a b -> f a::b) l []
  let rev_map f l = fold_left (fun a b -> f b::a) [] l
  let for_all f l = fold_left (fun a b -> f b && a) true l
  let exists f l = fold_left (fun a b -> f b or a) false l
  let mem x l = fold_left (fun a b -> x=b or a) false l
  let filter f l = fold_right (fun a b -> if f a then a::b else b) l []
  let partition f l = fold_right (fun a (p,q) -> if f a then (a::p,q) else (p,a::q)) l ([],[])
  let split l = fold_right (fun (a,b) (p,q) -> (a::p,b::q)) l ([],[])
end;;
```

Le cas des *arbres binaires* est également intéressant.

Un *arbre binaire* est :

- ou bien une feuille (*leaf*) sans élément

- ou bien un noeud (*node*) contenant un élément et deux sous-*arbres binaires* (récursivement)

Le module *BinaryTree* qui suit introduit un type *tree* et son catamorphisme *fold*, ensuite il utilise *fold* pour implémenter la taille (*size*), la profondeur (*depth*), le nombre de Strahler (*strahler*), puis d'autres fonctions de parcours de même sémantique que leurs homonymes sur les listes :

```
module BinaryTree = struct
  type 'a tree =
    | Leaf
    | Node of ('a tree) * 'a * ('a tree)
  let fold f init t =
    let rec loop t =
      match t with
      | Leaf -> init
      | Node(l,a,r) -> f (loop l) a (loop r)
    in loop t
  let size t  = fold (fun l a r -> l + r + 1) 0 t
  let depth t = fold (fun l a r -> max l r + 1) 0 t
  let strahler t = fold (fun l a r -> if l=r then l+1 else max l r) 1 t
  let rev t = fold (fun l a r -> Node(r,a,l)) Leaf t
  let iter f t = fold (fun l a r -> f a) () t
  let map  f t = fold (fun l a r -> Node(l,f a,r)) Leaf t
  let for_all f t = fold (fun l a r -> l && r && f a) true t
  let exists  f t = fold (fun l a r -> l or r or f a) false t
  let mem x t = fold (fun l a r -> l or r or x=a) false t
  let split t = fold (fun (m,q) (a,b) (p,r) -> Node(m,a,p),Node(q,b,r)) (Leaf,Leaf) t
end;;
```

Nous vous renvoyons à la thèse de Nicolas Janey pour une définition des nombres de *Horton-Strahler* (lire les paragraphes **3.2.1** à **3.2.3**) et pour leur application en botanique, en hydrogéologie et en topologie.

Le module *BinaryTree* se prête volontiers à une généralisation aux *arbres n-aires* tels que nous les avons définis au **chapitre 57**.
Ce sera le module *NaryTree* qui, après la définition du type *tree* et l'introduction des ses deux catamorphismes canoniques *fold_left* et *fold_right*, implémente les opérations sur les *arbres n-aires*, toujours avec une extrême concision :

```
module NaryTree = struct
  type 'a tree =
    Node of 'a * ('a tree list)
  let fold_left f g init t =
    let rec helper (Node(a,l)) =
      let loop = List.fold_left (fun a b -> g (helper b) a) init l
      in  f a loop
    in helper t
  let fold_right f g init t =
    let rec helper (Node(a,l)) =
      let loop = List.fold_right (fun a b -> g (helper a) b) l init
      in  f a loop
    in helper t
  let cons a b = a::b
  let size t = fold_left (fun a l -> l + 1) (+)  t
  let depth t = fold_left (fun a l -> l + 1) max 1 t
  let rev t = fold_left (fun a l -> Node(a,l)) cons [] t
  let iter f t = fold_left (fun a l -> f a) (fun a b -> ()) () t
  let map f t = fold_right (fun a l -> Node(f a,l)) cons [] t
  let rev_map f t = fold_left (fun a l -> Node(f a,l)) cons [] t
  let for_all f t = fold_left (fun a l -> l && f a) (&&) true t
  let exists  f t = fold_left (fun a l -> l or f a) (||) false t
  let mem x t = fold_left (fun a l -> l or x=a) (||) false t
end;;
```

À ce stade on peut se demander à quoi servent les *catamorphismes*.
Après tout on a déjà le *filtrage* qui est une facilité très puissante.
Et les modules exportent d'autres facilitées plus immédiates comme *rev*, *map* ou *filter*.
Alors on peut penser que les catamorphismes c'est concis, le concept est élégant, mais en pratique ce qui compte c'est l'inferface d'un module et non la concision de son implémentation.

Sans vouloir minimiser le bien fondé de ces deux remarques il faut tout de même dire que le programmeur fonctionnel expérimenté ne peut pas complètement faire l'économie des catamorphismes.

Pour deux raisons principales :

- En tant qu'utilisateur d'un module le programmeur fonctionnel peut être confronté à une situation où son besoin n'est pas couvert par l'interface du module (les fonctions classiques *rev*, *map* ou *filter*...). Dans ce cas il ne peut pas non plus coder son propre parcours à l'aide du filtrage car l'implémentation du type à parcourir est cachée par le module, on ne connaît pas son schéma, on ne peut pas filtrer ses valeurs. Il ne reste alors plus qu'une option disponible: paramétrer le *fold* dont le bon implémenteur de module aura eu soin d'équiper son interface.
- En tant qu'implémenteur de composants certifés ou critiques, paramétrer un *fold* est une option plus sûre, car on ne peut pas faire une récursion mal fondée. Un catamorphisme est *structurellement bien fondé*, l'utiliser c'est déjà une preuve de terminaison mais surtout c'est une base pour le *raisonnement inductif* et un premier pas important vers la preuve de programme.

Comme nous avons déjà vu beaucoup d'exemples de modules, jusqu'à la fin de ce chapitre nous allons nous placer dans le cadre plus confidentiel de la preuve de programme. Aussi, si vous n'êtes pas intéressé par la preuve de programme, vous pouvez sans conséquence aucune faire l'impasse sur cette fin de **partie VIII**.

Sinon vous allez voir comment remplacer la récursion par l'utilisation systématique de schémas de récursion afin de faciliter le raisonnement inductif.
Pour l'exemple nous allons nous concentrer sur les arbres binaires et le type :

```
type 'a tree =
  | Leaf
  | Node of ('a tree) * 'a * ('a tree)
```

Avec l'intention de lui adjoindre des opérations classiques comme *member, insert* et *remove*, autant dire que ce que nous voulons spécifier ce n'est plus un simple arbre binaire mais bien un arbre binaire de recherche (*binary search tree*).

Un arbre binaire de recherche est un arbre binaire qui vérifie les deux propriétés suivantes :

- l'élément porté par un fils gauche est toujours strictement plus petit que l'élément porté par son parent
- l'élément porté par un fils droit est toujours strictement plus grand que l'élément porté par son parent

Comme nous l'avons déjà mentionné à maintes reprise le système de types d'OCaml n'est que structurel, il n'est pas capable de distinguer un arbre binaire ordonné (par la propritété ci-dessus) d'un simple arbre binaire quelconque. Si bien que nous allons recourir au pis-aller habituel: un prédicat *ordered* nous dit si un arbre binaire est ordonné et ce prédicat nous sert de précondition pour les fonctions *member, insert* et *remove*.

```
let fold init f t =
  let rec loop t =
    match t with
    | Leaf -> init
    | Node(l,a,r) -> f (loop l) a (loop r)
  in loop t
```

Ce *fold* devrait faire l'affaire, il devrait suffire de le paramétrer pour obtenir le prédicat *ordered* que nous voulons.
Malheureusement ce que l'on cherche c'est à encadrer chaque arbre par un *minorant* et un *majorant*, or *Leaf* ne porte aucune valeur, dans ces conditions il n'est pas facile de l'encadrer, et pas facile non plus d'encadrer son parent.

Le prédicat *ordered* serait plus évident à implémenter si nous avions toujours accès à au moins une valeur initiale, il serait plus confortable que chaque feuille contienne au moins un élément.
Après tout OCaml exige bien une valeur initiale pour créer une référence ou pour créer un tableau alors pourquoi pas pour quelque chose de plus compliqué comme un arbre binaire de recherche ?
Sans plus d'états d'âme, nous interdisons carrément qu'un arbre binaire soit vide :

```
type 'a tree =
  | One  of 'a
  | Pair of 'a * 'a
  | Node of ('a tree) * 'a * ('a tree)
```

Puis nous équipons cet arbre binaire de son catamorphisme canonique :

```
let cata_rec one pair node t =
  let rec loop t =
    match t with
    | One a -> one a
    | Pair(a,b) -> pair a b
    | Node(l,a,r) -> node (loop l) a (loop r)
  in loop t
```

Est-ce qu'au moins cela nous facilite l'implémentation du prédicat *ordered* ? Oui, car il est désormais facile d'encadrer n'importe quel arbre par un intervalle (*minorant*, *majorant*) :

```
let ordered t =
  let a,b =
    cata_rec
    (fun a -> a,a)
    (fun a b -> a,b)
    (fun (l1,l2) a (r1,r2) ->
       if l1>l2 then l1,l2
       else if l2>a then l2,a
       else if a>r1 then a,r1
       else if r1>r2 then r1,r2
       else l1,r2
    )
    t
  in a<=b
```

Muni de ce prédicat et de notre catamorphisme canonique rien n'est plus simple que d'implémenter *member* c'est-à-dire une fonction de recherche d'un élément *x* dans *t* (un arbre binaire ordonné) :

```
let member x t =
  assert(ordered t);
  cata_rec
  (fun a -> x=a)
  (fun a b -> x=a or x=b)
  (fun l a r -> l or x=a or r)
  t
```

Mais attendez voir... c'est exactement la même fonction que *mem*, la recherche dans un arbre binaire quelconque! Où est l'intérêt d'un arbre binaire de recherche s'il n'accélère pas la recherche ?
Nous voudrions pouvoir rechercher un élément dans un temps proportionnel à la profondeur de l'arbre, et pas dans un temps proportionnel à sa taille.

Comment empêcher notre catamorphisme de parcourir l'arbre en totalité ?
Ce qu'il nous faudrait c'est un parcours moins avide, un parcours qui se laisse diriger. Ce parcours plus **lazy** nous l'appèlerons *caty_rec*, il est le frère de *cata_rec*, le même en plus paresseux :

```
let caty_rec one pair node t =
  let rec loop t () =
    match t with
    | One a -> one a
    | Pair(a,b) -> pair a b
    | Node(l,a,r) -> node (loop l) a (loop r)
  in loop t ()
```

Et il nous permet d'effectuer notre recherche en temps logarithmique :

```
let member x t =
  assert(ordered t);
  caty_rec
  (fun a -> x=a)
  (fun a b -> x=a or x=b)
  (fun l a r ->
     if x < a then l()
     else if x > a then r()
     else true)
  t
```

## 58. Les paramorphismes

Autant le dire tout de suite: le plus difficile est encore devant nous car il reste les opérations d'insertion (*insert*) et de suppression (*remove*).

Comment aborder l'insertion ?
Avec les mêmes exigences que pour l'appartenance: nous ne voulons pas reconstruire tout l'arbre mais seulement le chemin qui mène à la nouvelle feuille.
Or, pour ne reconstruire que ce chemin nous avons besoin de ses effluents, c'est-à-dire des anciennes composantes de chaque noeud que nous voulons remplacer.
Comme diraient les anglophones nous voulons 'eat our cake and have it too', et cela un catamorphisme ne nous le permet pas.

En résumer nous voulons à la fois :

- manger un noeud
- l'avoir encore sous les yeux et dans la main
- rester paresseux (ne pas parcourir tout l'arbre)

C'est ce petit miracle (ou plutôt ce schéma de récursion) que nous appelerons *paramorphisme*.
Pour un arbre binaire il prendra cette forme :

```
let para_rec one pair node t =
  let rec loop t () =
    match t with
    | One a -> one a
    | Pair(a,b) -> pair a b
    | Node(l,a,r) -> node (l,loop l) a (r,loop r)
  in loop t ()
```

Avec cette bagette magique l'insertion d'un élément *x* ne pose plus vraiment problème :

```
let insert x t =
  assert(not(member x t));
  para_rec
  (fun a -> if x > a then Pair(a,x) else Pair(x,a))
  (fun a b ->
     if x > b then Node(One a,b,One x)
     else if x < a then Node(One x,a,One b)
```

```
         else Node(One a,x,One b)
       )
       (fun (l1,l2) a (r1,r2) ->
          if x < a then Node(l2(),a,r1)
          else Node(l1,a,r2())
       )
       t
```

D'ailleurs notre baguette est tellement magique, utilisons la vite avant qu'on nous la confisque!
Vite, nous enlevons le minorant d'un arbre binaire de recherche :

```
let remove_min t =
  assert(ordered t);
  para_rec
  (fun a -> One a,a)
  (fun a b -> One b,a)
  (fun (l1,l2) a (r1,r2) ->
     match l1 with
     | One b -> r1,b
     | _ -> let b,c = l2() in Node(b,a,r1),c
  )
  t
```

Grâce à quoi nous pouvons finalement retirer un élément *x* quelconque :

```
let remove x t =
  assert(member x t);
  para_rec
  (fun a -> One a)
  (fun a b -> if x > a then One a else One b)
  (fun (l1,l2) a (r1,r2) ->
     if x < a then Node(l2(),a,r1)
     else if x > a then Node(l1,a,r2())
     else
       match l1,r1 with
       | One b,One c -> Pair(b,c)
       | One _,Node(_,_,_) -> r1
       | Node(_,_,_),One _ -> l1
       | Pair(b,c),One d | One b,Pair(c,d) ->
           Node(One b,c,One d)
       | _,_ ->
           let b,c = remove_min r1 in Node(l1,c,b)
  )
  t
```

Et voilà!
Nous avons implémenté un *Type Abstrait de Données* récursif sans utiliser directement aucune récursion.

Si vous souhaitez approfondir le sujet et faire connaissance avec de nouveaux schémas de récursion (*anamorphismes*, *hylomorphismes*, *apomorphismes*) je ne peux que vous recommander la lecture de Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire par **Fokkinga**.

Copyright © 2007 Damien Guichard. Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 3.0 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-nc-nd/3.0/legalcode

MCours.com