I11 : Informatique L1 1^{er} semestre
Gilles Bernot

Cours d'Informatique
Notes de cours

année 2006-2007
Université d'Évry

Avertissement au lecteur :

Ce polycopié n'est pas un document scolaire de référence sur le cours d'informatique, c'est seulement l'ensemble de mes propres notes de cours mises en forme. Il ne contient donc pas les explications détaillées qui ont été données en cours. En particulier tous les développements systématiques des exemples, expliquant comment le langage ML effectuerait les traitements, sont absents de ces notes. On trouvera également parfois quelques simples allusions à des concepts élémentaires, largement développés en cours mais qui sont routiniers pour tout informaticien.

G. Bernot

COURS 1

1. Informatique, Gigantisme, Méthodes
2. Buts du cours
3. Les langages de programmation
4. Classement des langages
5. Bibliographie
6. Structures de données, types de données, types, fonctions
7. Type de données des booléens
8. Type de données des entiers relatifs
9. Type de données des nombres « à virgule flottante »
10. Type de données des caractères
11. Type de données des chaînes de caractères
12. Les expressions bien ou mal typées

COURS 2

1. Enrichir le pouvoir d'expression : les déclarations
2. La portée d'une déclaration globale
3. Déclarations de fonctions
4. Les types produits cartésiens
5. Un argument produit cartésien v.s. plusieurs arguments
6. Expressions conditionnelles
7. Déclarations locales
8. Déclarations multiples

COURS 3

1. Typage des fonctions
2. Typage des fonctions avec déclarations locales
3. Fonctions partielles, traitement d'exceptions
4. Les types « enregistrement »
5. Champs explicites pour les enregistrements

COURS 4

1. Résolution de problèmes par découpages
2. Programmation structurée
3. Un exemple : la date du lendemain

4. Graphe d'appels des fonctions
5. Graphes d'appels avec récursivité mutuelle
6. La différence entre `let` et `let rec`
7. Cas des cycles minimaux

COURS 5

1. Exemples de fonctions récursives
2. Les types « liste »
3. Premières fonctions sur les listes
4. Premières fonctions récursives sur les listes

COURS 6

1. L'usage de `match` sur les listes
2. Seuls `[]` et `_ : :_` sont indispensables
3. Exemples de fonctions simples sur les listes

COURS 7

1. Sous-ensembles finis d'un type quelconque

COURS 8

1. Sous-ensembles finis d'un type ordonné

COURS 9

1. Preuves de terminaison de fonctions récursives

Cours No. 1

1 Informatique, Gigantisme, Méthode

Les cours d'informatique porteront sur l'apprentissage des bonnes méthodes de programmation, au moyen d'un langage de programmation spécialement conçu pour la pédagogie : ML. L'objectif est d'acquérir les « bonnes habitudes »qui permettront ultérieurement d'écrire des logiciels de grande taille et facilement modifiables.

Ceux qui n'ont jamais programmé ont presque une meilleure idée de l'informatique grandeur nature que ceux qui ont déjà écrit des programmes chez eux (impression fausse de mise au point petit à petit toujours possible, etc.). Donc ceux qui partent avec un handicap pour ce cours ne sont pas ceux qu'on croit.

L'informatique individuelle donne souvent de mauvaises habitudes, dues à la faible taille des programmes écrits (souvent 1 seul programmeur, donc organisation facile), aux faibles conséquences des erreurs (vies humaines, investissements perdus, expérience scientifique unique manquée...), et au peu d'exigence des utilisateurs.

Points clefs du génie logiciel : plusieurs développeurs, utilisateurs≠développeurs, gravité des erreurs.

La programmation « ludique » est dangereuse pour des projets en grandeur nature. Conséquences terribles sur centrales nucléaires, débit des vannes d'un barrage, gestion des ambulances, etc. Un oubli ou une erreur **n'**est **pas** réparable comme dans un « petit » programme (petit ≈ écrit par un seul homme en moins de quelques mois/années).

Gigantisme de l'informatique $\Longrightarrow$ méthodes, découpage des systèmes à réaliser, approche systématique, rigueur formelle (supérieure à une démonstration mathématique car tous les détails doivent être explicités dans un langage strictement symbolique, qui n'autorise que certaines formes bien délimitées au point d'être analysable automatiquement par ordinateur).

2 Buts du cours

Acquérir les premiers éléments de cette rigueur sur des problèmes de petite taille. Appréhender les premières notions de « programmation dans les règles de l'art ». Comprendre les méthodes de découpage de problèmes en sous-problèmes et comment combiner les solutions intermédiaires. Ne plus avoir ni une idée mythique de l'informatique, ni réductrice, en sachant quelles sont les techniques mises en jeu, leur rôle, leurs interactions.

Comprendre ce qu'est un langage de programmation, l'évolution des langages de programmation. Savoir utiliser les structures classiques de la programmation pour concevoir des programmes justes par construction.

Vecteur d'apprentissage = un langage de programmation spécialisé dans la mise en œuvre rigoureuse des méthodes de programmation : ML. Signifie Méta Langage [Historique du nom : premier langage pour programmer des stratégies automatiques de preuves de théorèmes]. Il généralise et impose par construction une approche que l'on doit suivre dans *tous* les **langages** de programmation pour réaliser un logiciel proprement.

3 Les langages de programmation

Un langage de programmation est un « vocabulaire » restreint et des règles de formation de « phrases » très strictes pour donner des instructions à un ordinateur. Le *moins* par rapport au français ou aux mathématiques est donc sa pauvreté. Le *plus* est qu'aucune « phrase »(=*expression*) n'est ambigüe : il n'existe pas 2 interprétations possibles. De plus, chaque phrase(=expression) est **exécutable**. Exécutable signifie qu'un programme part d'informations entrées sous forme de caractères/symboles et/ou de nombres, qu'il décrit des séquences de transformations de symboles et/ou de calculs sur ces entrées, pour aboutir à un résultat également symbolique. Exemple :

```
# let distance (x,y) = if x < y
                       then y-x
                       else x-y ;;
# distance (48,31) ;;
-: int = 17
#
```

`let` veut dire *Soit* comme en maths. Calculs sur 48 et 31, transformation des symboles `x` et `y` en des nombres, transformation des caractères « `distance` » en une expression qui est ensuite exécutée. [+ suivre les étapes de réduction de l'expression `if..then..else..`]

Il faut aussi disposer d'actions plus complexes que de simples opérations de calculette (accélérer, monter, descendre, faire un tour de piste en attente, etc.). On doit donc pouvoir :

- regrouper puis abstraire un grand nombre de données élémentaires (nombres, caractères...) pour caractériser une *structure de données* abstraite (`avion` = latitude, longitude, altitude, vitesse, compagnie, origine, destination, etc.)
- regrouper des suites de commandes élémentaires (additions, multiplications, statistiques, classifications, décisions...) pour organiser le *contrôle du programme* et le rendre plus lisible et logique (attribuer une piste à un avion = des inventaires pour vérifier qu'il reste de la place, déplacer dans la mémoire les avions sortis de piste, communiquer avec l'ordinateur de bord, etc.)

Donc : ce qui définit les langages de programmation, c'est

- la façon de représenter symboliquement les **structures de données**
- et la façon de gérer le **contrôle** des programmes.

4 Classement des langages

Langages de spécification Servent à décrire ce qu'un logiciel doit faire, sans dire encore comment le faire. Permettent d'établir formellement les tâches à remplir (eg. mettre un avion en attente, lui attribuer une piste, le faire atterrir...) et les catastrophes à éviter (collisions entre avions, panne sèche pour attente prolongée, arrivée tardive de secours d'urgence...). Ne sont pas des langages de programmation car ils ne sont pas exécutables. Par contre ils en ont toutes les autres particularités : vocabulaire restreint, règles très strictes de formation des « phrases » (formules), aucune ambiguïté possible (branche des maths appelée *logique*). Exemples : spécifications algébriques, VDM, Z, B...

Langages logiques Permettent de résoudre automatiquement des problèmes logiques en essayant systématiquement toutes les formes de solution possibles. Ne marche que pour des propriétés logiques de forme assez restreinte $(P_1 \& \cdots \& P_n) \implies P$. Très utile pour intelligence artificielle, pour interrogation de bases de données, pour « tester » une spécification. Mais assez consommateur de mémoire ou peu performant en temps en général. PROLOG et ses variantes.

Langages fonctionnels Effectuent des transformations des « entrées » vers les « sorties » comme des *fonctions* ou des *applications* des entrées vers les sorties. La loi de composition « rond », comme en maths, permet d'enchaîner des programmes.

Un exemple calculatoire : `f(g(h(x),g(y),z))` peut être vu comme un programme prenant `x`, `y` et `z` en entrée et donnant le résultat de `f` en sortie : $(f \ o \ g \ o \ (h \times g \times Id))$ en maths. La mémoire n'est pas manipulée explicitement. Ce sont les langages actuellement des plus rigoureux pour écrire des logiciels sûrs et prouvables. ML en fait partie. Possède plusieurs variantes dont CAML-LIGHT qui sera utilisé dans ce cours. Autre exemple : LISP et ses variantes, SCHEME...

Importance du typage, en termes d'ensembles de départ et d'arrivée des fonctions.

Langages à objets Les tâches que doit remplir le logiciel sont réparties entre plusieurs « *objets* » qui peuvent offrir des « services » de faible ampleur, mais la coopération des objets permet de réaliser des opérations complexes. Chaque objet possède une petite mémoire locale qui est manipulée explicitement dans les programmes. Très utilisés à l'heure actuelle en raison du style de programmation anthropomorphique qu'ils induisent, ces langages sont en fait plus difficiles à dominer que les langages fonctionnels. (Erreurs de programmation plus difficiles à localiser, preuves formelles de corrections impossibles.) Exemples : Eiffel, Smalltalk, C++, Java ... et OCAML qui allie cette démarche à un style fonctionnel.

Langages impératifs Une grosse *mémoire unique*, manipulée explicitement par le programme. Très difficiles à dominer, mais rapides à l'exécution car proche de la machine. Impossible d'écrire des programmes certifiables formellement, demandent environ un ordre de grandeur supplémentaire qu'en fonctionnel, en nombre de lignes de programme, pour résoudre un même problème. Restent encore parfois nécessaires pour des raisons de performances sur des machines légères, ou pour de très gros calculs. Exemples : ADA, C, Pascal, Cobol, Fortran, Basic, Langage Machine...

5 Bibliographie

Quelques références utiles disponibles à la bibliothèque sont :

1. la référence : X. LEROY, P. WEIS : « *Le manuel de référence du langage CAML* », InterÉditions, 1993.
2. relativement factuel : P. WEIS, X. LEROY : « *le langage CAML* », InterÉditions, 1993.
3. plutôt de niveau licence mais bien dans l'esprit du cours : G. COUSINEAU, M. MAUNY : « *Approche fonctionnelle de la programmation* », Édiscience, 1995.

6 Structures de données, types de données, types, fonctions

Rappel : les langages de programmation de haut niveau se distinguent par leur facilité à manipuler des *structures de données* arbitrairement complexes, et par un *contrôle des programmes* aussi aisé que possible.

Dans ce cours, on commence par les structures de données les plus simples.

Les structures de données en ML sont rigoureusement classifiées en plusieurs *types de données*.

Terminologie : un type de données est défini par 2 choses :

1. un ensemble, dont les éléments sont appelés les *données* du type de données
2. des *fonctions* (au sens mathématique du terme), que l'on appelle aussi les *opérations* du type, car elles sont utilisées par l'ordinateur pour effectuer des « calculs » sur les données précédentes.

Le nom de l'ensemble des données est appelé le *type*. Les éléments de l'ensemble de départ des fonctions sont souvent appelés les *arguments* de la fonction (ou encore ses paramètres). **Principe** fondamental en ML : les types de données sont tous *disjoints*. Ainsi une donnée possède un unique type.

7 Type de données des booléens

Il s'agit d'un ensemble de seulement 2 données, dites *valeurs de vérité* : `bool` = {`true`,`false`}.
Opérations qui travaillent dessus :

```
not : bool  ->  bool
&& : bool  x  bool  ->  bool
|| : bool  x  bool  ->  bool
```

On écrit facilement les tables de ces fonctions puisque le type est fini [et petit]. On remarque ainsi que : `||` et `&&` sont associatives, commutatives, distributive l'une par rapport à l'autre ; `true` est neutre pour `&&` et absorbant pour `||` ; `false` absorbant pour `&&` et neutre pour `||`.
George Boole [1815-1864] : mathématicien anglais qui s'est intéressé à ces propriétés algébriques.

```
# not true ;;
- : bool = false
# not false ;;
- : bool = true
# true && true ;;
- : bool = true
# true && false ;;
- : bool = false
# true || false ;;
- : bool = true
```

8 Type de données des entiers relatifs

Théoriquement `int` = $\mathbb{Z}$. En fait borné, avec borne dépendante de la machine, mais assez grande pour ignorer ce fait ici. (`int` comme « integers »).

Les opérations qui travaillent dessus : `+ - * / mod` et `= < ><= >=` sont de *type* (la notion de « typage » est importante) `int` × `int` → `int`, resp. `int` × `int` → `bool`.

```
# 4 / 2 ;;
-: int = 2
# 2 * 3;;
-: int = 6
# 2 < 3;;
-: bool = true
# 2 = 3;;
-: bool = false
# 3 = (2 + 1);;
-: bool = true
```

9 Type de données des nombres « à virgule flottante »

Historique : par opposition aux 2 chiffres après la virgule genre caisse enregistreuse.
`float` = *{ensemble des nbrs décimaux}*, avec cependant une précision limitée par la machine, mais erreurs négligeables dans le cadre de ce cours. Par abus d'approximations on considère même que `float`=$\mathbb{R}$ et on dispose des opérations suivantes :

`+. -. *. /.` et `exp log sin cos...` et `=. <. >. <=. >=.` qui sont de types `float` × `float` → `float`, resp. `float` → `float`, resp. `float` × `float` → `bool`.

Les comparaisons marchent sans le point. Les comparaisons sont les seuls cas de *surcharges d'opérateurs* que nous verrons dans ce cours.

```
# 4.0 ;;
-: float = 4.0
# 4 ;;
-: int = 4
# 10.0 /. 3.0 ;;
-: float = 3.33333333333
# (10.0 /. 3.0) *. 3.0 ;;
-: float = 10.0
# exp 1.0 ;;
-: float = 2.71828182846
# log (exp 1.0) ;;
-: float = 1.0
# 1.0 <=. 3.5 ;;
-: bool = true
```

Rappel : `int` n'est pas inclus dans `float` puisque deux types sont toujours disjoints. D'où `float_of_int : int -> float` et `int_of_float : float -> int` (partie entière).

10 Type de données des caractères

`char` ≈ *{ce qui peut se taper avec une touche au clavier}*

Lorsque l'on veut produire une donnée de type `char`, on doit l'écrire entre deux backquotes : `` `c` ``. La raison impose d'anticiper un peu sur le cours :

```
# c ;;
> Erreur : l'identificateur  c  n'a pas
> de valeur associee
# let c = 3 ;;
c : int = 3
# c ;;
- : int = 3
# `c` ;;
- : char = `c`
```

Lorsque l'on écrit « `c ; ;` » cette commande de ML veut dire « donner la valeur de `c` ». Le symbole « `c` » joue un rôle similaire à un symbole mathématique (comme par exemple π dont la valeur est par convention 3.14159...). Par conséquent, lorsque l'on écrit « `c` », cela ne veut pas dire « le caractère `c` », mais « la valeur du symbole `c` ». Il faut donc une convention pour dénoter le caractère « `c` » en soi, et ça explique pourquoi on ajoute des backquotes pour faire comprendre à ML qu'on veut parler du caractère mais pas de sa valeur.

Opérations : il y quelques opérations sans grand intérêt pour ce cours.

11 Type de données des chaînes de caractères

`string` = *{suites finies d'éléments du type `char`}*

Opérations : produire un string entre double-quotes `"abcd"`, concaténer deux strings avec l'opération binaire « `^` » de type `string` × `string` → `string`

```
# "bonjour";;
-: string = "bonjour"
# "bonjour" ^ " les amis";;
-: string = "bonjour les amis"
```

ainsi que `string_length` : `string` → `int`,
les comparaisons : `string` × `string` → `bool` (ordre alphabétique).

Enfin `sub_string` : `string` → `int` → `int` → `string` tel que `(sub_string s d n)` retourne la chaîne de caractères composée de `n` caractères consécutifs dans `s`, en partant du $\texttt{d}^{ieme}$ caractère de `s`. Attention, le caractère le plus à gauche de `s` est considéré comme le 0^{ieme}.

```
# sub_string "salut les amis" 6 3 ;;
- : string = "les"
```

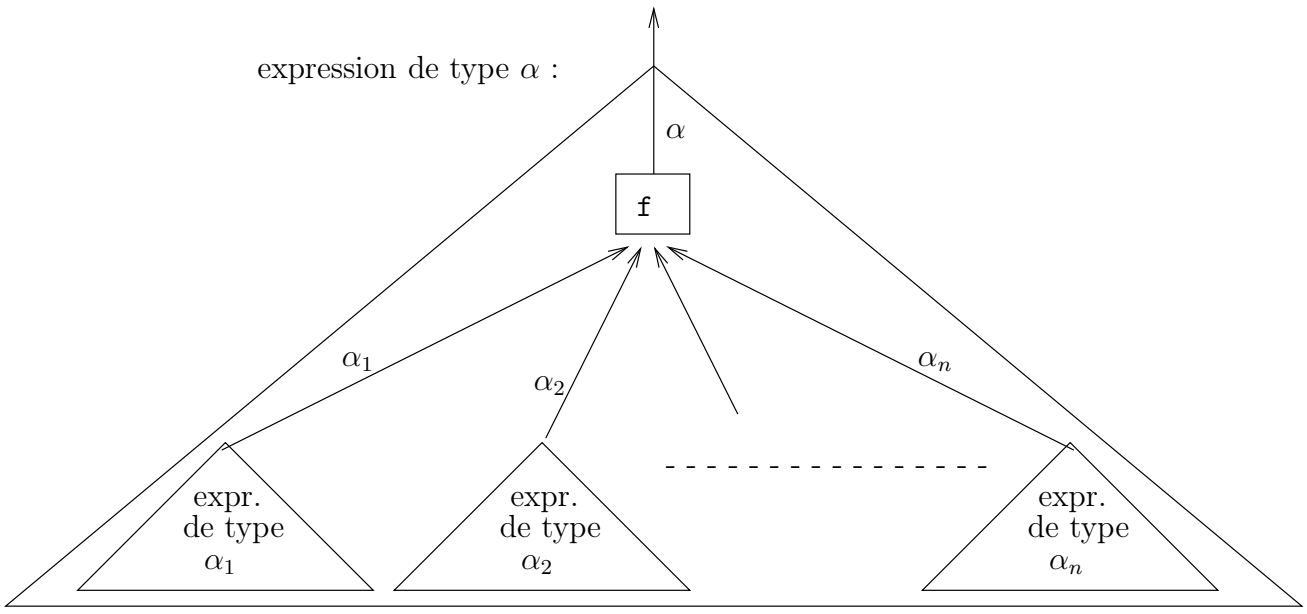
12 Les expressions bien ou mal typées

En généralisant ce qu'on a vu pour les types `bool`, `int`, `float`, `char` et `string`, le typage revient à composer des fonctions de sorte que les ensembles de départ coïncident avec les ensembles d'arrivée, exactement comme en mathématique élémentaire. Exemple, `+` prend deux entiers en arguments, pas autre chose.

```
# ((3 + 4) - 2) < (5 mod 3) ;;
- : bool = false
# "bonjour" + " les amis" ;;
> l'expression "bonjour" est du type string,
> mais est utilisee comme int
# 1.4 + 3.2 ;;
> l'expression 1.4 est du type float,
> mais est utilisee comme int.
# 1.0 + 3.0 ;;
> l'expression 1.0 est du type float,
> mais est utilisee comme int.
```

On peut par contre construire pas à pas des expressions aussi grosses que l'on veut.

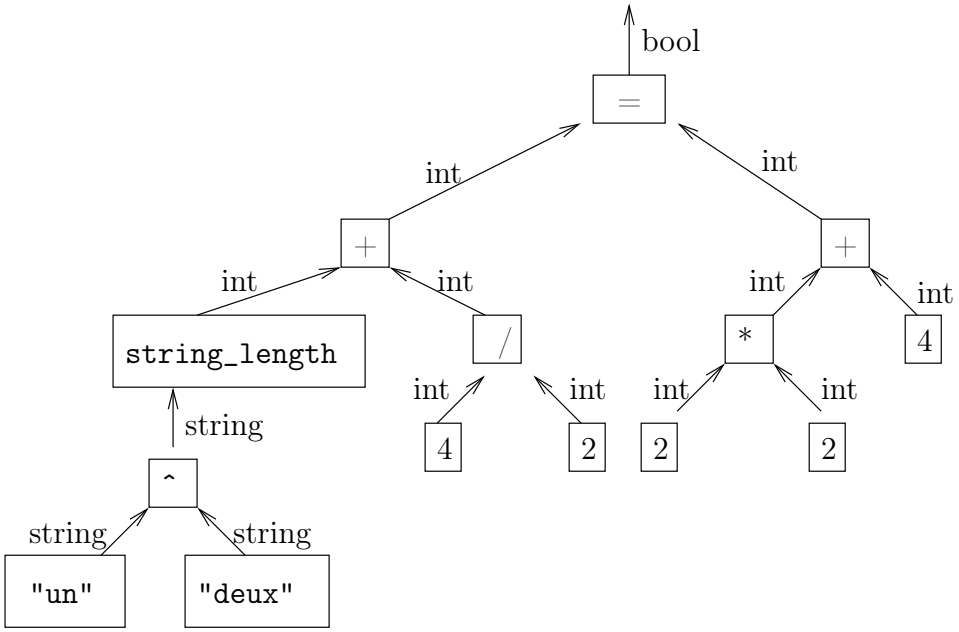
`f` : $\alpha_1 \times \alpha_2 \times \cdots \times \alpha_n \to \alpha$



Exemple :

```
# ((string_length ("un" ^ "deux")) + (4 / 2))
```

```
                          =     ((2 * 2) + 4);;
- : bool = true
```



On doit vérifier en chaque nœud de l'arbre que le type de la fonction qui s'y trouve est correctement placé dans cette expression.

Cours No. 2

Bilan provisoire : pour le moment on n'a vu quasiment qu'une « calculatrice typée ». Raison : il nous manque le contrôle : les déclarations.

1 Enrichir le pouvoir d'expression : les déclarations

Le contrôle est géré essentiellement par le mot clef `let` qui permet de *déclarer successivement* des valeurs nouvelles dans les programmes. C'est comme en math : de définition en définition on finit par manipuler des concepts très sophistiqués, alors que les axiomes de base sont très élémentaires.

Les types de données décrits durant les premiers cours sont assez minimalistes. Ils permettent de faire beaucoup de choses, mais les expressions sont longues à écrire.

Exemple : pour calculer le volume d'un cylindre : surface de la base=πrayon^2, volume=base$\times$hauteur. Supposons rayon=0.5 et hauteur=2 .

```
# 3.14159 *. 0.5 *. 0.5 ;;
-: float = 0.7853975
# 0.7853975 *. 2.0 ;;
-: float = 1.570795
```

Rappels : rôle du « ;; », la contrainte des « . » et « .0 » un peu partout pour le type de données `float`, etc.
Qui pourrait imaginer, en lisant ce programme, ce qu'il est en train de faire ?! On voudrait reconnaître π, le rayon, la hauteur, la base et le volume. En termes de tous les jours on écrirait :

> **En déclarant** (par approximation) que $\pi = 3.14159$
> **Soit** `rayon` $= 0.5$ le rayon du cylindre
> **Soit** `hauteur` $= 1.5$ la hauteur du cylindre
> **On note** `base` $= \pi \times$ `rayon`^{2} sa base
> **Et on définit** `volume` $=$ `base`$\times$`hauteur`

« Déclarer », « Soit », « On note », « On définit » : tout du pareil au même, on associe un *nom* à une *donnée* obtenue en *évaluant* une *expression* placée à droite du « = ». En ML :

`let` *nom* $=$ *expression* ;;

Le nom est souvent appelé un *identificateur* (puisqu'il sert à « donner une identité » à la valeur de l'expression).

```
# let pi = 3.14159 ;;
pi : float = 3.14159
# let rayon = 0.5 ;; (* le rayon du cylindre *)
rayon : float = 0.5
# let hauteur = 2.0 ;; (* la hauteur *)
hauteur : float = 1.5
# let base = pi *. rayon *. rayon ;;
base : float = 0.7853975
# let volume = base *. hauteur ;;
volume : float = 1.570795

# volume ;;
-: float = 1.17809625
```

Noter les `(* commentaires *)`.
On remarque que « - : » a laisse la place à « *nom* : ». Ca veut dire « On a défini *nom*, il est de type `float`, et est égal à ... »

2 La portée d'une déclaration globale

Maintenant, *continuant la même* ***session*** c'est-à-dire à la suite de ce que l'on a déjà programmé, donc dans un *contexte* où les identificateurs `pi`, `rayon`, etc. sont préalablement définis.

```
# let hauteur = 4.0 ;;
hauteur : float = 4.0

# volume ;;
-: float = 1.570795
```

Cela n'a pas multiplié le volume par deux. La raison en est simple : l'expression « `base *. hauteur` » a été calculée *au moment de la déclaration* de `volume`, puis le *résultat* mémorisé sous l'identificateur `volume`. Ensuite, lorsqu'on veut connaître la valeur de `volume`, ça la redonne sans refaire le calcul. (C'est logique, si on fait une déclaration, ce n'est pas pour recalculer la valeur de l'identificateur a chaque fois qu'on l'utilise, ce ne serait pas efficace.) Par contre maintenant si je redéclare [remarquer qu'on a le droit de le faire] :

```
# let volume = base *. hauteur;;
volume: float = 2.3561925

# volume;;
-: float = 2.3561925
```

Cela signifie qu'une déclaration *porte* sur toutes les commandes de programme qui sont écrites entre la commande du `let` et une éventuelle redéclaration du même identificateur, ou la fin de la session. Pour connaître la valeur d'un identificateur, il n'est pas nécessaire de regarder ce qui l'entoure au moment où on l'utilise. On regarde seulement le texte qui *précède* sa déclaration. C'est mieux quand on travaille à plusieurs sur un gros logiciel car ça veut dire que la « sémantique » d'un identificateur ne dépend pas des circonstances ; seul le texte que l'on a écrit « lexicalement » compte, c'est pourquoi cela s'appelle la *portée lexicale*.

3 Déclarations de fonctions

Mais alors comment faire pour dire que le `volume` dépend du rayon et de la hauteur ? et de même pour la base ? Facile : en réalité ce sont des fonctions, et non pas des constantes (la même différence qu'entre `true` qui est une valeur constante et `not` qui est une fonction de `bool` dans `bool`).

`base` : `float` → `float`
`volume` : `float`×`float` → `float`

En maths on écrirait $base(r) = \pi r^2$. En ML on écrit :

```
# let base r = pi *. r *. r;;
base: float -> float = < fun >
```

C'est pas plus difficile que ça. « `base (r)` » marche aussi, mais les parenthèses ne sont pas nécessaires. Le compilateur nous répond que l'on vient de (re)déclarer l'identificateur `base`, que son type est `float` → `float` (c'est donc bien une fonction), et que sa valeur est égale à ... quelque chose qui est une fonction. En effet : une fonction, en informatique, c'est une suite de transformations de symboles comme on l'a déjà vu : et elle est obtenue par une suite de manipulations *illisibles* avec des 0 et des 1. Ici, le compilateur a donc effectué cette transformation de l'expression « `r` *donne* `pi *. r *. r` » en du langage binaire, ca donne quelque chose d'illisible, donc mieux vaut se contenter de dire a l'utilisateur <`fun`> (pour « function »). L'information est incomplète, mais compréhensible.

Maintenant, on peut faire varier le rayon :

```
# base 1.0 ;;
-: float = 3.14159
# base 69.123 ;;
-: float = 15010.4828678
```

Pour plus de lisibilité on pourra écrire :

```
# let carre x = x *. x ;;
carre : float -> float = < fun >
# let base r = pi *. (carre r) ;;
base : float -> float = < fun >
```

Remarquer la place les parenthèses un peu déroutante au début : `(carre r)` au lieu de l'écriture plus habituelle $carre(r)$. La cause en est que sinon ML parenthèse toujours à gauche par défaut : `(pi *. carre) r`, et comme `carre` n'est pas un `float`, ça fait une erreur. Nouvelle habitude de parenthésage à prendre et à mémoriser absolument dans le cadre de ce cours.

4 Les types produits cartésiens

Le volume dépend de deux valeurs (rayon et hauteur). Lorsqu'une fonction `f` prend plusieurs arguments, c'est que l'ensemble de départ (le *type* de départ) est un **produit cartésien** :

```
# let f (x,y,z) = . . . . . ;;
```

et on l'utilise de même dans une expression `(f (x,y,z))`.
On écrira donc :

```
# let volume (r,h) = (base r) *. h ;;
volume : float * float -> float = < fun >
```

Cela veut dire qu'on a déclaré l'identificateur `volume`, son type est `float` $\times$ `float` $\rightarrow$ `float`, et sa valeur celle d'une fonction.

```
# volume ( 0.5 , 3.0 ) ;;
-: float = 2.3561925
```

On remarque « `*` » au lieu de « $\times$ » dans le type de départ de `volume` car « $\times$ » n'est pas au clavier.

Plus généralement, si α et β sont deux types de données existants en ML, alors leur produit cartésien existe également et est noté $\alpha*\beta$.

Rappelons qu'on type de données doit également être muni des opérations qui travaillent dessus ; ce sont les projections :

`fst` : $\alpha*\beta \rightarrow \alpha$
`snd` : $\alpha*\beta \rightarrow \beta$

```
# fst ( 3+5 , 1 ) ;;
-: int = 8
```

Important : ne pas confondre le produit cartésien, qui est un type, avec le produit des entiers (structure de données `int`) qui est une opération.

5 Un argument produit cartésien v.s. plusieurs arguments

Il est en fait possible de déclarer des fonctions avec la forme suivante

```
# let f x y z = ...
```

au lieu de

```
# let f (x,y,z) = ...
```

dans ce cas elle s'utilisent de la même façon avec des espaces entre les arguments au lieu des parenthèses et virgules :

```
# ... (f exp_1 exp_2 exdp_3) ...
```

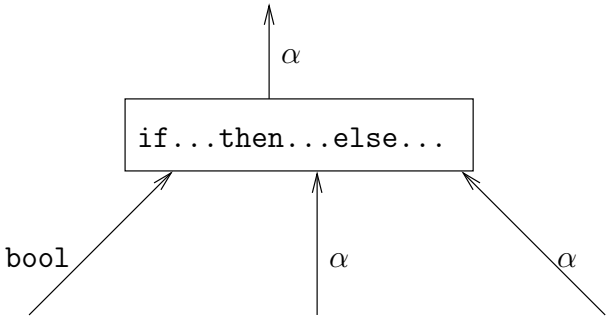
On dit alors que la première forme de `f` possède plusieurs arguments, alors que la seconde forme de `f` ne possède qu'un seul argument qui est un produit cartésien.

Dans le cadre de ce semestre de cours, on n'utilisera que la seconde forme (un seul argument).

6 Expressions conditionnelles

Hormis les déclarations, un élément de contrôle extrêmement utile est *la conditionnelle* :
si *qqchose* **alors** $expression_1$ **sinon** $expression_2$.

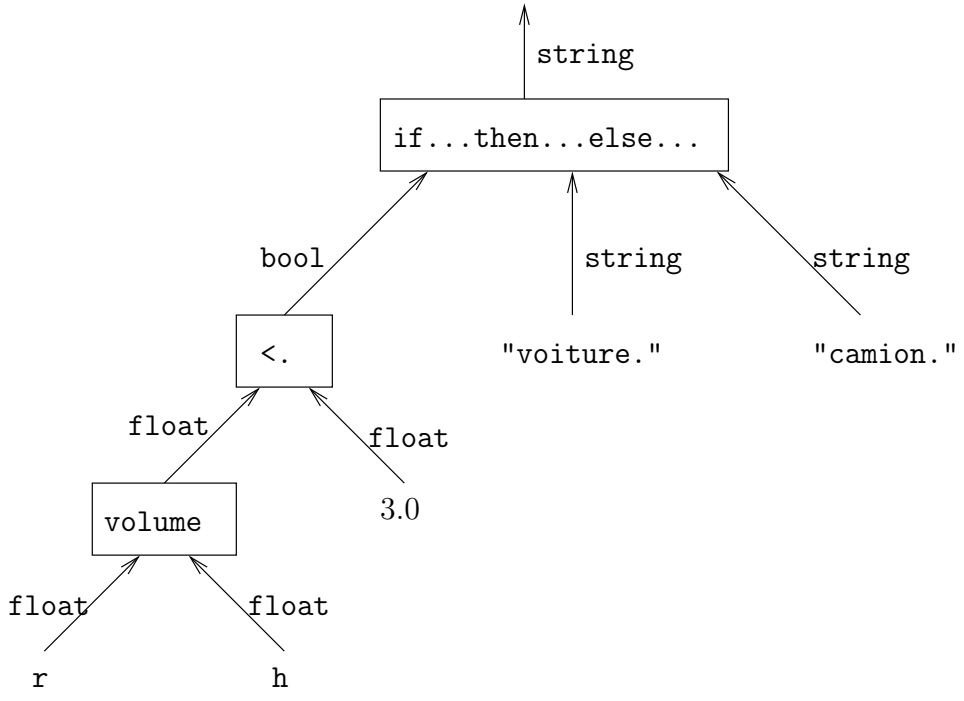
En fait, c'est simplement une fonction pour fabriquer de nouvelles expressions, mais son type est dit *générique* car deux de ses arguments sont de type quelconque (mais le même) :




```
# if (volume (0.5 , 3.0)) <. 3.0
     then "Prendre une voiture."
     else "Prendre un camion." ;;
- : string = "Prendre une voiture."
```

Ca marche bien évidemment aussi avec un `let` :

```
# let transport (r,h) = if volume (r,h) <. 3.0
          then "voiture."
          else "camion.";;
transport : float * float -> string = < fun >
```



ou encore :

```
# let abs x = if x > 0 then x else 0 - x ;;
abs : int -> int = < fun >
```

7 Déclarations locales

Il se peut que certaines déclarations soient utiles pour définir une fonction de manière plus lisible, mais ne méritent pas vraiment d'être associées définitivement à un identificateur.
Exemple : pour calculer ((17 mod 9) + 7) / (10 - (17 mod 9)) + (17 mod 9) on a intérêt à mémoriser provisoirement le résultat de (17 mod 9).
Dans ce cas on utilise une expression de la forme

`let` *provisoire* = *sous-expression* `in` *vraie-expression*

L'identificateur *provisoire* est alors « oublié » en dehors de la *vraie-expression*. C'est comme s'il n'avait jamais été défini.

```
# let p = 17 mod 9 in (p + 7) / (10 - p) + p ;;
-: int = 15
# p + 1 ;;
> l'identificateur  p  ne possede pas de valeur
```

Ca marche bien évidemment aussi avec un `let` :

```
# let transport (r,h) = let capaciteVoiture = 3.0
      in  if volume (r,h) <. capaciteVoiture
             then "voiture."
             else "camion.";;
transport : float * float -> string = < fun >
# capaciteVoiture ;;
> l'identificateur  capaciteVoiture
>    ne possede pas de valeur
```

Ca marche aussi pour des fonctions provisoires :

```
# let plusGrand (x,y,z,t) =
      let sup (a,b) = if a <. b then b else a
          in  sup ( (sup (x,y)) , (sup (z,t)) ) ;;
plusGrand: float * float * float * float
                                -> float = < fun >
# let toto = sup ( 3.0 , 4.0 ) ;;
> l'identificateur sup ne possede pas de valeur
```

[Attention au bon usage des parenthèses...]

Si l'identificateur provisoire était déjà déclaré, il se contente de changer provisoirement de valeur dans *vraie-expression*, et revient immédiatement à sa valeur initiale :

```
# let a = true ;;
a : bool = true
# let a = false
    in  if a
          then "a est provisoirement VRAI."
          else "a est provisoirement FAUX." ;;
- : string = "a est provisoirement FAUX."
# a ;;
- : bool = true
```

8 Déclarations multiples

Pour les déclarations locales comme globales, on peut de plus déclarer plusieurs identificateurs en même temps :

```
# let recadre x =
      let borneInf = 10 and borneSup = 25
      in if x < borneInf then borneInf
         else if x > borneSup then borneSup
              else x ;;
recadre : int -> int = < fun >
```

On peut mettre autant de `and` qu'on veut. Ca marche aussi bien pour les déclarations locales que les déclarations globales.

Attention, ne pas confondre `and`, commande de contrôle liée au `let`, et `&&` qui est la conjonction `bool`×`bool`→`bool`.

Cours No. 3

1 Typage des fonctions

Comment fait ML pour retrouver les ensembles de départ et d'arrivée des fonctions que l'on déclare par let ? *Il résout des équations.*

Il est important de savoir les résoudre également soi-même quand on programme car c'est le seul moyen de *vérifier partiellement* si l'on s'est trompé [+ analogie avec la preuve par neuf, ou les équations aux dimensions en physique].

`let` $f\ (x_1, x_2, \cdots, x_n) =$ *expression*

L'ensemble de départ est nécessairement $\alpha_1 \times \alpha_2 \times \cdots \times \alpha_n$ où α_i est le type de x_i. L'ensemble d'arrivée est nécessairement le type β de *expression*.

De plus, concernant *expression* :
- soit elle est réduite à l'une des variables x_i, et dans ce cas, $\beta = \alpha_i$ (sans indication supplémentaire, donc *généricité* comme pour `if..then..else`)
- soit il existe au moins une fonction $g : \gamma_1 \times \cdots \times \gamma_p \to \delta$ pour laquelle (g .. x_i ..) apparaît dans *expression* avec x_i en position j, et dans ce cas on doit considérer l'équation $\alpha_i = \gamma_j$
- plus généralement, on fait de même à chaque fois que l'on rencontre une fonction $h : \varepsilon_1 \cdots \varepsilon_q \to \nu$ appliquée à q sous-expressions $expr_1 \cdots expr_q$ de type η_1 à η_q

$$(h\ (expr_1\ , \cdots,\ expr_q))$$

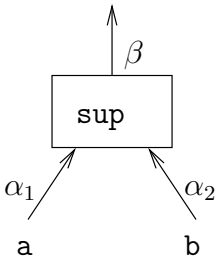
  en posant les q équations $\varepsilon_k = \eta_k$
- à la fin, il ne reste plus qu'à résoudre les équations pour trouver les α_i et β.

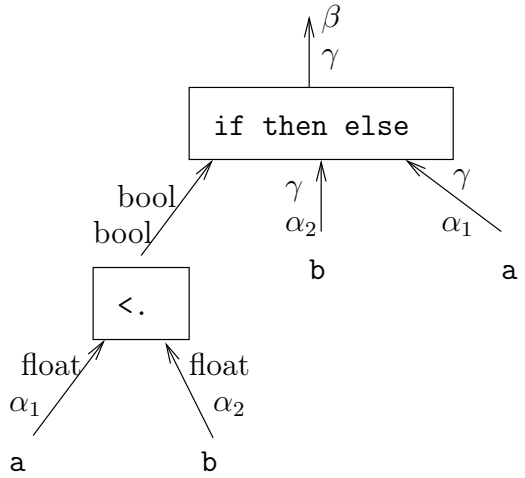
Bref, trouver le type d'une fonction c'est résoudre toutes les équations *type de l'entrée attendue = type de l'expression donnée en argument.*

Exemples :

```
# let sup (a,b) = if a <. b then b else a ;;
```

`a` de type α_1, `b` de type α_2, et le résultat de type β.





- La comparaison prend deux éléments du type `float`, donc α_1=`float` et α_2=`float`.
- `if..then..else` : `bool`$\times\gamma\times\gamma\to\gamma$ donne les équations `bool`=`bool`, $\gamma = \alpha_1$ et $\gamma = \alpha_2$.
- et l'expression donne $\beta = \gamma$.

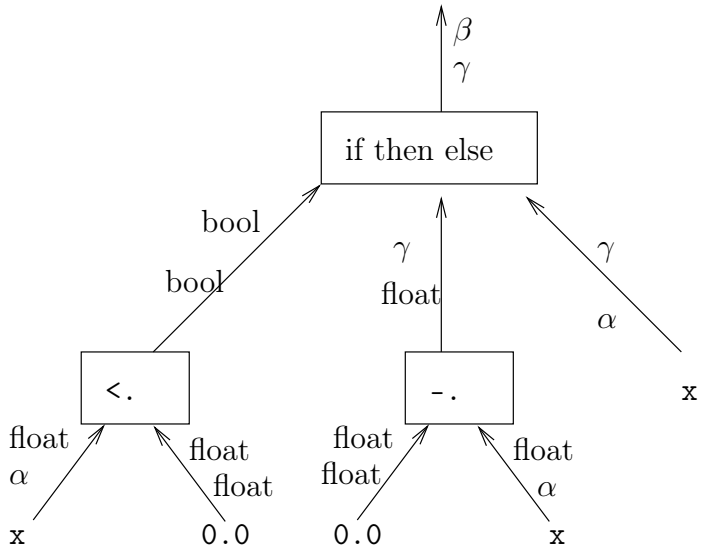
La solution de cet ensemble d'équations est $\alpha_1 = \alpha_2 = \gamma = \beta =$ `float`, donc $\alpha_1 \times \alpha_2 \to \beta$ est `float`$\times$`float`$\to$`float`. ML écrit donc :

```
sup: float * float -> float = < fun >
```

Autre exemple :

```
#let abs x = if x <. 0.0 then 0.0 -. x else x;;
```

On pose `x` de type α (rappel : ML note `'a`), l'expression résultat de type β (ML note `'b`) `if..then..else..` de type `bool`$\times\gamma\times\gamma\rightarrow\gamma$. Les équations qui résultent de l'arbre de l'expression sont alors α=`float`, `float`=`float`, `bool`=`bool`, `float`=`float`, α=`float`, `float`=γ, α=γ, γ=β.

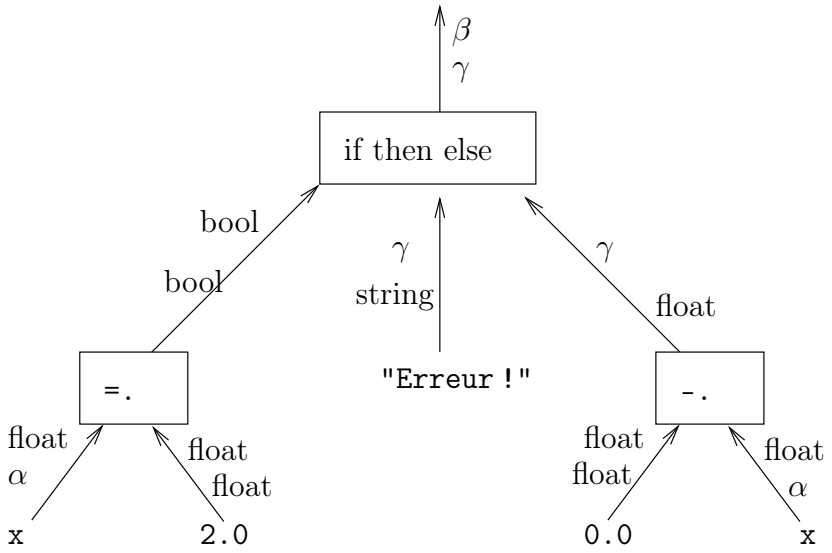


Donc on obtient `abs : float -> float`.

Lorsque le système d'équations ne possède pas de solution, le compilateur ML recrache la commande et explique pourquoi :

```
# let f x = if x =. 2.0 then "Erreur !"
                else x /. (x -. 2.0) ;;
```

En posant `x` :α, l'expression globale de type β, et
`if..then..else` : `bool`$\times\gamma\times\gamma\rightarrow\gamma$,



cela engendre les équations α =`float` (3 fois), un bon nombre de `float`=`float`, `string`= γ, `float`= γ, et $\beta=\gamma$. Ce système d'équation n'a pas de solution :

```
> l'expression  x /. (x -. 2.0) de type float
> est utilisee avec le type string.
```

Remarque : pour qu'il y ait une solution, il faut mettre des types *compatibles* aux branches `then` et `else` du `if`.

Il se peut aussi que les équations ne suffisent pas à déterminer toutes les variables en lettres grecques. Il reste alors une variable libre. On dit que la fonction est *générique*.

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

On remarque que comme l'ordinateur ne peut pas afficher les lettres grecques, il écrit `'a` au lieu de α, `'b` au lieu de β, etc. Une fonction générique peut être utilisée avec n'importe quel type à la place de la variable de type.

```
# Id 3 ;;
-: int = 3
# Id true ;;
-: bool = true
```

2 Typage des fonctions avec déclarations locales

Avec des fonctions provisoires déclarées par des `let..in..` :

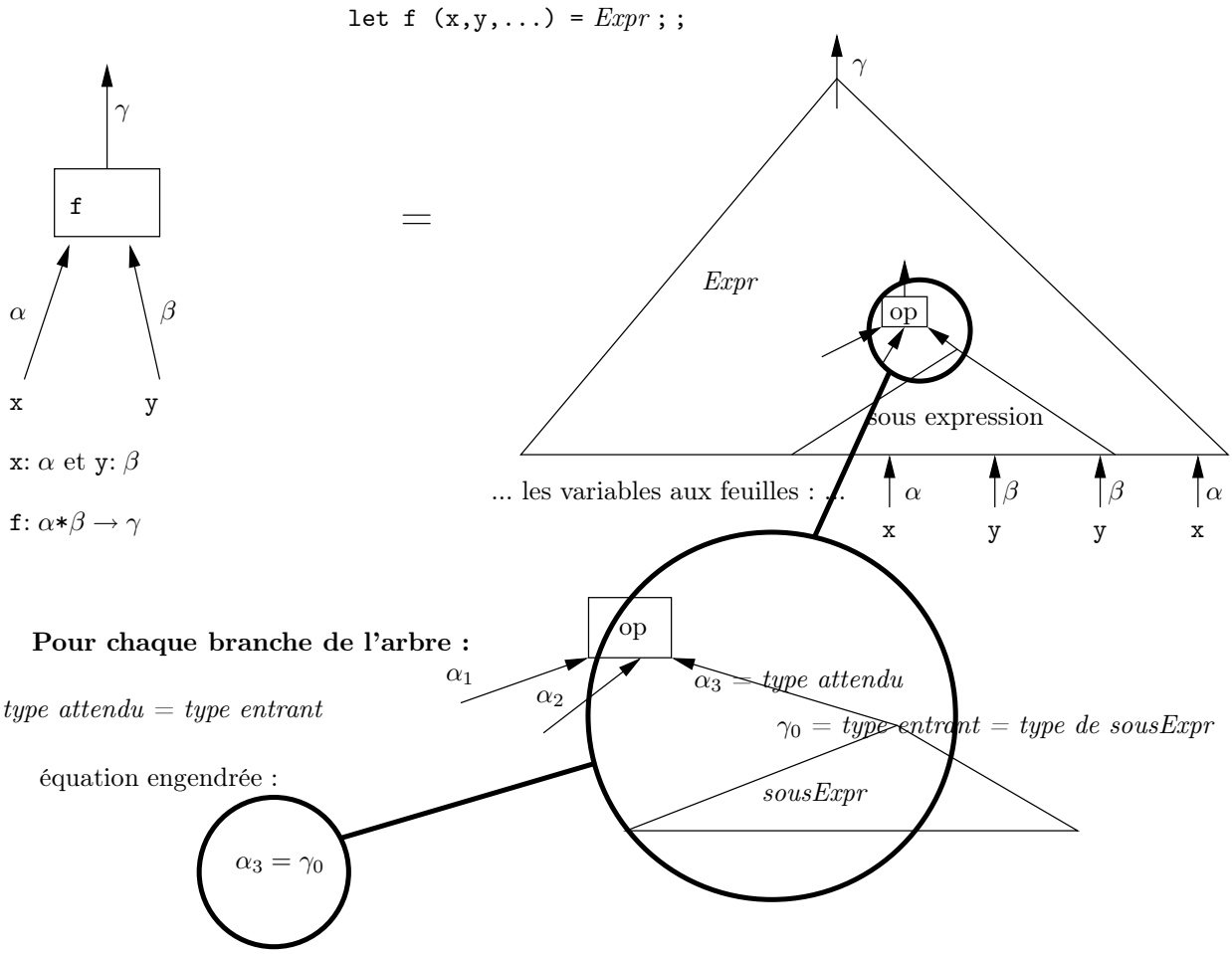
```
# let plusGrand (x,y,z,t) =
      let sup (a,b) = if a <. b then b else a
          in  sup (sup (x,y)) (sup (z,t)) ;;
```

On commence par typer ce qui est dans le `let..in`.
`sup : float*float→float`.
Puis on trace l'arbre de l'expression qui suit le `in`. On utilise comme des lemmes le typage des fonctions déclarées localement.

Ainsi, la sous expression `(sup (x,y))` engendre les équations α_1 =`float` et α_2 =`float`. De même `(sup (z,t))` engendre α_3 =`float` et α_4 =`float`. Enfin l'expression après le `in` engendre deux fois l'équation `float`=`float` et l'équation β =`float`. Bref :

```
plusGrand: float*float*float*float -> float = < fun >
```

Retenir le principe de base : trouver le type d'une fonction c'est résoudre toutes les équations *type de l'entrée attendue = type de l'expression donnée en argument.*



3 Fonctions partielles, traitement d'exceptions

Il peut arriver qu'on n'ait exceptionnellement pas envie, ou possibilité, de donner un résultat à une fonction pour certaines valeurs de son ensemble de départ. Par exemple les fonctions homographiques $f(x) = \frac{ax+b}{cx+d}$.
Il est maladroit, et incomplet, d'écrire :

```
# let f x = (x +. 1.0) /. (x -. 3.0) ;;
```

car `(f 3.0)` retourne des erreurs d'exécution (division par 0) qui donnent une information très incomplète pour l'utilisateur, ou pour mettre au point un logiciel plus gros qui utiliserait `f`.

En écrivant plutôt :

```
# let f x = if x =. 3.0 then
      failwith "3 est asymptote verticale de f"
    else (x +. 1.0) /. (x -. 3.0) ;;
f : float -> float = < fun >
```

on donne plus d'informations, et dans les termes du problème résolu. Si `f` passe dans la branche `then`, alors au lieu de renvoyer une valeur de l'ensemble d'arrivée `float`, ça retourne un message d'erreur, ce qui indique que `3.0` n'est pas dans le domaine de définition.

```
# f 2.0 ;;
- : float = -3.0
# f 3.0 ;;
> Exception: "3 est asymptote verticale de f"
```

Le type de `failwith` est `string`→ α. Ainsi, les types des branches `then` et `else` sont compatibles (α et `float` respectivement). On retient toujours le plus restrictif des deux dans ce cas là. [+ calcul du Type de cet exemple]

On peut bien sûr également *vouloir* faire échouer une fonction, même si le calcul du résultat n'échouerait pas :

```
# let convert_an_mois n  =  if n < 0
      then failwith "Donner un age positif"
      else n * 12 ;;
```

[+ calcul du Type de cet exemple]

4 Les types « enregistrement »

Jusqu'à maintenant, on a vu :

1. les structures de données de base offertes par ML (`bool`, `int`, `float`, `string`, produits cartésiens)
2. les premiers éléments du contrôle des programmes (`let`, `if`, `failwith`).

Cela permet d'avoir une abstraction acceptable du contrôle, mais pas encore sur les structures de données : s'il est possible maintenant grâce au `let` de définir ses propres fonctions, il n'est pas encore possible de *définir ses propres structures de données*.

Le moyen de plus simple est l'*enregistrement*. Il permet de regrouper un nombre fixé de données. Chacune d'elles s'appelle un *champ*.

De même que `let` permet de définir de nouvelles fonctions, le mot clef `type` permet de définir de nouveaux types. Pour définir un nouveau type grâce aux enregistrements on écrit :

```
# type nom = { champ1 : type1 ;
               champ2 : type2 ;
               . . .
               champn : typen }
```

Par exemple :

```
#type avion = { genre:string; compagnie:string;
                altitude:float ; gazole:float ;
                piste: int ; urgence: bool } ;;
Type avion defini.
```

On rappelle qu'un type de données est un *ensemble* de valeurs et des *fonctions* qui travaillent dessus.

L'ensemble des valeurs possibles d'un enregistrement est l'ensemble de toutes les combinaisons possibles des n champs. Il s'agit donc tout bêtement du produit cartésien $nom \approx type_1 \times type_2 \times \cdots \times type_n$.

Le fait de donner un *nom de champ* permet simplement de donner les diverses composantes de la donnée dans n'importe quel ordre. Il ne s'agit donc pas réellement du produit cartésien, mais de qqchose qui lui est isomorphe.

Il reste donc à définir les fonctions qui travaillent sur les types enregistrement.

On construit une donnée de ce type en remplaçant « : $type_i$ » par « = $valeur_i$ » où la valeur est du bon type bien sûr.

```
# let papaTango = { genre= "jet";
    compagnie= "AF"; altitude= 302.3 ;
    gazole=98.5 ; piste=3 ; urgence=false } ;;
papaTango : avion = {genre="jet";
  compagnie="AF"; altitude=302.3; gazole=98.5;
  piste=3; urgence=false}
```

Il n'est pas nécessaire de suivre le même ordre des champs que dans la déclaration du type (ML sait le retrouver) :

```
# let charlie = {  gazole=3.8; piste=1;
    compagnie="TAP"; altitude=886.5 ;
    urgence=false; genre= "biplan" } ;;
charlie : avion = {genre="biplan";
  compagnie="TAP"; altitude=886.5; gazole=3.8;
  piste=1; urgence=false}
```

Important : *en informatique il faut toujours distinguer les* valeurs *et leur* type. *C'est la raison pour laquelle on emploi deux points « : » pour indiquer qu'un identificateur appartient à un certain type et l'égalité « = » pour indiquer qu'un identificateur a pour valeur le résultat d'une certaine expression.*
Il ne faut pas confondre une valeur et son type, de même qu'en mathématique on ne doit pas confondre un élément appartenant à un ensemble et cet ensemble lui même. En écrivant un programme, il faut savoir à tout moment si l'on déclare un type ou si l'on écrit des expressions symboliques qui manipulent des valeurs appartenant aux types préalablement déclarés.

Pour les autres opérations des types enregistrements, il y en a autant que de champs : on retrouve les valeurs des champs en écrivant $nom.champ_i$

```
# let reserve = papaTango.gazole -.
                  papaTango.altitude *. 0.01 ;;
reserve : float = 95.477
```

On peut ainsi faire tous calculs utiles :

```
# let prioritaire a =
   let reserve = a.gazole -. a.altitude *. 0.01
   in   a.urgence or (reserve <. 15.0) ;;
prioritaire : avion -> bool = < fun >
```

```
# let remonte (a,h) = { genre=a.genre ;
    compagnie=a.compagnie;
    altitude=(a.altitude +. h);
    gazole=(a.gazole -. h *. 0.01);
    piste=a.piste; urgence=a.urgence } ;;
remonte : avion*float -> avion = < fun >
```

Donner l'ordre de remonter à un avion via `remonte` ne change pas l'avion `a` donné en paramètre de cette fonction. Cela rend simplement une autre valeur en sortie de la fonction.

```
# remonte (papaTango , 10.0) ;;
- : avion = {genre="jet"; compagnie="AF";
  altitude=312.3; gazole=98.4; piste=3;
  urgence=false}

# papaTango ;;
- : avion = {genre="jet"; compagnie="AF";
  altitude=302.3; gazole=98.5; piste=3;
  urgence=false}

# let papaTango = remonte (papaTango , 10.0) ;;
papaTango : avion = {genre="jet";
  compagnie="AF"; altitude=312.3;
  gazole=98.4; piste=3; urgence=false}
```

En bref : une structure de données « enregistrement » est défini par la commande `type`, suivie du nom qu'on veut lui donner. L'ensemble des valeurs possibles d'un type enregistrement est le produit cartésien des types des champs. On définit les champs d'un type par {$champ_1$:$type_1$; ... ; $champ_n$:$type_n$}. On peut ensuite caractériser une donnée d'un type enregistrement par {$champ_1$=.. ; ... ; $champ_n$=..}. Les opérations qui travaillent sur un type enregistrement donné sont les $.champ_i$ pour chaque champ défini dans la déclaration de type.

5 Champs explicites pour les enregistrements

Il est parfois lourd de devoir écrire `nom.champ` pour tous les champs d'une donnée. C'est le cas de la fonction `remonte`, et ce serait pire si l'on devait utiliser plusieurs fois dans la fonction chacun des champs.

Il est souvent bien pratique d'utiliser la facilité suivante offerte par ML : au lieu d'écrire

```
# let f (x,y,z,..) =
       . . . y.champ1 . . . y.champ2 . . .
       . . . y.champ2 . . . ;;
```

on peut écrire sous forme plus lisible :

```
# let f (x , {champ1 = u ; champ2 = v ; . . .} , z , ..) =
       . . . u . . . v . . .
       . . . v . . . ;;
```

Par exemple

```
#let informe { genre=g; compagnie=c; altitude=h;
      gazole=z; piste=p; urgence=u } =
        let pourGenre="Cet avion est un "
          and pourComp=", il appartient a "
          and vaBien=" et tout va bien"
          and vaMal=" et il faut appeler les pompiers"
        in
        if prioritaire a
          then pourGenre ^ g ^ pourComp ^ c ^ vaMal
          else pourGenre ^ g ^ pourComp ^ c ^ vaBien
 ;;
informe : avion -> string = < fun >

# informe papaTango ;;
- : string = "Cet avion est un jet, il
appartient a AF et tout va bien"

# informe charlie ;;
- : string = "Cet avion est un biplan, il
 appartient a TAP et il faut appeler les
 pompiers"
```

On peut aussi écrire une fonction `change` qui prend en entrée un avion ainsi qu'un nouveau numéro de piste, et une nouvelle altitude à prendre, et retourne le nouvel état qu'aura l'avion s'il effectue ces changements de piste et d'altitude :

```
# let change ( {genre=g; compagnie=c; altitude=h;
    gazole=z; piste=p; urgence=u } , num , alt )  =
    let consom = if alt >. h
                 then (alt -. h) *. 0.01
                 else (h -. alt) *. 0.01
    in {genre=g; compagnie=c; altitude=alt;
    gazole=(z -. consom); piste=num;
    urgence=u} ;;
change : avion*int*float -> avion = < fun >
```

On peut aussi imaginer qu'une piste d'atterrissage soit caractérisée par les informations suivantes :
- la compagnie de l'avion qui vient d'utiliser la piste
- la compagnie de l'avion qui est en cours d'atterrissage
- la compagnie de l'avion qui est prévu ensuite
- s'il est prioritaire.

```
#type piste = {precedent:string; actuel:string;
      prochain:string; priorite1:bool;
      priorite2:bool} ;;
```

A chaque atterrissage, il faut remettre à jour en introduisant l'avion d'après

```
# let suite ( c , p , {precedent=c0; actuel=c1;
     prochain=c2; priorite1=p1; priorite2=p2} )  =
     { precedent=c1; actuel=c2;
     prochain=c; priorite1=p2; priorite2=p } ;;
```

Cours No. 4

1 Résolution de problèmes par découpages

Le but de cette partie est d'introduire des éléments méthodologiques de programmation. Il s'agit d'apprendre les premiers principes de structuration de programmes.

Le développement de logiciels, comme dans toute activité liée à l'ingéniérie et la technologie, doit suivre cinq types d'activité :

- la spécification des besoins où l'on doit définir aussi précisément que possible ce que l'on attend du logiciel à développer, les services qu'il doit rendre, dans quelles conditions, etc. Bref, il s'agit de spécifier « le quoi ».
- le raffinement des spécifications pour définir, en plusieurs étapes de plus en plus détaillées, la structure interne du logiciel. Bref, il s'agit de proposer « un comment ».
- la programmation elle même, en suivant précisément la spécification détaillée obtenue par les raffinements.
- le test pour vérifier si le logiciel obtenu répond effectivement a ses spécifications (à chaque niveau de raffinement et en remontant vers la spécification des besoins).
- enfin la maintenance, qui doit gérer l'évolution des besoins en cours d'utilisation du logiciel, le faire évoluer et réparer les erreurs résiduelles.

2 Programmation structurée

L'essentiel de ce cours de premier semestre se limite à la phase de programmation (les énoncés des exercices pouvant être assimilées à des spécifications détaillées). Il faut bien prendre conscience toutefois que la programmation prend place au sein de la démarche précédemment décrite. Un élément majeur est donc la *lisibilité* des programmes. La technique de *programmation structurée* est le meilleur moyen d'écrire des programmes de qualité. Cette approche est fondée sur une stratégie *diviser pour régner* qui revient elle aussi à découper les problèmes traités en sous-problèmes (moins complexes) :

1. on s'attaque au problème général d'abord
2. on le résout sans s'embarrasser des détails, c'est-à-dire en admettant déjà programmés tous les problèmes subsidiaires
3. pour chaque problème subsidiaire on refait exactement la même démarche
4. à la fin il doit ne rester aucun problème subsidiaire
5. on reporte les fonctions programmées dans l'ordre qu'il faut pour que ML comprenne.

3 Un exemple : la date du lendemain

Etant donnée une date, c'est-à-dire jour, mois et année, on veut calculer celle du lendemain. Il est naturel d'enregistrer ces trois informations sous une seule dénomination qui abstrait la notion de date.

```
#type date = { jour:int; mois:int; annee:int };;
Type date defini.
```

Ensuite, on veut donc écrire une fonction `lendemain` qui prend une date en paramètre et retourne la date du lendemain :

```
# let lendemain d = if not (valide d)
    then failwith "date invalide."
```

Le domaine de définition n'est naturellement pas tout le produit cartésien `int`×`int`×`int`. Il y a des limites sur le numéro du jour, du mois, etc. Pour faciliter la lecture de cet exemple, on suppose que `valide` a déjà été programmé au préalable, ce qui rend lisible notre début de programme ici (si la date n'est pas valide, on échoue). Dans le cas contraire, on est dans le domaine de définition, et le cas facile est lorsqu'il ne s'agit pas du dernier jour du mois. Pareil, c'est un principe général, on suppose avoir programmé au préalable `nbJours` qui donne le nombre de jours de chaque mois. Ca dépend aussi de l'année (cause=bissextiles...) :

```
 else if not (d.jour = (nbJours (d.mois,d.annee)))
      then {jour=(d.jour + 1) ;
            mois=d.mois ;
            annee=d.annee }
```

Si c'est le dernier jour, le cas facile est lorsque ce n'est pas décembre.

```
      else if d.mois < 12
           then { jour=1 ;
                  mois=(d.mois + 1) ;
                  annee=d.annee }
```

Sinon ce n'est pas plus dur.

```
           else { jour = 1 ;
                  mois=1 ;
                  annee=(d.annee + 1) } ;;
lendemain : date -> date = < fun >
```

Seulement en réalité ça planterait car `valide` et `nbJours` ne sont pas programmés. Ca nous a aidé à réfléchir de les supposer écrits, mais ML est trop primaire pour suivre... Donc on écrit **avant** la fonction `valide` :

```
# let valide d = (d.annee >= 0)
    && (d.mois > 0) && (d.mois <= 12)
    && (d.jour > 0) &&
    (d.jour <= (nbJours d.mois d.annee)) ;;
valide : date -> bool = < fun >
```

et en vrai ça planterait encore parce que `nbJours` n'est pas programmée.

```
# let nbJours (m,a) = if m=2
    then (if bissextile a then 29 else 28)
    else if m=4 || m=6 || m=9 || m=11
         then 30
         else 31 ;;
nbJours : int*int -> int = < fun >
```

Et reblabla, ça planterait encore parce que `bissextile` n'est pas programmée.

```
# let bissextile a =
     ((a mod 4 = 0) && not (a mod 100 = 0))
     || (a mod 400 = 0) ;;
bissextile : int -> bool = < fun >
```

Naturellement, il est pénible d'écrire `d.annee`, `d.mois`, etc. On peut donc préférer la version suivante (cette fois remise dans le bon ordre, une fois que la réflexion a été menée).

```
# let bissextile . . .
# let nbJours . . .

# let valide { jour=j; mois=m; annee=a } =
    (a >= 0) && (m > 0) && (m <= 12)
    && (j > 0) && (j <= (nbJours (m,a))) ;;
valide : date -> bool = < fun >

# let lendemain ({ jour=j ; mois=m ; annee=a } as d) =
  if not (valide d)
    then failwith "date invalide."
    else if not (j = (nbJours (m,a)))
           then {jour=(j + 1) ;
                 mois=m ;
                 annee=a }
           else if m < 12
```

```
              then { jour=1 ;
                     mois=(m + 1) ;
                     annee=a }
              else { jour = 1 ;
                     mois=1 ;
                     annee=(a + 1) } ;;
lendemain : date -> date = < fun >
```

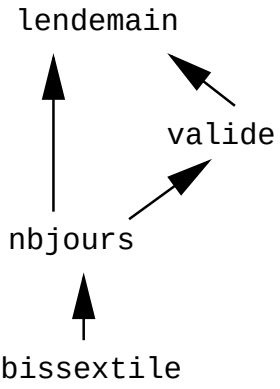
[Explications sur le `as`].

4 Graphe d'appels des fonctions

Pour résoudre le problème de la date du `lendemain`, on a supposé résolus les problèmes de calculer si une date est `valide` et de calculer le `nbJours` de chaque mois. Pour résoudre `valide`, on a supposé `nbJours` résolu. Pour résoudre `nbJours`, on a supposé résolu le prédicat pour savoir si une année est `bissextile`. On a résolu `bissextile` sans rien supposer. Comme ML n'est pas capable de donner une réponse en supposant résolus des problèmes traités ultérieurement, il a fallu après coup « retourner » l'ordre des fonctions.

Si une fonction fait appel à une autre, alors il faut programmer l'autre avant.

Cela conduit à considérer la relation binaire « *est appelée par* », et on peut représenter son graphe, le *graphe d'appels* :



Ainsi, le principe général est de résoudre par la méthode de décomposition en sous-problèmes, puis de tracer le graphe d'appel des fonctions ainsi obtenues, et enfin de trouver un ordre de programmation qui respecte ce graphe. Pour cela on applique le principe suivant : on choisit arbitrairement une fonction sans flèche entrante, on considère le graphe obtenu en supprimant cette fonction et toutes les flèches qui en sortent, et on recommence jusqu'à ce que le graphe soit vide.

```
# let bissextile =..
# let nbJours =..
# let valide =..
# let lendemain =..
```

Remarque importante : pour que ça marche il ne faut pas de cycle dans le graphe d'appel évidemment ! sinon la décomposition en sous-problèmes risque de tourner en rond, et on n'a rien résolu. . .

5 Graphes d'appels avec récursivité mutuelle

En l'état actuel de nos connaissances, on ne peut trouver un ordre de programmation des fonctions utiles pour résoudre un problème que si leur graphe d'appel ne contient aucun cycle.

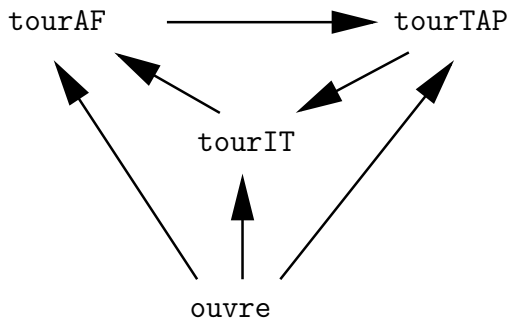
Il y a des cas où la décomposition en sous-problèmes possède des cycles, mais où l'on ne tourne pas en rond car on fait un nombre fini de parcours des cycles du graphe d'appel. On va étudier de plus près le cas des graphes d'appel qui possède un cycle, au travers d'un exemple.

On peut imaginer qu'une piste d'aéroport soit louée à parts égales par trois compagnies `AF`, `IT` et `TAP` et que le contrat stipule que successivement toutes les heures on doive attribuer la piste à une compagnie différente, en suivant

cet ordre. La journée peut commencer par l'une des trois compagnies au choix des responsables de l'aéroport. Il est alors de la responsabilité de l'aéroport de fournir chaque jour une chaîne de caractères appartenant à l'ensemble { `"AF"` , `"IT"` , `"TAP"` } (la compagnie qui commence).
Chaque compagnie, qui veut vérifier à une heure donnée si elle peut utiliser la piste, peut faire appel à l'une des fonctions `tourAF`, `tourIT` ou `tourTAP` respectivement. Chacune prend en arguments le nom de la compagnie qui a commencé, l'heure qu'il est, et retourne un booléen. Pour programmer ces trois fonctions, on est tenté d'appliquer la décomposition en sous-problèmes suivante :

- si l'heure n'est pas un horaire d'ouverture, elle n'est pas valide et on est hors du domaine de définition, sinon, on va distinguer l'heure qui `ouvre` l'aéroport des autres heures :
- supposons par exemple que l'aéroport ouvre à 6h. Pour une heure commençant par 6 (i.e. de 6h à 6h59) il suffit de regarder si l'argument est la bonne compagnie
- pour les autres heures valides, `tourAF` est vraie si `tourTAP` était vraie à l'heure précédente, `tourIT` est vraie si `tourAF` était vraie à l'heure précédente, et `tourTAP` est vraie si `tourIT` était vraie à l'heure précédente.

Le graphe d'appels possède donc un cycle :



Il n'est donc pas possible de trouver un ordre des `let` pour définir les fonctions les unes après les autres qui respecte le graphe d'appels. On dit alors que les fonctions sont *mutuellement récursives*. Comment faire pour résoudre le problème de cette façon ?

On définit les 3 fonctions en même temps avec le `and` *déjà vu, et on place le mot clef* `rec` *qui veut dire qu'il y a récursivité entre ces fonctions, c'est à dire qu'elles « s'utilisent circulairement ».*

```
# let ouvre n = let debut = 6 and fin = 23 in
    if n = debut then true
    else if (n > debut) && (n < fin) then false
         else failwith "horaire incorrect"  ;;
ouvre : int -> bool = < fun >
# let rec tourAF (orig,n) = if ouvre n
        then orig = "AF"
        else tourTAP orig (n-1)
 and tourIT (orig,n) = if ouvre n
        then orig = "IT"
        else tourAF orig (n-1)
 and tourTAP (orig,n) = if ouvre n
        then orig = "TAP"
        else tourIT orig (n-1) ;;
```

et ML répond d'un seul bloc :

```
tourAF : string*int -> bool = < fun >
tourIT : string*int -> bool = < fun >
tourTAP : string*int -> bool = < fun >
```

6 La différence entre `let` et `let rec`

Ecrire seulement (sans le `rec`)

```
# let tourAF (orig,n) = . . .
```

```
 and tourIT (orig,n) = . . .
 and tourTAP (orig,n) = . . . ;;
```

ne marcherait pas. ML persisterait à affirmer qu'il ne peut pas utiliser `tourTAP`, `tourAF`, `tourIT` car ces identificateurs ne sont pas définis. On reçoit le même message d'erreur que si l'on écrit

```
# let x = x + 3 ;;
```

sans avoir défini `x` au préalable : l'identificateur « `x` »utilisé à droite du « `=` » n'est pas défini.

Par contre, en définissant `x` *avant* :

```
# let x = 2 ;;
x: int = 2
# let x = x + 3 and y = x+1 ;;
x: int = 5
y: int = 3
```

la réponse de ML montre que l'identificateur `x` à droite du `=` dénote la valeur de `x` *avant le* `let`. C'est pour cette raison que malgré le `and`, qui définit `tourTAP`, `tourAF` et `tourIT` en même temps, ML les considère comme non définis lorsqu'ils sont utilisés à droite des « `=` » sans le « `rec` ».

Le mot `rec` a simplement pour rôle de demander à ML d'aller voir la valeur de ces trois identificateurs *à l'intérieur* du `let` lui-même plutôt que plus haut dans le texte du programme.

Attention : il faut toujours se convaincre que ça ne va pas tourner indéfiniment. L'usage du `rec` est dangereux de ce point de vue car on peut « emballer la machine » ! Par exemple, si l'on s'interdit d'utiliser le modulo 2, on peut penser à évaluer la parité d'un entier par récursivité croisée entre `pair` et `impair` :

```
# let rec pair n = if n=0 then true
                   else impair (n-1)
   and impair n = if n=0 then false
                  else pair (n-1) ;;
pair : int -> bool = < fun >
impair : int -> bool = < fun >
```

En développant un exemple avec un argument négatif, on constate qu'il est crucial de ne pas oublier les nombres négatifs !

```
# let rec pair n = if n = 0 then true
    else if n < 0 then impair (n+1)
    else impair (n-1)
  and impair n = if n = 0 then false
    else if n < 0 then pair (n+1)
    else pair (n-1) ;;
```

7 Cas des cycles minimaux

Un cas très courant est lorsqu'une fonction est récursivement dépendante d'elle même. Le cycle dans le graphe d'appel est alors réduit à une seule boucle.

Par exemple la factorielle d'un entier naturel `n` vaut 1 sur `n`=0, et sinon le produit de `n` avec la factorielle de `n-1`. Là encore, le `rec` résout la question, et comme il n'y a qu'une fonction, il n'est pas nécessaire d'utiliser le `and`.

```
# let rec fact n = if n < 0 then failwith ".."
     else if n = 0 then 1
          else n * (fact (n-1)) ;;
```

On fait le graphe d'appel très simple et on développe à la main l'exemple `fact 3`.

Cours No. 5

1 Exemples de fonctions récursives

On peut aussi écrire la fonction `pgcd`

```
# let rec pgcd (p,q) = if p < q then pgcd q p
    else if q = 0 then p
         else pgcd ( q , (p mod q) ) ;;
pgcd : int*int -> int = < fun >
# pgcd (45,60) ;;
- : int = 15
```

On développe l'exemple `pgcd(45,60)`.

Avec la récursivité, la fonction puissance utilisée comme une boîte noire en TD peut maintenant être programmée :

```
# let rec puissance (x,n) = if n=0 then 1.0
    else if n<0 then puissance ( (1.0 /. x) , (0-n) )
         else x *. (puissance (x,(n-1))) ;;
```

Graphe d'appel et développement de `puissance 2.0 -2`.

Plus rapide car on fait moins d'opérations pour les grands nombres :

```
# let rec puissance (x,n) = if n=0 then 1.0
   else if n<0 then puissance ( (1.0 /. x) , (0-n) )
        else let y = puissance (x,(n/2))
             in if pair n then y *. y
                else x *. y *. y ;;
```

On peut améliorer la fonction qui détermine le tour des compagnies aériennes.

```
(* ouvre reste inchange *)
# let ouvre n = let debut = 6 and fin = 11 in
    if n = debut then true
    else if (n > debut) && (n < fin) then false
         else failwith "horaire incorrect"  ;;
ouvre : int -> bool = < fun >
# let precedent comp = if comp = "AF"
     then "TAP"
     else if comp = "TAP" then "IT"
          else if comp = "IT" then "AF"
           else failwith "nom incorrect" ;;
# let rec tour (orig,comp,n) =
    if ouvre n then orig = comp
      else tour ( orig , (precedent comp) , (n-1) ) ;;
```

Et graphe d'appel.

Calculer la somme des éléments d'un intervalle [a,b] :

```
# let rec total (a,b) = if a > b
    then failwith "bornes incorrectes"
    else if a = b then a
         else b + (total (a , (b - 1))) ;;
total : int*int -> int = < fun >
```

Savoir si un nombre est premier :

```
let premier n =
    let rec divisibleapres (i,m) =
        if i*i > m then false
        else if m mod i = 0 then true
             else divisibleapres (i+1,m)
    in if n <= 1 then false
       else not (divisibleapres (2, n)) ;;
```

Le nombre d'entiers premiers dans un intervalle :

```
let rec nbpremiers (a,b) = if a > b then 0
    else if premier a then 1 + nbpremiers (a+1,b)
         else nbpremiers (a+1,b) ;;
```

2 Les types « liste »

On se souvient que la programmation repose sur deux aspects : les types de données et le contrôle.

- Du côté du contrôle, on a vu comment programmer des fonctions grâce aux déclarations (locales ou globales), comment les combiner pour former des expressions bien typées, avec des primitives particulières comme `if.then.else`, `failwith` ainsi que la récursivité.
- Du côté des types de données, on a vu les types de base (des booléens aux chaînes de caractères) et les enregistrements qui permettent de regrouper un nombre fixé de données de types différents pour former un nouveau type.

Il ne nous manque plus qu'une chose fondamentale : la possibilité de regrouper un nombre arbitraire de données pour travailler dessus dans leur ensemble. Par exemple, les aiguilleurs du ciel ont a gérer un ensemble d'avions dont le cardinal peut varier dans le temps, et peut aussi être parfois très grand. Il n'est donc pas possible de regrouper cet ensemble d'avions dans un enregistrement car le nombre de champs d'un enregistrement est fixe ; il ne peut pas varier dans le temps. Ce que l'on veut, c'est pouvoir gérer des *listes d'avions* dont le contenu, la taille et l'ordre peut varier (par exemple la liste des avions dont est responsable un aiguilleur donné). Pour cela, ML offre le type des listes.

Les types « liste » sont utiles chaque fois qu'il faut gérer un nombre *inconnu à l'avance* d'éléments appartenant au *même type*. Exemples : carnet d'adresses, tout ensemble fini, file d'attente, stock, liste de membres d'une association, de participants à un examen, etc.

On rappelle qu'un type de données est défini par un ensemble de données (dont le nom est appelé le type) et des fonctions qui travaillent dessus. Ici, on veut faire des listes d'éléments du même type (par exemple d'avions). On considère donc un type quelconque donné α. Le nom du type « liste » correspondant est α `list` (par exemple `avion list`).
c'est l'ensemble des séquences $[v_1; v_2; \cdots; v_n]$ où chacun des v_i est de type α (séquence vide si $n = 0$).

$$\alpha \texttt{ list} = \{[v_1; v_2; \cdots; v_n] | n \in I\!N, v_i \in \alpha\}$$

Les fonctions principales qui travaillent sur les listes sont :

La liste vide : `[]` : α `list`.

La « construction » de liste par ajout d'un élément à gauche : `_ : :_` : $\alpha \times (\alpha$ `list`$) \rightarrow \alpha$ `list`. Par exemple

```
# let l1 = 5 :: [3;1;4] ;;
l1 : int list = [5; 3; 1; 4]
# let l2 = "abc":: ["glop"; "truc"];;
l2 : string list = ["abc"; "glop"; "truc"]
```

Attention de ne pas confondre « `: :` » et « `^` » !
Attention, *une liste doit toujours ne contenir que des élément de même type.* En particulier :

```
# "abc" :: [2; 1; 35];;
> erreur type string utilise avec int
```

La tête de liste (l'élément à gauche) : `hd` : α `list` $\rightarrow \alpha$

```
# hd l1 ;;
- : int = 5
# hd l2 ;;
- : string = "abc"
# hd [] ;;
> erreur
```

(évidemment ça ne marche pas sur une liste vide)

La queue de liste : `tl` : α `list` $\rightarrow$ α `list`

```
# tl l1 ;;
- : int list = [3; 1; 4]
# tl l2 ;;
- : string list = ["glop"; "truc"]
# tl [] ;;
> erreur
```

(même remarque)

La longueur : `list_length` : α `list` $\rightarrow$ `int`

```
# list_length [] ;;
- : int = 0
# list_length l1 ;;
- : int = 4
# list_length l2 ;;
- : int = 3
```

La concaténation : `_@_` : (α `list`) $\times$ (α `list`) $\rightarrow$ α `list`
`[`u_1 `;` ... `;` u_m `]` `@` `[`v_1 `;` ... `;` v_n `]` est égal à `[`u_1 `;` ... `;` u_m `;` v_1 `;` ... `;` v_n `]`

```
# [3;4;5] @ [78;6] ;;
- : int list = [3; 4; 5; 78; 6]
# [] @ l2 ;;
- : string list = ["abc"; "glop"; "truc"]
```

et même remarque importante, tous les éléments doivent avoir le même type

```
# l1 @ l2 ;;
> erreur, l2 de type string list
> utilise avec int list.
```

+ un exemple ultra simple d'utilisation, et typage de la fonction (par exemple `pariteDuPremier`)

3 Premières fonctions sur les listes

On suppose le type avion défini préalablement dans la session :

```
# let genreDuPremier l = (hd l).genre ;;
genreDuPremier : avion list -> string = < fun >
```

On remarque que comme on cherche « `.genre` » sur un élément de la liste, ML déduit que α=`avion` [+ calcul du type]. ***Attention****, cette programmation est trop simpliste :*

```
# genreDuPremier [charlie;papaTango] ;;
- : string = "biplan"
```

```
# genreDuPremier [papaTango] ;;
- : string = "jet"
# genreDuPremier [] ;;
> erreur non rattrapee dans hd
```

Un meilleur programme est

```
# let genreDuPremier l =
  if list_length l = 0 then failwith "vide!"
  else (hd l).genre ;;
```

Du même tonneau :

```
# let second l = if list_length l < 2
   then failwith "trop court"
   else hd (tl l) ;;
second : 'a list -> 'a = < fun >

# let troisieme l = if list_length l < 3
   then failwith "trop court"
   else hd (tl (tl l)) ;;
troisieme : 'a list -> 'a = < fun >
```

On remarque que comme aucune équation ne lie α, `second` et `troisieme` sont génériques [+ calcul du type de `second`].

```
# let ote2premiers l = if list_length l < 2
   then failwith "trop court"
   else tl (tl l) ;;
ote2premiers : 'a list -> 'a list = < fun >

# let double l = l @ l ;;
double : 'a list -> 'a list = < fun >
# double [2;3] ;;
- : int list = [2; 3; 2; 3]

# let append3 (l1,l2,l3) = l1 @ l2 @ l3 ;;
append3 : 'a list * 'a list * 'a list
                          -> 'a list = < fun >
# append3 ( [3;4] , [1;2;3;4;5] , [3] ) ;;
- : int list = [3; 4; 1; 2; 3; 4; 5; 3]
```

4 Premières fonctions récursives sur les listes

```
# let rec longueur l = if l = [] then 0
             else 1 + (longueur (tl l)) ;;
longueur : 'a list -> int = < fun >

# let rec select (l,n) = if n <= 0
    then failwith "position negative interdite !"
    else if list_length l = 0
        then failwith "position trop grande !"
        else if n = 1 then hd l
             else select ( tl l, n-1 ) ;;
select: 'a list * int -> 'a = < fun >

# let rec begaye l = if l = [] then []
    else (hd l)::(hd l)::(begaye (tl l)) ;;
begaye: 'a list -> 'a list = < fun >
```

Cours No. 6

1 L'usage de `match` sur les listes

Avec les listes, il est possible d'utiliser `match` pour récupérer des valeurs dans des variables. Le moyen de faire cela est fondé sur le principe suivant :

- soit une liste est vide, et elle est donc de la forme `[]`
- soit elle est non vide, et elle possède donc un élément le plus à gauche, si bien qu'on peut l'écrire sous la forme `x : :l`. Si une liste `L` est non vide, il est facile de trouver `x` et `l` :
  - `x` = `hd L`
  - `l` = `tl L`

La primitive de contrôle `match` permet alors de faire une programmation par disjonction des cas (vide / non vide). On écrit

... `match` *uneListe* `with`
  `[]` $\rightarrow$ *expression*$_1$
| `x : :l` $\rightarrow$ *expression*$_2$

et cette grosse expression a pour valeur celle de *expression*$_1$ ou celle de *expression*$_2$ selon le cas. C'est souvent plus facile car ca évite de calculer `list_length`, `hd` ou `tl`. Par exemple, au lieu de

```
# let genreDuPremier L =
  if list_length L = 0 then failwith "vide!"
  else (hd L).genre ;;
```

on peut écrire

```
# let genreDuPremier L = match L with
      [] -> failwith "vide!"
  | x::l -> x.genre ;;
genreDuPremier: avion list -> string = < fun >
```

C'est plus direct et ça fait la même chose qu'avant. En plus, on peut faire plusieurs raisonnements par cas imbriqués :

... `match` *uneListe* `with`
  `[]` $\rightarrow$ *expression*$_1$
| x_1 `: :[]` $\rightarrow$ *expression*$_2$
| x_1 `: :`x_2 `: :l` $\rightarrow$ *expression*$_3$

Ainsi, on peut définir `second` comme suit :

```
# let second L = match L with
      [] -> failwith "vide!"
  | x::[] -> failwith "trop court"
  | x::(y::l) -> y ;;
second : 'a list -> 'a = < fun >
```

Cela revient en fait à traiter successivement les liste de longueur 0, puis 1, puis plus de 2. On peut généraliser le raisonnement à n'importe quelle longueur :

... `match` *uneListe* `with`
  `[]` $\rightarrow$ *expression*$_1$
| x_1 `: :[]` $\rightarrow$ *expression*$_2$
| x_1 `: :`x_2 `: :[]` $\rightarrow$ *expression*$_3$
| `. . .`
| x_1 `: :`x_2 `: :... : :`x_n `: :l` $\rightarrow$ *expression*$_{n+1}$

Il n'est pas obligatoire de donner ces $n+1$ cas dans cet ordre là. L'ordre peut être quelconque et c'est parfois utile.

```
# let ote2premiersbis l = match l with
     []  ->  failwith "trop court"
    | x::[] -> failwith "trop court"
    | x::y::l0 -> l0 ;;
ote2premiersbis : 'a list -> 'a list = < fun >
```

On développe `ote2premiersbis [1]` et `ote2premiersbis [1;2;3;4]`.

En fait on peut écrire un programme plus simple grâce à cette propriété bien utile de `match` : dans une expression de la forme

... `match` *uneListe* `with`
  $cas_1 \rightarrow expression_1$
| $cas_2 \rightarrow expression_2$
| . . .
| $cas_n \rightarrow expression_n$

c'est toujours le premier cas qui marche qui est retenu. Ainsi si *uneListe* répond à la fois aux cas_i et cas_j avec $i < j$, alors $expression_i$ sera retenue. Dans notre cas, on peut donc écrire :

```
# let ote2premiersbis l = match l with
      x::y::l0 -> l0
    | l1 -> failwith "trop court" ;;
ote2premiersbis : 'a list -> 'a list = < fun >
```

2 Seuls [] et _ : :_ sont indispensables

Toutes les autres opérations sur les listes peuvent être reprogrammées grâce au `match` :

Tête de liste :

```
# let hdbis l = match l with
      []  -> failwith "liste vide."
    | x::l0 -> x ;;
hdbis : 'a list -> 'a = < fun >
```

On développe l'exemple de `hdbis [1;2;3;4]`.

Queue de liste :

```
# let tlbis l = match l with
      []  -> failwith "liste vide."
    | x::l0 -> l0 ;;
tlbis : 'a list -> 'a list = < fun >
```

On développe l'exemple de `tlbis [1;2;3;4]`.

Longueur : cette fois il faut en plus raisonner par récurrence.

La décomposition en sous-problème plus simple est obtenue par diminution de la taille de la liste en traitant successivement des listes auxquelles on ôte la tête.

```
# let rec list_lengthbis l = match l with
      [] -> 0
    | x::l0 -> (list_lengthbis l0) + 1 ;;
list_lengthbis : 'a list -> int = < fun >
```

On développe l'exemple de `list_lengthbis [1;2]`.

Concaténation de deux listes : même raisonnement sur le premier argument.

```
# let rec concat (l1,l2) = match l1 with
      [] -> l2
    | x::l -> x::(concat (l,l2)) ;;
concat : 'a list * 'a list -> 'a list = < fun >
```

On développe l'exemple de `concat ([1;2],[3])`.

Rappel : il est crucial de s'assurer que la suite de décomposition en sous-problème termine. Ici, on peut le prouver par l'argument que la taille de la liste traitée est inconnue mais *finie*, et on enlève un élément à chaque appel récursif de fonction.

On écrit le plus souvent des fonctions récursives sur les listes car on ne connaît pas à l'avance le nombre d'éléments qu'il faut traiter dans la liste. Par exemple pour répéter chaque élément d'une liste 2 fois de suite, en conservant l'ordre de la liste, on écrit :

```
# let rec repete l = match l with
      []  -> []
    | x::l0 -> x::x::(repete l0) ;;
```

On développe `repete [‘b‘;‘c‘]` à la main pour comprendre le fonctionnement de la fonction pas à pas.

3 Exemples de fonctions simples sur les listes

Extraire le dernier élément d'une liste (rappel : `hd` fournit le premier élément mais le langage ML ne fournit pas de fonction « primitive » donnant le dernier) :

```
# let rec dernier l = match l with
      [] -> failwith "vide!"
    | x::[] -> x
    | x::l1 -> dernier l1 ;;
dernier : 'a list -> 'a = < fun >
```

En suivant le principe de décomposition offert par `match` sur les listes, on écrit d'abord la ligne suivante en fait :

```
    | x::y::l0 -> dernier (y::l0) ;;
```

et comme on n'a pas utilisé la distinction entre `y` et `l0`, on a remplacé `y : :l0` par `l1`. De plus, on a utilisé le fait que `match` choisit le *premier cas* qui marche : ainsi, le cas où `l1` est vide n'arrive jamais (sinon, le programme aurait été faux puisqu'il faudrait retourner `x`).

De façon similaire, le début de la liste, c'est-à-dire celle obtenue en supprimant le dernier élément :

```
# let rec debut l = match l with
      []  -> failwith "vide!"
    | x::[] -> []
    | x::l0 -> x::(debut l0) ;;
debut : 'a list -> 'a list = < fun >
```

Un palindrome est une suite d'éléments qui est symétrique par rapport à son milieu. On peut utiliser les fonctions précédentes pour programmer le prédicat `palindrome` :

```
# let rec palindrome l = match l with
    [] -> true
  | x::[] -> true
  | x::l0 -> (x = (dernier l0))
             & (palindrome (debut l0)) ;;
palindrome : int list -> bool = < fun >
```

On développe un exemple impair qui marche `[2;5;2]`, un exemple pair qui marche `[2;3;3;2]` et un qui ne marche pas `[1;2]`.

Cours No. 7

1 Sous-ensembles finis d'un type quelconque

Un grand classique de l'informatique est la gestion de « fichiers », au sens commun du terme, c'est-à-dire d'un ensemble fini d'informations de même nature : carnet d'adresses, fichier de stock, fichier de cartes d'identité, d'immatriculations de voiture, etc. Dans son principe, la gestion de telles bases de données revient tout simplement à gérer un *ensemble fini* (éventuellement grand) d'informations de même nature. Bref, il s'agit de gérer des sous-ensembles finis d'un type de donné fixé (souvent un type enregistrement).

Un moyen assez simple (bien que peu efficace pour les grand cardinaux) pour gérer de tels ensembles est d'utiliser des listes *non redondantes*. C'est ce que nous allons faire ici. Les opérations ensemblistes les plus courantes effectuées sur ces bases de données sont l'*insertion* d'un élément, la *recherche* d'un élément, la *suppression* d'un élément, la *mise à jour* d'un élément, l'*union* et l'*intersection* de deux ensembles, la *différence symétrique*, etc. [+ expliquer à quoi cela correspond en termes de « bases de données »].

Afin de fixer les idées, nous allons considérer que les éléments que l'on gère sont d'un type *personne* qui est un enregistrement, mais il est important de bien garder en tête que toutes les fonctions que nous allons donner marchent pour n'importe quel type, *pourvu que l'on ait programmé une fonction qui compare deux éléments du type*[1].

```
# type personne = { nom:string; prenom:string;
    secu:int; adresse:string list } ;;
Type personne defini.
```

Comme on le sait, le numéro de sécu caractérise une personne, donc

```
# let egal (p1,p2) = p1.secu = p2.secu ;;
egal : personne * personne -> bool = < fun >
```

[+ rappeler la différence entre les deux « = »]. On peut aussi programmer un `test` pour savoir si une personne répond bien à un numéro de secu donné :

```
# let test (p,n) = p.secu = n ;;
test : personne * int -> bool = < fun >
```

On va donc gérer des ensembles finis de personnes en utilisant des listes de type `personne list`. La fonction la plus simple est de savoir si une base de données est vide

```
# let vide l = match l with
      [] -> true
    | l -> false ;;
vide : 'a list -> bool = < fun >
```

On constate que le type des éléments n'est pas utilisé. Donc la fonction est plus générale que prévu (tant mieux).
Le test d'appartenance :

```
# let rec appartient (p,l) = match l with
      [] -> false
    | x::l' ->  if egal (x,p) then true
                else appartient (p,l') ;;
appartient : personne * (personne list)
                        -> bool = < fun >
```

Et on développe l'exemple de la liste

```
# let carnet = let a=["Univ Evry";"Evry cedex"]
```

[1] ici `egal` ou `test` selon ce que l'on veut faire

```
  in [{nom="Dupont"; prenom="Gus";
                  secu=1; adresse=a} ;
     {nom="Durand"; prenom="Paul";
                  secu=2;  adresse=a} ;
     {nom="Duval"; prenom="Agnes";
                  secu=3; adresse=a} ] ;;
carnet : personne list = [{nom="Dupont"; . . . ]
```

avec `appartient ({...Durand...} , carnet)`.

Dans toute la suite on supposera n'avoir à faire qu'à des listes correctes, c'est-à-dire non redondantes. Toutes les opérations doivent préserver cette correction. Ainsi l'insertion n'est pas tout à fait réduite à « : : »

```
# let insert (p,l) = if appartient (p,l)
    then l
    else p::l ;;
insert : personne * (personne list)
              -> personne list = < fun >
```

Et on développe un exemple dans chaque branche avec `carnet`.
La recherche, on donne un numéro de sécu, et on obtient un booléen qui dit s'il y a dans la liste une personne ayant ce numéro.

```
# let rec recherche (n,l) = match l with
      [] -> false
    | p::l' -> if test (p,n) then true
               else recherche (n,l') ;;
recherche : int * (personne list) -> bool = < fun >
```

La suppression :

```
# let rec supprime (n,l) = match l with
      [] -> []
    | p::l' -> if test (p,n) then l'
               else p::(supprime (n,l')) ;;
supprime : int * (personne list)
                 -> personne list = < fun >
```

Et on développe `supprime ( {...Durand...} , carnet )`.

Seules les listes non redondantes sont correctes pour représenter un ensemble :

```
# let rec correcte l = match l with
      [] -> true
    | x::l' -> if appartient (x,l') then false
               else correcte l' ;;
correcte : personne list -> bool = < fun >
```

Et on développe sur l'exemple `carnet`.
Trouver l'adresse à partir du numéro de sécu :

```
# let rec trouveADR (n,l) = match l with
      [] -> failwith "Numero inconnu."
    | p::l' -> if test (p,n) then p.adresse
               else trouveADR (n,l') ;;
trouveADR : int * (personne list)
                  -> string list = < fun >
```

Trouver le nom à partir du numéro de sécu :

```
# let rec trouveNOM (n,l) = match l with
      [] -> failwith "Numero inconnu."
    | p::l' -> if test (p,n)
               then p.prenom ^ " " ^ p.nom
               else trouveNOM (n,l') ;;
trouveNOM : int * (personne list)
                        -> string = < fun >
```

On développe `trouveNOM (0,carnet)` et idem avec `1`.
La mise a jour (ici un changement d'adresse)

```
# let rec ajour (n,a,l) = match l with
      [] -> failwith "Numero absent."
    | p::l' -> if test (p,n)
              then {nom=p.nom; prenom=p.prenom;
                   secu=n; adresse=a} :: l'
              else p::(ajour n a l') ;;
ajour : int * (string list) * (personne list)
                     -> personne list = < fun >
```

L'union :

```
# let rec union (l1,l2) = match l1 with
      [] -> l2
    | p::l -> if appartient (p,l2)
              then union (l,l2)
              else p::(union (l,l2)) ;;
union : (personne list) * (personne list)
                -> personne list = < fun >
```

L'intersection :

```
# let rec inter (l1,l2) = match l2 with
      [] -> []
    | p::l -> if appartient (p,l1)
              then p::(inter (l1,l))
              else inter (l1,l) ;;
inter : (personne list) * (personne list)
                 -> personne list = < fun >
```

La différence :

```
#let rec prive (l1,l2) = match l1 with
      [] -> []
    | p::l -> if appartient (p,l2)
              then prive (l,l2)
              else p::(prive (l,l2)) ;;
prive : (personne list) * (personne list)
                  -> personne list = < fun >
```

La différence symétrique (tout ce qui est dans un seul des deux ensembles) :

```
# let rec delta (l1,l2) = match l1 with
      [] -> l2
    | p::l -> if appartient (p,l2)
              then delta ( l , (supprime (p.secu,l2)) )
              else p::(delta (l,l2)) ;;
delta : (personne list) * (personne list)
                      -> personne list = < fun >
```

Chercher le rang d'un élément dans la liste, en donnant un message d'erreur s'il n'est pas dedans :

```
# let rec cherche (n,l) = match l with
      [] -> failwith "Pas dedans."
    | p::l' -> if test (p,n) then 1
               else 1 + (cherche (n,l')) ;;
cherche : int * (personne list) -> int = < fun >
```

On développe `cherche (2,carnet)` et `cherche (4,carnet)`.
Rendre une liste non redondante, même si elle l'était avant :

```
# let rec epure l = match l with
      [] -> []
    | p::l' -> if appartient (p,l')
               then epure l'
               else p::(epure l') ;;
epure : personne list -> personne list = < fun >
```

Cours No. 8

1 Sous-ensembles finis d'un type ordonné

La gestion de « fichiers » (au sens commun du terme) est en fait presque toujours optimisée en tirant parti d'un *ordre total* sur l'ensemble des éléments du type géré. On utilise encore des listes non redondantes, mais cette fois elles sont en plus triées. Il s'agit toujours de gérer des sous-ensembles finis d'un type de donné fixé (souvent un type enregistrement). Et les opérations ensemblistes les plus courantes effectuées sur ces bases de données sont toujours l'*insertion* d'un élément, la *recherche* d'un élément, la *suppression* d'un élément, la *mise à jour* d'un élément, l'*union* et l'*intersection* de deux ensembles, la *différence symétrique*, etc.

Afin de fixer les idées, nous allons encore considérer des éléments de type *personne*, mais il est important de bien garder en tête que toutes les fonctions que nous allons donner marchent pour n'importe quel type, *pourvu que l'on ait programmé des fonctions qui comparent deux éléments du type*, ici `egal` pour l'égalité, et `infstrict` pour l'ordre strict total :

```
# type personne = { nom:string; prenom:string;
    secu:int; adresse:string list } ;;
Type personne defini.
# let egal (p1,p2) = p1.secu = p2.secu ;;
egal : personne * personne -> bool = < fun >

# let infstrict (p1,p2) = p1.secu < p2.secu ;;
infstrict : personne * personne -> bool = < fun >
```

On va donc gérer des ensembles finis de personnes en utilisant des listes *triées* de type `personne list`. Croissantes pour fixer les idées (des programmes similaires marchent pour les listes décroissantes). Première fonction : vérifier la croissance et la non redondance d'une liste.

```
# let rec croissante l = match l with
      [] -> true
    | [x] -> true
    | x::y::l' -> if infstrict (x,y)
                  then croissante (y::l)
                  else false ;;
croissante : personne list -> bool = < fun >
```

Et on développe l'exemple de la liste

```
# let carnet = let a=["Univ Evry";"Evry cedex"]
  in [{nom="Dupont"; prenom="Gus";
                   secu=1; adresse=a} ;
     {nom="Durand"; prenom="Paul";
                   secu=2;  adresse=a} ;
     {nom="Duval"; prenom="Agnes";
                   secu=3; adresse=a} ] ;;
carnet : personne list = [{nom="Dupont"; . . . ]
```

Le test d'appartenance et l'insertion deviennent, *en supposant que la liste est croissante* :

```
# let rec appartient (p,l) = match l with
      [] -> false
    | x::l' ->  if infstrict p x then false
                else if egal (p,x) then true
                   else appartient (p,l') ;;
appartient : personne * (personne list)
                        -> bool = < fun >

# let rec insert (p,l) = match l with
```

```
      [] -> [p]
    | x::l' -> if infstrict (p,x) then p::l
               else if egal (p,x) then l
                 else x::(insert (p,l')) ;;
insert : personne * (personne list)
             -> personne list = < fun >
```

On développe `appartient ( {...Durand...} , carnet )`.
Trier une liste quelconque :

```
# let rec trie l = match l with
      [] -> []
    | p::l' -> insert ( p , (trie l') )  ;;
trie : personne list -> personne list = < fun >
```

Dans toute la suite on supposera n'avoir à faire qu'à des listes croissantes. Toutes les opérations doivent préserver cette propriété.
La suppression :

```
# let rec supprime (n,l) = match l with
      [] -> []
    | p::l' -> if n < p.secu then l
               else if n = p.secu then l'
                 else p::(supprime (n,l')) ;;
supprime : int * (personne list)
                 -> personne list = < fun >
```

Et on développe `supprime ( {...Durand...} , carnet )`.
Trouver l'adresse à partir du numéro de sécu :

```
# let rec trouveADR (n,l) = match l with
      [] -> failwith "Numero inconnu."
    | p::l' -> if n < p.secu
               then failwith "Numero inconnu."
               else if n = p.secu then p.adresse
                    else trouveADR (n,l') ;;
trouveADR : int * (personne list)
                  -> string list = < fun >
```

Trouver le nom à partir du numéro de sécu :

```
# let rec trouveNOM (n,l) = match l with
      [] -> failwith "Numero inconnu."
    | p::l' -> if n < p.secu
               then failwith "Numero inconnu."
               else if n = p.secu
                  then p.prenom ^ " " ^ p.nom
                  else trouveNOM (n,l') ;;
trouveNOM : int * (personne list)
                        -> string = < fun >
```

On développe `trouveNOM (0,carnet)` et idem avec `1`.
La mise a jour (ici un changement d'adresse)

```
# let rec ajour (n,a,l) = match l with
      [] -> failwith "Numero absent."
    | p::l' -> if n < p.secu
               then failwith "Numero absent."
```

```
               else if n = p.secu then
                 {nom=p.nom; prenom=p.prenom;
                   secu=n; adresse=a} :: l'
                 else p::(ajour (n,a,l')) ;;
ajour : int * (string list) * (personne list)
                     -> personne list = < fun >
```

L'union :

```
# let rec union (l1,l2) = match l1 with
      [] -> l2
    | p::l1' -> ( match l2 with
                  [] -> l1
                | q::l2' -> if infstrict (p,q)
                  then p::(union (l1',l2))
                  else if egal (p,q)
                    then union (l1',l2)
                    else q::(union (l1,l2'))
                ) ;;
union : (personne list) * (personne list)
                -> personne list = < fun >
```

Après l'union, il reste à programmer l'intersection, la différence, la différence symétrique, la recherche du rang d'un élément dans la liste. Enfin nous terminerons ce chapitre sur les listes triées avec la programmation du tri reconnu le plus rapide en moyenne, et souvent utilisé pour des listes de grande taille.

L'intersection :

```
# let rec inter (l1,l2) = match l1 with
      [] -> []
    | p::l1' -> ( match l2 with
                  [] -> []
                | q::l2' -> if infstrict (p,q)
                  then inter (l1',l2)
                  else if egal (p,q)
                    then p::(inter (l1',l2'))
                    else inter (l1,l2')
                ) ;;
inter : (personne list) * (personne list)
                 -> personne list = < fun >
```

La différence :

```
# let rec prive (l1,l2) = match l1 with
      [] -> []
    | p::l1' -> ( match l2 with
                  [] -> l1
                | q::l2' -> if infstrict (p,q)
                  then p::(prive (l1',l2))
                  else if egal (p,q)
                    then prive (l1',l2')
                    else prive (l1,l2')
                ) ;;
prive : (personne list) * (personne list)
                 -> personne list = < fun >
```

La différence symétrique (tout ce qui est dans un seul des deux ensembles) :

```
# let rec delta (l1,l2) = match l1 with
```

```
      [] -> l2
    | p::l1' -> ( match l2 with
                  [] -> l1
                | q::l2' -> if infstrict (p,q)
                  then p::(delta (l1',l2))
                  else if egal (p,q)
                    then delta (l1',l2')
                    else q::(delta (l1,l2'))
                ) ;;
delta : (personne list) * (personne list)
                      -> personne list = < fun >
```

Chercher le rang d'un élément dans la liste, en donnant un message d'erreur s'il n'est pas dedans :

```
# let rec cherche (n,l) = match l with
      [] -> failwith "Pas dedans."
    | p::l' -> if n < p.secu
               then failwith "Pas dedans."
               else if n = p.secu then 1
                 else 1 + (cherche (n,l')) ;;
cherche : int * (personne list) -> int = < fun >
```

On développe `cherche (2,carnet)` et `cherche (4,carnet)`.

Enfin un exemple de tri plus rapide que le tri « par insertions » vu plus haut est fondé sur le principe suivant.

1. on prend le premier élément `p` de la liste, on sépare la liste en deux listes plus petites appelées `petits` et `grands` : `petits` contenant en vrac les éléments plus petits que `p` et `grands` ceux plus grands
2. on trie selon le même principe les listes `petits` et `grands`
3. il ne reste plus qu'à les mettre bout à bout avec `p` au milieu.

```
type deuxlistes = { petits: personne list ;
                    grands: personne list } ;;
Type deuxlistes defini.
```

La séparation se programme comme suit :

```
# let rec separe (p,l) = match l with
      [] -> { petits=[] ; grands=[] }
   | x::l' -> let d = separe (p,l') in
              if infstrict (x,p) then
                { petits=(x::d.petits) ;
                  grands=d.grands }
              else if egal (x,p) then d
               else { petits=d.petits ;
                  grands=(x::d.grands) } ;;
separe : personne * (personne list)
                     -> deuxlistes = < fun >
```

Et enfin le tri plus rapide :

```
let rec trirapide l = match l with
      [] -> []
    | p::l' -> let d = separe (p,l') in
               (trirapide d.petits) @
                  (p::(trirapide d.grands)) ;;
trirapide : personne list
                     -> personne list = < fun >
```

Cours No. 9

1 Preuves de terminaison de fonctions récursives

L'un des sujets majeurs de l'informatique est la correction des programmes. Pour cela, il faut d'abord définir ce que l'on attend d'une fonction programmée, puis prouver que cette fonction répond à cette attente.

Pour aborder rigoureusement cette question, la définition de ce que l'on attend d'une fonction doit être exprimée sous forme d'une propriété mathématique qui lie les arguments d'entrée de la fonction et le résultat calculé par cette fonction.

Par exemple, si l'on considère la fonction suivante :

```
let rec somme n = if n < 0
    then failwith "uniquement des naturels SVP"
    else if n = 0 then 0
         else n + (somme (n - 1)) ;;
```

l'objectif est de démontrer que pour tout entier naturel n, `somme`(n) calcule la somme des n premiers entiers. Une propriété mathématique qui peut permettre de s'en convaincre est par exemple :

$$\forall n \in I\!N,\ \texttt{somme}(n) = \frac{n(n+1)}{2}$$

Cependant la propriété précédente n'aura de sens que si `somme`(n), qui est n'est défini que par un programme informatique, possède une valeur mathématique dans $I\!N$. *Cela suppose donc de prouver au préalable que la fonction fournit une valeur pour tous les* $n \in I\!N$. Ceci impose de prouver en premier lieu que la fonction *termine* puis que la valeur qu'elle retourne après terminaison (éventuellement non définie, via un `failwith`) satisfait la propriété.

Ce chapitre traite les preuves de terminaison préalables, le suivant traitera de les preuves de propriétés.

Le premier cas facile, mais qui doit être énoncé est le suivant :
si une fonction n'est pas récursive et n'appelle que des fonctions qui terminent, alors elle termine.

Le cas des fonctions récursives est naturellement plus complexe :
ce qui fait qu'une fonction récursive termine, c'est que les appels récursifs successifs qu'elle engendre se font avec des arguments qui convergent vers un cas de base, en un nombre fini d'appels récursifs.
Les cas de base sont par définition les arguments de la fonction qui n'engendrent pas de nouvel appel récursif. Par exemple, la fonction `somme` admet tous les entiers négatifs ou nuls comme cas de base avec cette définition.

Dans l'exemple de la `somme`, ce qui assure que la fonction termine, c'est que les appels récursifs successifs font *décroître strictement* un entier naturel (l'entier n). En effet, dans $I\!N$, la propriété suivante est vraie :

il n'existe pas de suite infinie strictement décroissante

On en déduit donc que la suite des arguments des appels récursifs de `somme` est finie (puisqu'elle est strictement décroissante) et que par conséquent la fonction `somme` termine.

Cette façon de prouver la terminaison d'une fonction est générale, cependant les arguments des fonctions ne sont pas toujours des entiers naturels. Il faut donc définir en toute généralité des relations d'ordre sur n'importe quel type de donnée (booléens, listes, etc.) qui possèdent cette propriété de terminaison :

Définition : Une relation d'ordre $\leq$ sur un ensemble quelconque E est dite *bien fondée* (ou encore *nœthérienne*) si et seulement si il n'existe pas de suite infinie d'éléments de E strictement décroissante pour cet ordre.

Exemple :

- $I\!N$ muni de l'ordre habituel est nœthérien.
- Par contre $Z\!\!Z$ ne l'est pas puisque par exemple la suite $u_n = -n$ est infinie et strictement décroissante.
- $I\!R$ ou $I\!R^+$ ne le sont pas non plus puisque par exemple la suite $u_n = \frac{1}{n}$ est strictement décroissante et infinie.
- L'ensemble `bool`=`{false,true}` muni de l'ordre tel que `false`<`true` est nœthérien. En effet, toute suite strictement décroissante est bloquée sitôt qu'elle a atteint `false` car il n'existe pas de booléen plus petit. On en déduit que les suites strictement décroissantes sont de longueur au plus 2 (`true` suivi de `false`) ; il n'y a donc pas de suite infinie strictement décroissante.

Définition : Soient $(E_i, \leq_i)$ des ensembles munis de relations d'ordre. L'*ordre lexicographique* sur le produit cartésien $E_1 \times \cdots \times E_n$ est défini par : $(a_1, \cdots, a_n) \leq (b_1, \cdots, b_n)$ si et seulement si, soit $(a_1, \cdots, a_n) \leq (b_1, \cdots, b_n)$, soit il existe un indice j compris entre 1 et n tel que $a_j <_j b_j$ et pour tout indice $k < j$ $a_j = b_j$ (j peut éventuellement être égal à 1 auquel cas aucun indice k n'est à considérer).

Théorème : Si $(E_i, \leq_i)$ sont des ensembles nœthériens pour $i = 1..n$, alors le produit cartésien $E_1 \times \cdots \times E_n$, muni de l'ordre lexicographique associé, est également nœthérien.

Pour prouver que la fonction suivante termine

```
let rec decode_base (b,l) = match l with
        [] -> 0
 |   n::[] -> 0
 | n::m::r -> if n = 0
        then decode_base ( b , m::r )
        else decode_base ( b , (n-1)::(m+b)::r ) ;;
```

on considère le produit cartésien N^n où n est la longueur de la liste `l`, et on considère le n-uplet formé par la liste donnée en argument de chaque appel récursif (complétée à gauche par des 0 pour les listes plus courtes que `l`). L'appel récursif est alors strictement plus petit que l'appelant pour l'ordre lexicographique, donc on a une suite strictement décroissante, donc la fonction termine.

De manière générale, supposons qu'une fonction récursive `f` soit définie sous la forme

`let rec f` $\vec{X_0}$ `= ...`
`...`
`...` `(f` $\vec{X_1}$`) ...`
`...` `(f` $\vec{X_1}$`) ...`
`...`

où $\vec{X_0}$ est un n-uplet de variables et les $\vec{X_i}$ sont les appels récursifs.
Pour prouver que `f` termine, il suffit de définir une application μ du domaine de définition de la fonction `f` dans un ensemble nœthérien, telle que : pour tout appel récursif avec les arguments $\vec{X_i}$ apparaissant dans la fonction, on a $\mu(\vec{X_0}) < \mu(\vec{X_i})$.