Introduction à la programmation avec CAML

Jean-Michel Hélary
IFSIC

Cours C100
Version 2.2
juin 2001

Table des matières

Avertissement

Ce polycopié présente une introduction aux principes de la programmation, basée sur l'approche dite *fonctionnelle*, et illustrée par l'utilisation du langage de programmation CAML. Comme pour toute utilisation, cela requiert l'apprentissage des éléments de ce langage, afin de comprendre les exemples donnés et de pouvoir écrire ses propres exemples. L'approche décrite dans ce cours devra – c'est absolument indispensable – être complétée par des Travaux Pratiques sur machine. Cela est d'autant plus facile que le langage CAML est distribué gratuitement par INRIA et peut facilement être installé sur de nombreuses plates-formes (dos, Windows 3.xx, Windows95, WindowsNT, Macintosh, Unix, etc.).

Cependant, **ce polycopié n'est pas un manuel d'utilisation du langage CAML**. Notamment, des constructions importantes du langage, telles que les filtres, sont volontairement ignorées. Des manuels du langage CAML existent déjà et sont facilement accessibles, que ce soit sous forme d'ouvrage ([LW]), ou par internet (le site [inria] contient de nombreuses références et informations).

Des compléments sur le langage CAML (extraits de la bibliothèque de base et types construits) sont toutefois donnés, en annexe, à la fin de ce polycopié. Ils sont extraits du manuel en ligne sur Internet, disponible en anglais.

Rennes, juin 1999

© IFSIC 1999

Chapitre 1

Algorithmes, programmes, machines

Ce chapitre constitue une introduction générale à la programmation. Il doit beaucoup aux deux ouvrages [FH] et [AH-DG] auxquels il emprunte une partie de leur chapitre introductif et de leurs exemples.

1.1 Informatique et information

Définition de l'informatique (Larousse) :

> Science du *traitement* automatique et rationnel de l'*information* en tant que support des connaissances et des communications.

Toujours dans ce même Larousse, **définition de l'information** (dans son acceptation informatique) :

> Élément de connaissance susceptible d'être *codé* pour être conservé, *traité* ou communiqué.

Les deux concepts importants apparaissant dans cette définition sont ceux de *codage* et de *traitement*.

1.2 Codage

Le codage d'une information, que ce soit pour la stocker, la traiter ou la transmettre, se fait au moyen de *symboles*: lettres, chiffres, pictogrammes, signaux (lumineux, sonores, électromagnétiques, électroniques, ...), etc.

Exemples:

information	codage	nature du signal
“les voitures passent”	vert	signal lumineux
“un signal d’appel téléphonique”	beep	signal sonore
“Pharmacie”	✙	pictogramme

La plupart du temps, les symboles sont regroupés selon des règles, et on associe un élément de connaissance à un groupe de symboles construit selon ces règles.

Exemples:

information	codage	nature du signal
“l’année courante”	MCMXCXVI	suite de lettres (chiffres romains)
“le caractère ‘+‘”	2B	codage ASCII en hexadécimal
“un pack de lait”	▮▯▮▯▮▯▮▯▮▯▮	code barre

On appelle *langage* un ensemble de symboles ou de groupes de symboles construits selon certaines règles.

On appelle *syntaxe* l’ensemble des règles de construction des groupes de symboles. Par exemple, la syntaxe de la langue française est l’ensemble des règles selon lesquelles on peut grouper les mots. La syntaxe est souvent décrite par une *grammaire*.

La *sémantique* d’un langage exprime la signification associée aux groupes de symboles (correspondance *codage* $\leftrightarrow$ *élément de connaissance*). Mais cette correspondance est souvent floue, puisqu’une même connaissance peut avoir *plusieurs* représentations symboliques, et réciproquement, un même symbole peut correspondre à plusieurs connaissances.

Exemples :

- le nombre entier qui s’énonce “soixante et onze” en français hexagonal peut être codé :
  - 71 (codage décimal)
  - LXXI (codage romain)
  - septante un (codage du langage naturel “français parlé en Belgique ou en Suisse”)
  - 1000111 (codage binaire)
- réciproquement, le groupe de symboles 71 peut représenter:
  - le résultat d’un dénombrement (il y a 71 admis à l’examen de psychologie informatique)
  - une étiquette attribuée à une classe d’objets (la ligne de bus 71 du Syndicat d’économie Mixte des Transports en Commun de l’Agglomération Rennaise dessert la commune de Chevaigné)

- le code téléphonique du département de Haute-Loire
- la notation octale du code ASCII du caractère 9

Seul le "contexte" (ensemble d'information englobant) permet en général de coder/décoder sans ambiguïté une information.

1.3 Algorithmes

Comme il a été vu plus haut, l'informatique s'occupe de "traiter" des informations. Par *traitement* d'une information, on entend *élaboration, à partir d'informations connues (les* données*), d'autres informations (les* résultats*), et ceci par un* agent exécutant. La notion de *traitement* recouvre en fait deux niveaux qu'il convient toujours de bien distinguer : d'une part la *description*, d'autre part l'*exécution*. Ajoutons l'agent exécutant, et nous avons les trois concepts suivants :

description : la *méthode de passage* des données aux résultats, (*modèle d'exécution*) est décrit dans un texte, parfois de manière codée (en soi, c'est aussi une information!),

exécution : une réalisation effective du traitement est mise en œuvre sur des données spécifiques,

agent exécutant : c'est l'entité effectuant une exécution. Cette entité est donc capable de mettre en œuvre la méthode.

Dans le contexte de l'informatique, la description sera souvent exprimée à l'aide d'un *algorithme*, l'exécutant est appelé *processeur* (ou *machine*) et une exécution est appelée *processus*. Cependant, le traitement de l'information n'est pas limité au seul domaine de l'informatique: le code de l'information, la méthode de traitement, peuvent être plus vagues que ce qui est exigé par un traitement automatique, et le processeur n'est pas nécessairement un appareil (ce peut être un humain). Mais il y a lieu d'insister sur le fait que, indépendamment du contexte, les trois concepts mis en évidence ci-dessus doivent toujours être distingués. A cet égard, l'exemple "tarte à la crème" suivant aide à prendre conscience de cette distinction. Considérons le contexte culinaire : on y "fait la cuisine", en réalisant des recettes. Une recette de cuisine est un modèle d'exécution, décrit dans un livre ou sur une fiche; l'agent exécutant est en général un être humain; l'exécution consiste à mettre en œuvre la recette en utilisant des ingrédients concrets. L'élémentaire bon sens permet de ne confondre la recette - qui, en soi, n'est pas nourrissante à moins d'être papivore, - ni avec l'exécutant - qui n'est pas comestible à moins d'être anthropophage - ni avec le processus de préparation et de cuisson. Il est clair que ces trois éléments ne se situent pas sur le même plan!

Un autre point important concerne la *cohérence* entre ces trois éléments : la méthode de passage doit être adaptée à l'exécutant (ce dernier doit "savoir" la mettre en œuvre), et les processus d'exécutions doivent avoir les moyens de se

dérouler, en ayant accès aux *ressources* nécessaires (au minimum, les données spécifiques d'un processus doivent être accessibles par le processeur: le cuisinier doit avoir les ingrédients, ou être capable de déclencher un autre processus pour les acquérir, en envoyant ses enfants faire les courses par exemple).

Dans la suite, nous réserverons le terme de *machine* à l'association d'un agent exécutant (processeur) et d'un ensemble d'opérations "élémentaires" ou "primitives" que l'agent connaît (qu'il sait exécuter).

Concentrons nous plus précisément sur la notion d'*algorithme*, puisque c'est un concept fondamental dans le domaine informatique (mais pas seulement) [1].

Définition : Un algorithme est la description *finie* d'une méthode de résolution d'un problème, à l'aide d'un enchaînement d'opérations primitive.

Précisons: les opérations primitives sont celles que sait réaliser une machine. Un algorithme est donc toujours relatif à une (classe de) machine donnée.

Exemple 1

1. Machine : un(e) élève de l'école primaire - disons CE1 - sachant effectuer les additions à un chiffre (il connaît les *tables d'addition* des entiers de 1 à 9).
2. Traitement : additionner deux nombres entiers. La transformation est alors une *fonction* qui, à deux nombres entiers *donnés* fait correspondre un nombre entier *résultat*.
3. Algorithme : la manière d'effectuer cette opération, décrite comme enchaînement d'additions élémentaires, avec une disposition adéquate des données.

Exemple 2

1. Machine : un processeur muni des opérations suivantes : *addition de deux entiers, multiplication par deux, division par deux, reconnaissance de la parité d'un entier, comparaison avec 0*
2. Traitement : multiplier deux nombres entiers
3. Algorithme : méthode de la multiplication russe

Cet algorithme est basé sur l'égalité de la division par 2 :

$$\forall a \ :: \ a \in \mathbf{N} \ :: \ a = (\frac{a}{2}) \times 2 + a \ \mathbf{mod} \ 2$$

où **mod** désigne le reste. D'où, si a et b sont deux entiers :

$$a \times b = (\frac{a}{2}) \times (b \times 2) + (a \ \mathbf{mod} \ 2) \times b$$

---

1. 

Le terme provient du nom propre **al'Khwarizmi** (l'homme de Kwarazm, actuellement Khiva en Ouzbekhistan), pseudonyme de Abu Ja'far Muhammad, mathématicien persan (790 - 850) – qui fut aussi le premier auteur à mentionner le terme *al-djabr*, devenu algèbre (ilm al-jabr wa'l-mukabala), signifiant la restauration de quelque chose de brisé ou l'agrandissement d'une chose incomplète.

Bien entendu, on sait aussi que, si $a = 0$, alors $a \times b = 0$.

Sachant que a **mod** 2 vaut 1 si a est impair et 0 autrement, on en déduit une description possible de l'algorithme :

calcul du produit de a par b :

- si $a = 0$ le résultat est 0,
- si $a \neq 0$, alors diviser a par 2, multiplier b par deux, *calculer le produit* des deux nombres résultant de ces opérations, et, si a est impair, ajouter b au résultat.

Cette description est *récursive*, puisque la résolution du problème avec les données a et b nécessite, si $a \neq 0$, la résolution du *même* problème sur des *données différentes*, à savoir $\frac{a}{2}$ et $b \times 2$. Nous étudierons très en détail la méthodologie de construction d'algorithmes récursifs et la preuve de leur bon fonctionnement, puisque de tels algorithmes sont très bien adaptés au style de programmation fonctionnelle, notamment. La présentation ci-dessus contient déjà sa propre *preuve*, puisqu'elle est basée sur une propriété mathématiquement établie.

Pour compléter cet exemple, montrons une exécution possible, par un processeur "humain", avec deux données codées sous forme décimale, puis sous forme binaire

a	b	accumulation
171	28	28
85	56	56
42	112	
21	224	224
10	448	
5	896	896
2	1792	
1	3584	3584
	Résultat	4788

a	b	accumulation
10101011	11100	11100
1010101	111000	111000
101010	1110000	
10101	11100000	11100000
1010	111000000	
101	1110000000	1110000000
10	11100000000	
1	111000000000	111000000000
	Résultat	1001010110100

On remarque que l'exécution avec des données codées en binaire est plus simple, puisqu'elle utilise essentiellement des *décalages*. D'où l'importance du codage - voire de la *structuration* - des données pour l'efficacité des processus d'exécution, l'une de nos préoccupations majeures dans les enseignements de programmation.

En conclusion, il y a lieu d'insister sur le fait qu'il faut toujours distinguer les deux points suivants :

- la *description* de l'enchaînement des opérations (algorithme)
- l'*enchaînement* lui-même (processus)

Cette réflexion se fonde sur deux observations illustrées par les exemples "triviaux" ci-dessous :

1. *avance d'un pas et recommence indéfiniment* est une description finie (elle utilise 36 lettres, 5 espaces, 1 apostrophe) d'un processus infini (en

d'autres termes, l'exécution d'un tel algorithme par une machine sachant *avancer d'un pas* durera un temps infini).

2. Un processus utilise des ressources, notamment des *données* qu'il peut acquérir, lorsqu'il en a besoin, dans le monde extérieur; l'algorithme, modèle d'exécution de ce processus, est rédigé dans l'ignorance de ces données. Par exemple, l'algorithme décrit par le texte *ajouter 18,6% au prix hors taxe* est une rédaction qui fait abstraction du prix hors taxe en question.

1.4 Programmes

Un algorithme est décrit à l'aide d'un langage (fondé sur des symboles). Ce langage doit être compréhensible par l'exécutant. Par *compréhensible*, on entend que l'exécutant doit savoir décoder les textes écrits dans ce langage (conformément à sa syntaxe). Ceci implique notamment que l'exécutant doit :

- comprendre les enchaînements de primitives,
- comprendre les primitives elles-mêmes,
- comprendre où et comment acquérir les ressources nécessaires à une exécution, où et comment communiquer les résultats de cette exécution (communication avec le monde extérieur à la machine, ou *interface*)

Les ordinateurs constituent une classe de machines capables de réaliser automatiquement du traitement d'information (leur structure et leur fonctionnement sont détaillés dans d'autres enseignements *initiation système, architecture, etc.*). En particulier, chaque machine "physique" possède un catalogue d'opérations primitives (les instructions du langage machine) qu'elle peut enchaîner séquentiellement, et un organe (l'unité de contrôle) capable de décoder chaque opération primitive. Les langages machine sont en général très frustres, et difficilement compréhensibles par l'être humain. C'est pourquoi on a inventé des langages plus évolués, dits *langages de haut niveau*, dans lesquels les rédacteurs d'algorithmes (souvent des êtres humains) s'expriment plus aisément, et donc de manière plus fiable. Ces langages ne sont donc pas compréhensibles par les machines physiques, mais par des machines *abstraites*. Grossièrement, si $\mathcal{L}$ désigne un langage évolué et $\mathcal{P}$ un langage primitif, une machine abstraite comprenant le langage $\mathcal{L}$ peut être définie sur une machine concrète comprenant le langage $\mathcal{P}$ (un ordinateur et son langage machine, par exemple) comme la conjonction de cette machine concrète et d'un algorithme de traduction du langage $\mathcal{L}$ vers le langage $\mathcal{P}$, cet algorithme étant lui-même écrit dans le langage $\mathcal{P}$.

Définition On appelle *langage de programmation* un langage compréhensible par une machine abstraite définie sur un ordinateur.

Définition On appelle *programme* un algorithme écrit dans un langage de programmation.

Un programme est donc un cas particulier d'algorithme. Il est particulier essentiellement à deux titres :

1. il est exprimé dans un langage de programmation, dont les règles de grammaire sont en général très contraignantes et précises (contrairement à un langage naturel - langue vivante, etc. - un langage de programmation ne supporte aucune ambiguïté),
2. les processus associés supposent l'existence d'une machine capable de comprendre ce langage.

Pour conclure ce chapitre introductif, nous présentons un algorithme célèbre, l'*algorithme d'Euclide*, décrivant une méthode de calcul du plus grand commun diviseur (PGCD) de deux nombres entiers. La date (approximative) de cet algorithme, IIIème siècle av. JC, montre incidemment que la notion d'algorithme n'a pas attendu l'ère de l'informatique pour se manifester [2].

Description en langage naturel (français hexagonal de la fin du XXème siècle):

> *le pgcd des deux nombres a et b est égal à a lorsque ces deux nombres sont égaux. Dans le cas contraire, remplacer le plus grand des deux nombres par leur différence, et recommencer.*

Cette description est suffisamment claire pour un être humain comprenant ce langage naturel, et sachant en outre comparer, soustraire, et décider s'il y a lieu de continuer ou non. La notion de *répétition* (" ... et recommencer") y est toutefois assez intuitive, n'étant pas bien précisé ce qu'il faut recommencer.

Description en langage CAML

```
let rec pgcd = function (a, b) ->
                  if a = b
                     then a
                     else if a>b then pgcd(a-b,b)
                                 else pgcd (a, b-a);;
```

Ici, la répétition est exprimée de manière *récursive*, et l'on indique, pour chacun des cas possibles, l' *expression* dont l'évaluation donne le résultat. De plus, le programme est défini comme étant une fonction, au sens mathématique du terme, qui, à un couple d'arguments a et b, fait correspondre la valeur de l'expression appropriée. C'est à partir du texte et des indications qu'il contient, ici la présence d'opérateurs tels que - ou > applicables, dans le langage CAML,

2. La multiplication russe, vue plus haut, en constitue un autre exemple, puisque, comme indiqué dans [AS], des exemples d'utilisation de cet algorithme ont été trouvés dans les papyrus du Rhind, un des plus vieux documents mathématiques existants, écrit environ 1700 ans av. JC par un scribe égyptien nommé **A'h-Mose**.

uniquement aux nombres entiers, qu'il est possible de déterminer que les arguments comme le résultat de cette fonction sont des nombres entiers (voir au chapitre 2 la notion de *type*). Noter la présence, dans le texte, de symboles n'ajoutant rien à la compréhension de l'algorithme lui-même, mais nécessaires dans le contexte de ce langage pour être conforme à sa syntaxe: par exemple le double point-virgule `;;` qui marque la fin de ce texte, ou encore le *mot-clef* `rec`.

Description en langage Pascal

```
function pgcd(a, b : integer) : integer ;
    var x, y : integer ;
    begin
        x:= a ; y:=b ;
        while x<>y do
              begin
                  if x>y then x:=x-y
                         else y:=y-x
        end;
        pgcd := x
end ;
```

Ici, la répétition est exprimée de manière *itérative*, c'est-à-dire que l'on indique explicitement la condition d'arrêt, les actions à effectuer lors de chaque étape, avec la gestion explicite par le programmeur de résultats de calculs intermédiaires enregistrés dans les *variables* x ou y (cette gestion nécessite la déclaration, la mise à jour, etc.). Là aussi, l'algorithme est défini comme une fonction, dont la **signature** - c'est-à-dire le nombre et le type des arguments ainsi que le type du résultat - sont explicites. La différence de style entre les deux descriptions est importante. La description CAML est basée sur l'expression de ce qu'on veut obtenir, alors que la description Pascal exprime le "comment faire pour obtenir ce qu'on veut". Ces différences entre le style *déclaratif* dans le premier cas et le style *impératif* dans le deuxième cas, feront bien sûr l'objet d'une discussion plus approfondie dans les cours de programmation.

Description en langage d'assemblage, en langage machine Ces descriptions seront faites dans l'enseignement de système. Ce qu'il faut remarquer, c'est essentiellement que les programmes sont alors complètement dépendants de la machine physique. C'est pourquoi on qualifie ces langages de *bas niveau*, ce qui n'a évidemment aucune tonalité péjorative, mais exprime le fait que la machine exécutante est très proche de la *couche physique*.

Chapitre 2

Valeurs et types

Dans la suite de ce cours, les exemples que nous choisirons pour illustrer les concepts seront présentés - sauf mention contraire - dans le formalisme du langage CAML, mais leur signification a souvent une portée plus générale. Commençons par présenter les éléments du système de dialogue CAML ([LW]).

2.1 Comment dialoguer avec une machine CAML

Dans sa forme la plus immédiate, le langage CAML est utilisé au sein d'un système de dialogue avec le programmeur. Un tel mode d'utilisation interactif est particulièrement bien adapté au programmeur souhaitant se familiariser avec les concepts du langage au fur et à mesure de leur acquisition. Un dialogue se déroule au cours d'une *session*, et consiste en une séquence de *phrases* entrées au clavier par le programmeur, chaque phrase étant immédiatement suivie de l'affichage au terminal d'une *réponse* par le système. Chaque interlocuteur doit signaler la fin de chacune de ses "interventions" (phrase ou réponse), signifiant à l'autre que c'est à son tour d'intervenir, et qu'il est prêt à l'"écouter". Le système affiche un caractère d'*invite* au début d'une ligne :

```
#
```

Le programmeur peut alors entrer une phrase au clavier [1], dont il signale la fin par deux points-virgules consécutifs (suivis d'un retour-chariot pour valider la frappe) :

```
# 45;; (* un nombre entier! *)
```

Le système affiche alors sa réponse sur la ligne suivante, suivie d'une nouvelle invite :

1. les portions de phrase délimitées par une paire (* *) ne sont pas prises en compte par le système : il s'agit de *commentaires*.

```
# 45;;
 -:int = 45
#
```

Par convention, les réponses du système (sauf l'invite) seront typographiées en *italique*, pour les distinguer des phrases entrées par le programmeur.

Cette séquence de phrases et réponses s'appelle un *dialogue au niveau principal* (top-level session). Elle est lancée par le programmeur grâce à une commande qui "allume" la machine CAML en mode dialogue (cette commande dépend des installations). De même, c'est le programmeur qui peut terminer le dialogue, en tapant la phrase `quit();;`. Cette phrase termine la session en cours, et "éteint" la machine CAML.

Dans la suite, nous découvrirons les différentes formes de phrase que l'on peut formuler lors d'une session. Signalons que ce mode d'utilisation de CAML n'est pas le seul; en mode dialogue, les phrases peuvent aussi être entrées à partir d'un fichier; on peut aussi compiler des séquences de phrase, afin d'en différer l'exécution (mode *batch*).

2.2 Valeurs

Les entités primitives manipulées par les algorithmes sont des *valeurs*, qui peuvent être dénotées. Une *notation de valeur* est un identificateur prédéfini, associé à la valeur qu'il dénote, et ceci de manière indissoluble. Par exemple :

`123` dénote l'entité dont la valeur est un nombre entier qui correspond à la notation décimale 123,

`'A'` dénote l'entité dont la valeur est le caractère qui correspond à l'écriture latine A,

`"bonjour"` dénote l'entité dont la valeur est la chaîne de caractères délimitée par les deux symboles `"` ,

`+` dénote l'entité fonctionnelle dont la "valeur" est l'opérateur d'addition de deux nombres.

En CAML, on utilise les notations de valeurs suivantes:

- entiers : notation standard (suite de chiffres décimaux, précédée éventuellement du caractère `+` ou `-`),

```
# 45;;
 -:int=45
# -208;;
 -:int=-208
```

- réels : notation standard avec *point décimal* et, éventuellement, *facteur d'échelle* :

```
#  18.045;;
```

```
 -:float=18.045
# 1.76e+3;;
 -:float=1760
# 4.12e-2;;
 -:float=0.0412
```

- caractères : entre deux délimiteurs ‘ (backquote) on trouve soit le signe typographique (‘A‘, ‘9‘), soit la séquence ‘\ *code*‘, où `code` est le code ASCII du caractère (‘\ 013‘ :retour-chariot), soit la séquence ‘\ *lettre*‘ où une lettre remplace le code pour certains caractères spéciaux(‘\ n‘ qui dénote aussi le retour-chariot), etc.

```
# ‘&‘;;
 -:char=‘ &‘
# ‘\ 048‘;; (* code ASCII du caractère ‘0‘ *)
 -:char=‘0‘
# ‘\ n‘ (* désigne le caractère RC - retour chariot *)
 -:char=‘\ n‘
```

- booléens : les deux valeurs notées `true`, `false`

```
# true;;
 -:bool=true
# false;;
 -:bool=false
```

- chaînes de caractères : la séquence de caractères délimitée par une paire de ":

```
#  "a";;
 -:string=”a”
#  "James Bond 007";;
 -:string=”James Bond 007”
#  "";; (* la chaîne vide *)
 -:string=””
```

2.3 Types

On regroupe les valeurs de même nature à l'aide de la notion de *type*. Un type est défini par l'ensemble des valeurs que peuvent prendre les entités y appartenant, et par un ensemble d' opérations applicables à ces valeurs:

type ≡ ({valeurs},{opérations})

Les langages de haut niveau fournissent un certain nombre de *types de base*, appelés aussi types *primitifs*, correspondant aux types de valeurs "usuelles". En général, on trouve :

- le type *logique* ou *booléen*, dont les valeurs servent à *tester* ({*faux, vrai*}, {**non, et, ou**})
- le type *entier*, dont les valeurs servent à *dénombrer* ({un sous-ensemble des entiers}, { opérations arithmétiques entières, opérations de comparaison})
- le type *réel* ou *flottant*, dont les valeurs servent à *mesurer* ({un sous-ensemble des réels},{opérations arithmétiques flottantes, opérations de comparaison})
- le type *caractère*, dont les valeurs servent à *noter* ({ les caractères: lettres, chiffres, ponctuation, etc},{ opérations de comparaison})

De plus, les langages offrent souvent la possibilité de *construire* de nouveaux types, à partir de types déjà construits (notamment des types primitifs), en utilisant des symboles appelés *constructeurs de type*. On trouve, par exemple, les types de *t-uples*, de *listes* – qui sont définis dans la section suivante– ,*fonctionnels* – vus au chapitre 4 –, *produits*, *sommes*, *tableaux*, etc. Enfin chaque type, qu'il soit primitif ou construit, peut parfois être nommé à l'aide d'identificateurs.

2.3.1 Types primitifs en CAML

Le langage CAML définit les types primitifs suivants, avec leurs identificateurs : (voir les exemples de phrase de la section précédente) :

- `bool` pour les booléens
- `int` pour les entiers
- `float` pour les flottants
- `char` pour les caractères
- `string` pour les chaînes

Les opérateurs sur ces types primitifs sont dénotés de la manière suivante, avec les règles de priorité habituelles :

- `bool` : **not** (négation), & (conjonction), **or** (disjonction)
- `int` : `-` (opposé ou soustraction), `+`, `*`, `/` (quotient entier), **mod** (reste).
- `float` : les mêmes (sauf **mod**), mais suivis d'un point : `-.`, `+.`, `*.`, `/.`
- `string` : ^ (concaténation)

En outre, CAML offre des opérateurs primitifs de comparaison "universels", en ce sens qu'ils permettent de comparer deux valeurs d'un même type quelconque (à l'exception toutefois des types fonctionnels) : ce sont l'*égalité* et sa négation, notées respectivement `=` et <>, et les opérateurs d'inégalité`>`, `>=`, `<`, `<=`

```
# 4+3=8;;
 -:bool=false
# 'A'<>'a';;
 -:bool=true
```

Une construction comprenant des notations de valeurs et des opérateurs s'appelle une *expression*. Ces constructions seront détaillées au chapitre 3

Exemples

```
# -(4+5);;
 -:int=-9
# 7/3;;
 -:int=2
# 7 mod 3;;
 -:int=1
# 7.0/.3.0;;
 -:float=2.33333333333
# 4>=5;;
 -:bool=false
# not (4<3);;
 -:bool=true
# "abc"^ "def";;
 -:string="abcdef"
# "abcd"<"abddfg";;
 -:bool=true
```

2.3.2 Exemple de type construit en CAML : les t-uples et les listes

La construction de type permet de définir de nouveaux types à partir de types déjà connus, par exemple à partir des types de base. Dans les phrases du langage, on peut écrire des valeurs d'un type construit en combinant des valeurs des types composants avec des symboles appelés *constructeurs de valeur*. Les types construits sont désignés par des expressions de type dans lesquelles les identificateurs de type sont combinés avec des symboles appelés *constructeurs de type*. En CAML, seuls les *constructeurs de valeur* peuvent apparaître dans les phrases; les constructeurs de type n'apparaissent que dans les réponses de la machine CAML [2]. Dans ce qui suit, nous étudions l'exemple des *t-uples* et des *listes*.

2. le programmeur peut toutefois définir lui-même des identificateurs de *certains* types, appelés *types définis par l'utilisateur (user-defined types)*; ces définitions se font dans des phrases de *définition de type*.

2.3.2.1 les types de t-uples

Une valeur de type *t-uple* est une séquence de plusieurs valeurs, de types respectifs t_1,t_2,...,t_n. Un type de *t-uple* est donc défini par un entier n (le nombre de composants) et les n types composants. Le constructeur de valeur pour les t-uples est le symbole "," (la virgule), et le constructeur de type est le symbole "*" (l'étoile).

```
# 1,2;;
 -:int*int= 1 , 2
```

On peut aussi entourer une valeur de t-uple par des parenthèses pour accroître la lisibilité, mais ce n'est pas obligatoire.

```
# (1,'1',"1");;
 -:int*char*string= 1 , '1' , "1"
```

Il est possible d'écrire des *expressions* t-uples :

```
# (4-3, 5>3);;
 -:int*bool= 1 , true

# (2, 'a');;
 -:int*char=2 , 'a'

# ("je suis "^"venu" , "j'ai "^"vu","j'ai "^"vaincu");;
 -:string*string*string="je suis venu" , "j'ai vu" , "j'ai vaincu"
```

On peut aussi combiner les constructions, comme dans ce triplet de paires :

```
# (('a',97),('b',97+1),('c',97+2));;
 -:(char*int)*(char*int)*(char*int)= ('a',97) , ('b',98) , ('c',99)
```

2.3.2.2 Les types de liste

Une valeur de type *liste* est une séquence d'un nombre *quelconque* de valeurs, toutes d'un *même type* `t`. Un type de liste est donc construit à partir d'un type de base.

Fondamentalement, une valeur de type *liste de* `t` est donc :

- soit la valeur *séquence vide*
- soit un doublet, formé d'une valeur de type `t` et d'une valeur de type *liste de* `t`. La première composante symbolise l'élément de *tête* et la deuxième le *reste* de la liste.

Cette définition est donc récursive, au sens où une valeur de type liste comporte une valeur de même type. Certains langages offrent des outils permettant l'ex-

pression, par le programmeur, de telles définitions (et c'est le cas de CAML). Cet aspect ne pourra être abordé qu'après l'étude de la récursion (chapitre 6).

Le langage CAML fournit en fait deux constructeurs de valeurs de liste, l'un dénotant la liste vide et l'autre permettant de construire une liste par composition. Il fournit en outre une notation de valeur de liste par énumération.

Constructeur de valeur Liste vide : elle est dénotée par le symbole `[]` (une paire de crochets),

Constructeur de valeur par composition Le symbole `::` (une paire de "deux-points") permet de construire une liste à partir d'un élément (qui devient la tête de la liste) et d'une liste (qui devient le reste de la liste),

Énumération Une valeur de type liste peut etre dénotée en énumérant les valeurs de ses éléments, séparées par des points-virgules, et délimitées par des crochets. C'est la notation utilisée par la machine pour afficher une valeur de type liste dans une réponse. Mais une telle notation *n'est pas* un constructeur de valeur.

Le constructeur de type fourni par le langage est dénoté par le mot-clef **list**.

Exemples

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

# "un"::["deux"; "trois"];;
 -:string list = ["un"; "deux"; "trois"]
```

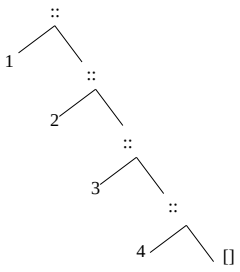
Le constructeur `::` est *associatif à droite* :

```
# 1::2::[3; 4];;
 -:int list = [1; 2; 3; 4]

# 1::(2::[3; 4]);;
 -:int list = [1; 2; 3; 4]
```

Par contre, la construction `(1::2)::[3; 4]` n'a pas de sens, car dans la construction `(1::2)` la notation 2 ne dénote pas une valeur de type liste. De même, la construction `1::[2]::[3; 4]` n'a pas de sens, car dans la construction `[2]::[3; 4]` la notation `[2]` ne dénote pas une valeur de type entier.

La notation $[e_1 \ ; \ e_2 \ ; \ \ldots \ ; e_n]$ est une abréviation de la notation $e_1 :: e_2 :: \ldots :: e_n :: []$. Pour mettre en évidence l'aspect "doublet" on représente parfois les listes graphiquement, comme des peignes dont les dents contiennent les éléments de la liste; ceci fait apparaître la construction de la liste à partir de ses éléments, du constructeur `[]` et de l'utilisation répétée du constructeur



Enfin, il est possible de combiner entre eux les constructeurs de valeur, par exemple en construisant des *listes de t-uples*, des *t-uples comportant des listes*, des *listes de listes*, etc.

```
# [('a',97); ('b',98); ('c',99)];;
 -:(char*int) list = [('a',97); ('b',98); ('c',99)]

# (['a'; 'b'; 'c'],[97; 98; 99]);;
 -:(char list)*(int list) = ['a'; 'b'; 'c'],[97; 98; 99]

# [["Basile"; "Genevieve"; "Odilon"; "Edouard"];
      ["Melaine"; "Tatiana"]; ["Remi"; "Roseline";
      "Prisica"; "Marius"; "sebastien"]];;
 -: string list list = [[ .....
```

La machine interactive CAML permet ainsi de tester immédiatement l'effet des opérateurs sur les valeurs primitives ou construites. Les phrases que nous venons d'écrire sont des combinaisons de valeurs et d'opérateurs. Le chapitre suivant va permettre de généraliser de telles constructions, en introduisant les notions d'*identificateurs* et de *contexte*.

Chapitre 3

Contexte, expressions

Dans ce chapitre, nous nous intéressons aux moyens offerts par les langages de *haut niveau* pour représenter les entités manipulées dans les algorithmes. De tels langages fournissent:

- l'identification des entités et de leur type,
- l'identification des opérations (ou fonctions),
- la gestion d'un contexte.

3.1 Contexte

3.1.1 Identificateurs

Un algorithme manipule des *entités*. Celles-ci ne se limitent pas à une simple notation de valeur, mais peuvent aussi être désignées par un nom. Ainsi, chaque entité possède trois champs : un *identificateur*, un *type* et une *valeur*.

Un identificateur est un symbole formé à l'aide de caractères en obéissant à certaines règles propres à chaque langage de programmation.
Exemples x, y, *toto*, *pgcd*, X_25, ...

Nous avons déjà rencontré, au chapitre précédent, des identificateurs utilisés pour dénoter des valeurs. De tels identificateurs sont évidemment réservés à cet usage, et ne peuvent donc pas être utilisés pour nommer d'autres valeurs. De même, certains identificateurs désignent des opérations bien précises du langage (par exemple l'identificateur **let** qui va être introduit plus loin). On les appelle des *mots-clefs*. Les notations de valeur et les mots-clefs sont appelés *identificateurs prédéfinis*. Les autres sont dits *définis par le programmeur*. Les identificateurs (prédéfinis ou non) constituent les éléments lexicaux de base d'un langage.

En CAML, les identificateurs définis par le programmeur sont des séquences de caractères qui commencent par une lettre, et peuvent comporter des lettres,

des chiffres, et le caractère _ (souligné). *Tout autre caractère y est interdit.* Les minuscules et les majuscules sont distinctes. La longueur d'un identificateur est pratiquement illimitée (mais 80 caractères - longueur d'une ligne d'écran standard - semble être une borne raisonnable!).

3.1.2 Liaison

Lorsqu'une valeur est nommée par un identificateur, on dit qu'il existe une *liaison* entre l'identificateur et la valeur. L'identificateur est alors *lié*. Une telle liaison sera notée, pour les besoins de ce cours, à l'aide du symbole ~ (attention, cette notation n'est pas utilisée dans le langage CAML, mais uniquement dans le "métalangage" utilisé ici pour les explications). Ainsi :

`x ~ 123` signifie que l'identificateur `x` est lié à la valeur `123`. Comme cette notation de valeur est une notation d'entier, on en déduit (c'est implicite) que `x` désigne une valeur de type entier.

3.1.3 Gestion du contexte

Lors d'un processus d'exécution de programme, des liaisons sont établies, rompues, utilisées, etc.

Définition : On appelle *contexte* d'un processus l'ensemble des liaisons existantes à un instant donné du processus.

La notion de contexte est donc relative à un processus donné, et le contexte d'un processus est dynamique en ce sens qu'il peut évoluer pendant le processus, en fonction de l'enchaînement d'opérations décrit dans le programme.

Parmi les opérations susceptibles de modifier le contexte, on peut trouver:

- la *définition* d'une nouvelle liaison,
- la *destruction* d'une liaison existante,
- la *modification de valeur* liée à un identificateur,
- la *ré-initialisation* du contexte.

Certaines sont explicites, c'est-à-dire qu'elles peuvent figurer comme telles dans le texte d'un programme, d'autres sont implicites, c'est-à-dire qu'elles seront réalisées à certains points d'un processus sans pour autant apparaître dans le texte du programme.

La *définition* est en général explicite. Les langages de programmation offrent toujours un *opérateur de définition* permettant d'établir de nouvelles liaisons. De tels opérateurs sont notés et dénommés différemment selon les langages. En CAML, l'opération de définition est notée par le mot-clef **let.**

Exemples :

```
# let x = 123;;
 x:int = 123
```

La liaison entre l'identificateur `x` et la valeur `123` a été établie, comme en témoigne la réponse de la machine : au lieu d'un - (tiret) en premier champ, elle affiche l'identificateur `x`. On dit que cette phrase est une *définition* de l'identificateur `x`.

```
# let un_chiffre='5'
 un_chiffre:char = '5'
# let Cesar = ("je suis "^"venu" , "j'ai "^"vu","j'ai "^"vaincu");;
 Cesar:string*string*string="je suis venu" , "j'ai vu" , "j'ai vaincu"
```

Par contre, les opérations de *destruction* ou de *ré-initialisation* sont presque toujours implicites. Autrement dit, les langages de programmation n'offrent en général pas la possibilité d'indiquer qu'une liaison doit être détruite, ou que le contexte doit être réinitialisé (oubli ou destruction de toutes les liaisons définies depuis le début du processus).

Enfin, les opérations de *modification de valeur*, appelées en général *affectations*, sont explicites dans les langages *impératifs* mais absentes dans les langages *déclaratifs*. Nous reviendrons ultérieurement sur ce point très important. L'approche que nous considérons dans la suite de ce document est *déclarative*, et plus spécifiquement *fonctionnelle*. Retenons pour l'instant que, dans cette approche, l'opération d'affectation n'est pas possible.

L'ordre dans lequel les opérations de modification du contexte sont effectuées peut être significatif. Pour en tenir compte, il y a lieu de considérer un contexte comme une *liste*, c'est-à-dire un ensemble dont les éléments sont énumérés dans un certain ordre. De plus, nous adopterons les règles suivantes, qui permettront ensuite d'expliquer plus facilement un certain nombre de mécanismes mis en œuvre dans les processus (sauf mention contraire, l'ordre adopté va de gauche à droite, c'est-à-dire de la tête vers la queue).

Règle de définition : lors de la définition d'une liaison, la liaison créée est mise *en tête* du contexte.

Exemple : dans le contexte
`[x ~ 123 ; z ~ "bonjour" ; v ~ true]` l'exécution de la définition
let `pi = 3.1416` va donner le nouveau contexte
`[pi ~ 3.1416 ; x ~ 123 ; z ~ "bonjour" ; v ~ ,true]`

Règle de visibilité ou d'homonymie : Si un contexte comporte plusieurs liaisons homonymes, c'est-à-dire de même identificateur, seule la liaison la plus à gauche -c'est-à-dire la plus récente - est visible.

Exemple Dans le contexte

```
[z ~ 45 ; pi ~ 3.1416 ; x ~ 123 ; z ~ "bonjour" ; v ~ true]
```

c'est la liaison `z ~ 45` qui est visible. La liaison `z ~ "bonjour"` existe, mais elle est *masquée*.

Outre les opérations de modification, on trouve aussi les opérations de *consultation* du contexte. Celles-ci permettent de se référer au contexte, et sont utilisées dans les processus d'*évaluation d'expressions*, comme nous allons le voir dans la prochaine section. En général, la donnée d'une consultation est un identificateur, et son résultat est la valeur à laquelle cet identificateur est lié. Si l'identificateur n'apparaît dans aucune des liaisons (absent du contexte), on dit qu'il n'est pas lié, ou encore qu'il est *libre*. Dans ce dernier cas, nous désignerons conventionnellement par le symbole $\perp$ "l'absence de valeur" de l'identificateur. Dans une telle opération, la règle de visibilité ci-dessus est appliquée, en d'autres termes, la recherche dans le contexte est effectuée à partir de la tête jusqu'à ce que l'identificateur donné soit trouvé (succès) ou que tout le contexte ait été parcouru (échec).

3.2 Expressions

Parmi les traitements exprimés dans les programmes, l'un des plus courants est le calcul de nouvelles valeurs à partir de valeurs déjà connues, ce que l'on appelle l'*évaluation*. Les *expressions* constituent, dans un langage, une forme permettant d'exprimer de tels calculs. Une expression est constituée de valeurs et d'opérations permettant de les combiner. Chaque composante de l'expression (valeur ou opération) est désignée par un identificateur (prédéfini ou non).

La *syntaxe* du langage explique comment on peut construire des expressions, c'est-à-dire selon quelles règles les identificateurs composant l'expression peuvent être groupés. La *sémantique* explique comment est menée l'évaluation d'une expression, en fonction du contexte.

3.2.1 Syntaxe

Une expression peut être :
- une notation de valeur,
- un identificateur,
- une application d'entités fonctionnelles,
- une expression construite avec des constructeurs de valeurs.

Parmi les entités fonctionnelles, on trouve les opérateurs primitifs, tels que ceux qui ont été vus au chapitre précédent, mais aussi des *fonctions* définies par le programmeur. Cette dernière notion sera abordée de manière détaillée au chapitre suivant.

Exemples

`4` : notation de valeur entière.

`('a',true)` : notation de valeur de type `char*bool`

`x` : identificateur.

`x*.3.5` : application de l'entité fonctionnelle primitive désignée par `*.` aux deux arguments désignés par `x` (identificateur) et `3.5` (notation de valeur réelle).

`pgcd (n1, n2)` : application de l'entité fonctionnelle désignée `pgcd` à un couple (2-uple) de deux valeurs désignées par `n1` et `n2`

`(z*.3.5, y-3)` : expression construite comme doublet de deux expressions.

Une expression peut être formée de manière *récursive*, autrement dit une expression peut contenir des (sous-)expressions, elles-mêmes décomposables, et ce jusqu'au niveau des entités lexicales de base, à savoir les identificateurs ou les notations de valeur.

Exemple L'expression
`(x-6)*pgcd(y-2,15)-max(y, z+4)` peut être décomposée comme l'indique la figure 3.1:

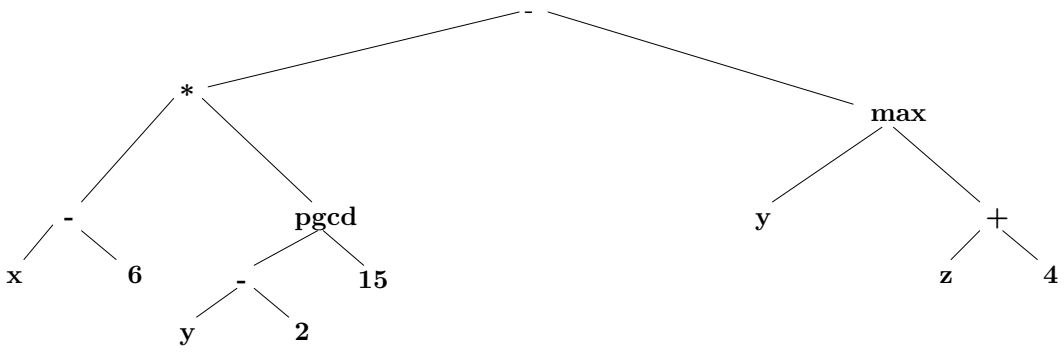


FIG. 3.1 – *expression complexe*

3.2.2 Sémantique

Cette section décrit comment est menée l'évaluation d'une expression. Cette évaluation repose sur un modèle appelé *modèle par substitution*, que l'on trouve dans de nombreux langages, notamment en CAML. L'opération d'évaluation a pour données une *expression* et un *contexte* , et fournit en résultat une *valeur* et le *type* de cette valeur si l'opération réussit, ou $\perp$ si l'opération échoue.

Le processus d'évaluation est mené selon les règles suivantes:

1. si l'expression est une notation de valeur, le résultat est la valeur dénotée et son type (indépendamment du contexte).
2. si l'expression est un identificateur, il y a deux cas:
   (a) l'identificateur est lié dans le contexte: le résultat est alors la valeur et le type figurant dans la *première* liaison présente dans le contexte-c'est-à-dire la liaison résultant de la *plus récente* définition de cet identificateur.
   (b) l'identificateur n'est pas lié dans le contexte: le résultat est $\perp$ (l'évaluation échoue)
3. si l'expression est une application d'entité fonctionnelle, alors:
   (a) chacune des sous-expressions constituant ses arguments est évaluée (l'ordre dans lequel ces expressions sont évaluées n'est en général pas spécifié dans la sémantique).
   (b) si l'une de ces évaluations échoue, le résultat est $\perp$
   (c) si toutes ces évaluations réussissent, la cohérence des types est vérifiée. Cette vérification est établie en vérifiant que les types obtenus lors de l'évaluation des arguments correspondent bien aux types d'arguments prévus dans la définition de la fonction:
      i. si au moins l'un des types d'argument n'est pas conforme au type des arguments de l'entité fonctionnelle, l'évaluation échoue (résultat $\perp$),
      ii. si la cohérence des types est vérifiée, alors les valeurs obtenues lors de l'évaluation des arguments sont *substituées* aux expressions arguments. Le résultat est la valeur obtenue en appliquant l'entité fonctionnelle aux valeurs, et le type de cette valeur.
4. Si l'expression est construite avec des constructeurs de valeurs, alors chaque expression est évaluée selon les règles précédentes.

Exemples

Contexte d'évaluation	Expression	valeur	type	commentaire
tout contexte	4	4	entier	cas 1
[x ˜ 123]	x	123	entier	cas 2.a
[x ˜ 123]	y	$\perp$	$\perp$	cas 2.b
[x ˜ '1' ; y ˜ 140 ; x ˜ 123]	x	'1'	caractère	cas 2.a
[x ˜ '1' ; y ˜ 140 ; x ˜ 123]	$y + z$	$\perp$	$\perp$	cas 2.b pour z, puis 3.b
[x ˜ '1' ; y ˜ 140 ; x ˜ 123]	$y + x$	$\perp$	$\perp$	cas 3.c.i (type de x incorrect)
[a ˜ 3 ; b ˜ 1]	$a + b * 4$	7	entier	cas 3.c.ii, éval. de a et $b * 4$
[a ˜ 3 ; b ˜ 1]	$(a + b) * 4$	16	entier	cas 3.c.ii, éval. de $a + b$ et 4
[a ˜ 3 ; b ˜ 1]	$a + b = 4$	*true*	booléen	cas 3.c.ii, éval. de $a + b$ et 4
[a ˜ 3 ; b ˜ 1]	$a * b = 4$	*false*	booléen	cas 3.c.ii, éval. de $a * b$ 4
tout contexte	(4-3, 5 >3)	(1, *true*)	entier*booléen	cas 4 puis 1
[a ˜ 3 ; b ˜ 2]	$[a + b; a * b; a/b; a$ **mod** $b]$	[5;6;1;1]	liste ent	cas 4 puis 3.c.ii

Bien entendu, les exemples précédents pourraient tous être testés directement à l'aide de la machine CAML interactive, dans la session ci-dessous :

```
# 4;;
 - : int = 4
# let x=123;;
 x : int = 123
# x;;
 -:int = 123
# y;;
 > Toplevel input:
>y;;
> ^
> Variable y is unbound.
# let x='1' and y=140;; (* définition simultanée de deux ident.
                                                dans une même phrase *)
 x :char= '1'
 y :int= 140
# x;;
 - : char = '1'
# y + z;;
 > Toplevel input:
>y+z;;
 >  ^
> Variable z is unbound.
# y+x;;
 Toplevel input:
>y+x;;
>  ^
> Expression of type char
> cannot be used with type int
# let a=3 and b=1;;
 a : int = 3
 b : int = 1
# a+b*4;;
 - : int = 7
# (a+b)*4;;
 - : int = 16
# a + b = 4;;
 - : bool = true
# a*b = 4;;
 - : bool = false
# (4-3, 5>3);;
```

```
 - : int * bool = 1, true
# let b=2;;
 b : int =2
# [a+b;a*b;a/b;a mod b];;
 - : int list = [5; 6; 1; 1]
```

3.3 Phrases CAML

Nous avons déjà rencontré deux sortes de phrases dans le langage CAML :

1. les phrases de définition
2. les phrases d'expression

Les premières permettent de définir une nouvelle liaison. Si leur exécution réussit, la réponse commence par l'identificateur qui vient d'être défini (suivie du type et de la valeur) : le contexte est agrandi avec cette nouvelle liaison. Les secondes, au contraire, n'expriment que l'évaluation d'une expression, relativement au contexte existant. La réponse commence par un - (tiret), indiquant que le résultat de l'évaluation n'a pas été nommé : seuls le type et la valeur du résultat sont affichés. A la fin de l'évaluation, le contexte est le même qu'au début.

Les deux schémas suivants résument les deux formes de phrase, compte-tenu des concepts qui ont déjà été abordés :
phrase expression : *expression* `;;`
phrase définition : **let** *ident_1*`=`*expr_1* **and** *ident_2*`=`*expr_2* **and** `...` **and** *ident_n*`=`*expr_n* `;;`
(si $n = 1$, il s'agit d'une définition simple; si $n > 1$, il s'agit d'une définition simultanée).

Dans le cas d'une définition simultanée, toutes les phrases obtenues en changeant l'ordre des définitions sont équivalentes.

Dans la section suivante, nous allons voir que le contexte peut être modifié durant une évaluation. Toutefois, dans notre modèle d'évaluation, ces modifications seront temporaires, de sorte que les contextes de début et de fin d'évaluation seront les mêmes. Autrement dit, toutes les liaisons crées durant le processus d'évaluation seront détruites avant la fin du processus.

3.4 Définitions locales

3.4.1 Intérêt

Supposons que l'on veuille évaluer l'expression `(4+5*5+9)*(4+5*5+9)*4+(4+5*5+9)+4`. Si l'on fournit cette expression à la machine CAML, l'évaluation va bien être

effectuée, mais de manière inefficace car la sous-expression `4+5*5+9` va être évaluée trois fois :

```
# (4+5*5+9)*(4+5*5+9)*4+(4+5*5+9)+4 ;;
 -:int = 5818
```

Si l'on veut éviter à la machine de refaire plusieurs fois le même travail, il faut *nommer* le résultat de cette sous-expression, autrement dit introduire cette valeur dans le contexte afin de pouvoir la retrouver grâce à son nom.

Une première solution serait la suivante :

```
# let x = 4+5*5+9 ;;
 x : int = 38
# x*x*4+x+4 ;;
 - : int = 5818
```

L'ennui est que la liaison `x ~ 38` a été introduite dans le contexte, et ne peut plus être détruite. A l'issue de l'évaluation, le contexte n'est plus le même. Le langage CAML nous offre alors la possibilité d'effectuer une définition *locale*, ayant pour effet une liaison *temporaire* :

```
# let x = 4+5*5+9 in x*x*4+x+4 ;;
 - : int = 5818
(* la liaison x ~ 38 n'est plus dans le contexte : *)
# x ;;
 > Toplevel input:
>x;;
> ^
> Variable x is unbound.
```

La phrase ci-dessus, bien que commençant par le mot-clef **let**, est une *expression*. Celle-ci comporte une définition locale (de l'identificateur `x`), dont l'effet est temporaire.

Il est évidemment possible de nommer le résultat de l'évaluation: on obtient alors une *définition* :

```
# let y = let x = 4+5*5+9 in x*x*4+x+4 ;;
 y : int = 5818
```

Détaillons l'évolution du contexte lors du processus d'exécution de cette phrase. Sa structure est :

let `y` = **let** `x = expr_x` **in** `expr_y`, où `expr_x` est `4+5*5+9` et `expr_y` est `x*x*4+x+4`.

La liaison `x ~ valeur(expr_x)` sera créée, utilisée dans l'évaluation de `expr_y`

puis supprimée à la fin de cette évaluation.

Contexte initial :

`[ ]` (c'est-à-dire vide)

let `y = ...` introduit une liaison sur `y` et lance le processus d'évaluation de `expr_y`:

`[y ~ ?]`, le ? signifiant "processus d'évaluation en cours". Il s'agit du processus d'évaluation de `expr_y`

`...`**let** `x = 4+5*5+9` **in** `...` introduit une liaison *temporaire* sur `x` et lance le processus d'évaluation de `expr_x` :

`[x ~ ? ; y ~ ?]` le souligné signifie : liaison temporaire

processus d'évaluation de `expr_x = 4+5*5+9` : résultat `38`, lié à `x`

`[ x ~ 38 ; y ~ ?]`

`... x*x*4+x+4` processus d'évaluation de `expr_y =x*x*4+x+4` (dans le contexte courant): résultat `5818`, lié à `y`

`[ x ~ 38 ; y ~ 5818]`

suppression de la liaison temporaire sur `x` :

`[y ~ 5818]` qui est le contexte final.

Essayons :

```
# y;;
 -: int = 5818

# x;;
 > Toplevel input:
>x;;
> ^
> Variable x is unbound.
```

3.4.2 Expressions non terminales

Nous venons de voir qu'une expression `e` peut commencer par une définition locale :

let `id = e`$_1$ **in** `e`$_2$`,` où `e`$_1$ et `e`$_2$ sont des expressions

Si c'est le cas, on dit que l'expression `e` est *non terminale*; elle est alors composée :

- d'une définition locale impliquant un identificateur (`id`) et une expression de définition (`e`$_1$)
- d'une expression (`e`$_2$), dans laquelle on peut utiliser la liaison locale de l'identificateur, et appelée pour cette raison l'*expression cible*.

Dans le cas contraire, l'expression est dite *terminale*.

Exemples `x*x+y*y` est une expression terminale

let `x=4+5*5+9` **in** `x*x*4+x+4` est composée de la définition locale **let** `x=4+5*5+9` et de l'expression cible `x*x*4+x+4`. Le schéma général de la figure 3.2.a décrit la structure des expressions non terminales. La figure 3.2.b illustre l'exemple précédent.

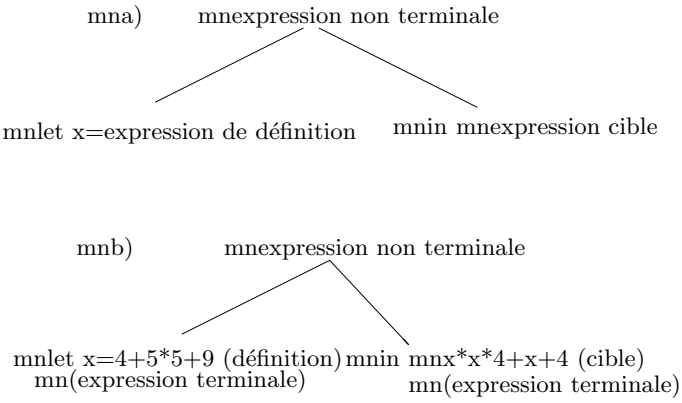


FIG. 3.2 – *Expressions non terminales*

```
# let x= let c=97 in (('a',c),('b',c+1),('c',c+2));;
```
*x:(char*int)*(char*int)*(char*int)= ('a',97) , ('b',98) , ('c',99)*

3.4.3 Définitions locales emboîtées. Structure d'une expression

Il est possible d'introduire plusieurs niveaux de définitions locales, c'est-à-dire des définitions locales dans des expressions servant elles-mêmes à des définitions locales. En effet, chacune des deux expressions composant une expression non terminale (l'expression de définition et l'expression cible) peuvent elles-mêmes être non terminales. Par exemple :

let `x=1` **in** `let y=x+1` **in** `x*x+y*y;;`

est une expression non terminale de la forme **let** `id=`e_1 **in** e_2, avec

e_1 est l'expression `1` (terminale)

e_2 est l'expression non terminale **let** `y=x+1` **in** `x*x+y*y` de la forme **let** `id=`e_{21} **in** e_{22}, où

e_{21} est l'expression `x+1` (terminale)

e_{22} est l'expression `x*x+y*y` (terminale).

L'imbrication des définitions locales traduit une relation *hiérarchique* de dépendance des identificateurs intervenant dans l'expression globale à évaluer. Dans l'exemple précédent, `y` dépend de `x`, et l'expression globale dépend elle-même de `x` et de `y`. On pourra représenter une telle hiérarchie par un schéma arborescent, comme celui de la figure 3.3 qui représente l'exemple ci-dessus.

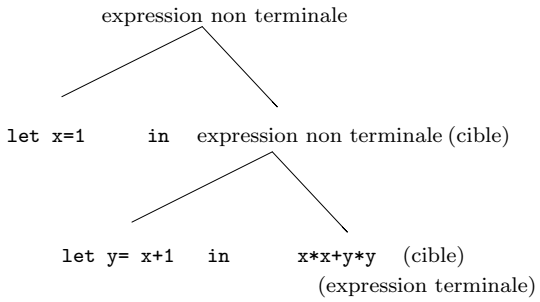


FIG. 3.3 – *Expression avec deux niveaux d'imbrication*

Enfin, toujours avec cet exemple, l'évaluation va être menée comme suit :

Le processus d'évaluation de l'expression se termine lorsque le processus d'évaluation de e_2 est terminé, la valeur de `e` étant celle de e_2. De même, le processus d'évaluation de e_2 se termine lorsque le processus d'évaluation de e_{22} est terminé, la valeur de e_2 étant celle de e_{22} . Lors du processus d'évaluation de `e`, la liaison temporaire `x ~ 1` va être établie et ne sera détruite qu'à la fin du processus d'évaluation de e_2. Pour évaluer e_2, la liaison temporaire `y ~ valeur(x+1)` va être établie (l'évaluation de `x+1` utilise la liaison `x ~ 1`) et ne sera détruite qu'à la fin du processus d'évaluation de e_{22}. Enfin, l'évaluation de e_{22} utilise les deux liaisons temporaires sur `x` et `y`. La liaison sur `y` sera supprimée *avant* la liaison sur `x`.

```
# let x=1 in let y=x+1 in x*x+y*y;;
- : int = 5
```

3.4.4 Niveau et portée des identificateurs

Les imbrications successives de définitions introduisent des *niveaux* d'expressions et de définitions d'identificateurs.

1. Toute phrase comporte une expression, soit seule (si c'est une phrase expression : `#expr;;`) soit nommée (si c'est une phrase de définition : `#let id = expr;;`). L'expression `expr` et, s'il est présent, l'identificateur `id`, sont au niveau 0.
2. Dans une expression non terminale de niveau k, donc de la forme **let** `id=` e_1 **in** e_2, l'identificateur `id` et les deux expressions e_1 et e_2 sont au

niveau $k+1$.

Ces niveaux d'imbrication définissent aussi la *portée* de identificateurs. . *Statiquement*, c'est-à-dire au niveau de texte du programme, la portée est la zone de texte dans laquelle l'identificateur est lié. *Dynamiquement*, c'est-à-dire lors d'un processus d'exécution, la portée d'un identificateur définit l'intervalle de temps pendant lequel la liaison établie lors de sa définition est présente dans le contexte.

1. Le niveau 0 s'appelle encore le *niveau global de la session*, ou *niveau supérieur* (*top-level*). Les identificateurs définis au niveau 0 sont dits *définis globalement* : la liaison établie lors de leur définition reste dans le contexte jusqu'à la fin de la session. *La portée statique des identificateurs définis au niveau 0 commence à la fin de la phrase qui les définit, et se termine lors de la rencontre de la première phrase* `quit();;` *indiquant la fin de la session.*
2. *La portée statique d'un identificateur défini localement au niveau* k $(k \geq 1)$ *est l'expression cible associée à la définition locale de l'identificateur :* **let** `id=`e_1 **in** e_2 indique que la portée statique de `id` est l'expression e_2. Dynamiquement, la liaison reste dans le contexte le temps de l'évaluation de l'expression e_2.

Exemples:

```
# let x=1+4;;  (* x, et son expression de définition 1+4,
                                  sont au niveau 0; la portée de x est glo-
bale *)
 x : int = 5
# let x=1 in x+4;;
  (* expression de niveau 0;
     x, son expression de définition 1 et son expression cible x+4 sont de ni-
veau 1;
     la portée de x est l'expression cible x + 4 *)
 - : int = 5

# let x= let y=1 in y+4;;
(* x et son expression de définition let y=1 in y+4 sont au niveau 0;
   y, son expression de définition 1 et son expression cible y+4 son-
tau niveau 1;
    la portée de x est globale;
    la portée de y est son expression cible y + 4 *)
 x : int = 5

# let x=1 in let y=x+3 in x+y;;
  (* expression de niveau 0;
```

```
     x, son expression de définition 1 et son expression cible let y=x+3 in x+y sont de ni-
veau 1;
     y, son expression de définition x + 3 et son expression cible x +
y sont au niveau 2;
     la portée de x est son expression cible   let y = x + 3  in x + y;
     la portée de  y est son expression cible x + y *)
 -: int = 5
```

Enfin, il faut se rappeler qu'une liaison peut être *masquée* par une liaison homonyme plus récente (règle de visibilité) :

```
# let x=2 in let y=0 in let y=2*x+y in x*x+y*y;;
(* x est au niveau 1, y (lié à 0) au niveau 2,
y (lié à la valeur de 2 * x + y  =  2 * 2 + 0  = 4) au niveau 3;
cette dernière liaison masque la liaison y~0 lors de l'évaluation de x*
x + y * y *)
 -: int = 20

# let x=2 in let y=2*x in let y=0 in x*x+y*y;;
(* le même raisonnement permet de prévoir la réponse ci-dessous : *)
 -: int = 4

# let x = let c=97 in (('a',c),('b',c+1),('c',c+2));;
 x: (char*int)*(char*int)*(char*int))=(('a',97),('b',98),('c',99))
```

3.4.5 Définitions locales et simultanées

Comme l'indique le diagramme syntaxique de la figure 3.6 (page 35), il est possible d'effectuer *simultanément* plusieurs définitions *locales*, de même qu'il est possible d'utiliser des définitions locales dans des définitions simultanées. La structure des expressions vue précédemment doit alors être complétée comme suit : une expression peut être soit *terminale*, soit *non terminale*. Une expression non terminale se présente sous la forme :

let id_1=e_1 **and** id_2=e_2 **and** ... **and** id_n=e_n **in** e_c

Elle contient donc n définitions locales simultanées (c'est-à-dire n identificateurs et n expressions servant à les définir) et *une* expressions cibles, e_c. Les règles de niveaux et portées ne changent pas : si l'expression non terminale est de niveau k, alors les n identificateurs, les n expressions de définition et l' expression cible sont au niveau $k + 1$. La portée statique de chacun des n identificateurs est l'expression cible, et les n liaisons n'existent dans le contexte que durant le processus d'évaluation de l'expression cible. La figure 3.4 est une représentation graphique d'une telle expression.

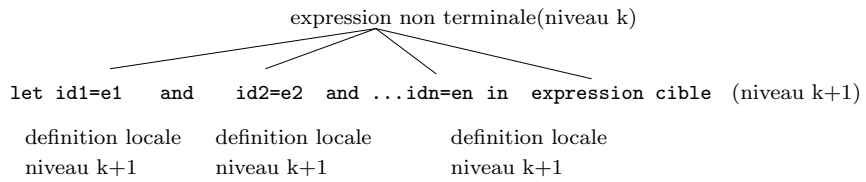


FIG. 3.4 – *Définitions locales simultanées*

Cependant, il y a lieu de faire très attention à la structure d'une telle expression, les erreurs de syntaxe étant fréquentes. De même, il faut gérer correctement les relations de dépendance entre identificateurs. Donnons plusieurs exemples :

```
# let x=1 and y=x+1 ;;
 > Toplevel input:
>let x=1 and y=x+1 ;;
>                  ^
> Variable x is unbound.
```

En effet, les définitions de x et de y étant *simultanées*, ces deux identificateurs sont indépendants. Ainsi, le processus d'évaluation de $x + 1$ opère dans un contexte *ignorant* la liaison $x \tilde{} 1$. La phrase ci-dessus est d'ailleurs équivalente :

```
# let y=x+1 and x=1 ;;
 > Toplevel input:
>let y=x+1 and x=1 ;;
 >          ^
> Variable x is unbound.
```

dont on pouvait prévoir l'échec!

```
# let x=5*5+4*9-6 in x+4 and x-3 ;;
 > Toplevel input:
>let x=5*5+4*9-6 in x+4 and x-3 ;;
>                       ^^^
> Syntax error.
```

En effet, la phrase doit être de la forme **let** `x=`e_1 **in** e_2`;;`. Ici, e_1 `: 5*5+4*9-6` est conforme à la syntaxe des expressions, mais e_2 `: x+4` **and** `x-3` *ne l'est pas*.

```
# let x=1 and y=2+3 in x*x+y*y ;;
 - : int = 26
```

Ici, les deux identificateurs *locaux* `x` et `y` sont définis simultanément (tous deux au niveau 1).

```
# let t= let x=1 in x*x and u= let x=2+3 _f in x*x ;;
 t : int = 1
u : int = 25
```

Ici, les deux définitions *simultanées* de `t` et `u` utilisent chacune une définition locale. `t` et `u` sont définies simultanément au niveau 0, et deux liaisons homonymes, temporaires mais *indépendantes* (de niveau 1) seront créées, utilisées respectivement dans les deux évaluations de `x*x` puis supprimées.

```
# let a= let x=1 and y=2 in x+y
      and b= let x="z" and z="x" in x=z ;;
 a : int = 3
 b : bool = false
```

Ici, `a` et `b` sont définis simultanément (au niveau 0). L'expression définissant `a` utilise les deux définitions locales simultanées (de niveau 1) des identificateurs `x` et `y` (de type `int`), et l'expression définissant `b` utilise les deux définitions locales simultanées (de niveau 1) des identificateurs `x` et `z` (de type `string`).

```
# let Cesar = let ego1 = "je suis " and ego2 = "j'ai " in
                    (ego1^"venu" , ego2^"vu", ego2^"vaincu") ;;
 Cesar:string*string*string="je suis venu" , "j'ai vu" , "j'ai
vaincu"
```

Terminons par un exemple dans lequel nous allons définir une phrase (en français) - c'est-à-dire une valeur de type *string* en CAML- en utilisant des définitions locales dépendantes. Considérons la phrase `"la maman des poissons elle est bien gentille"`. Nous considérons cette phrase comme formée d'un *groupe sujet*, d'un *groupe verbal* et d'un *attribut*. Le groupe sujet est formé d'un *groupe nominal* (`"la maman"`) et d'un *complément* (`"des poissons"`). Le groupe verbal est formé d'un *pronom* (`"elle"`) et d'un *verbe* (`"est"`). Les définitions du *groupe nominal* / et du *complément* doivent rester locales à la définition du groupe sujet (elles n'interviennent pas dans la définition des autres groupes); de même pour les définitions du pronom et du sujet, qui doivent rester locales à celle du groupe verbal.

On a donc la structure *arborescente* de la figure 3.5 :

d'où la définition CAML (^ désigne l'opérateur de concaténation des chaînes) :

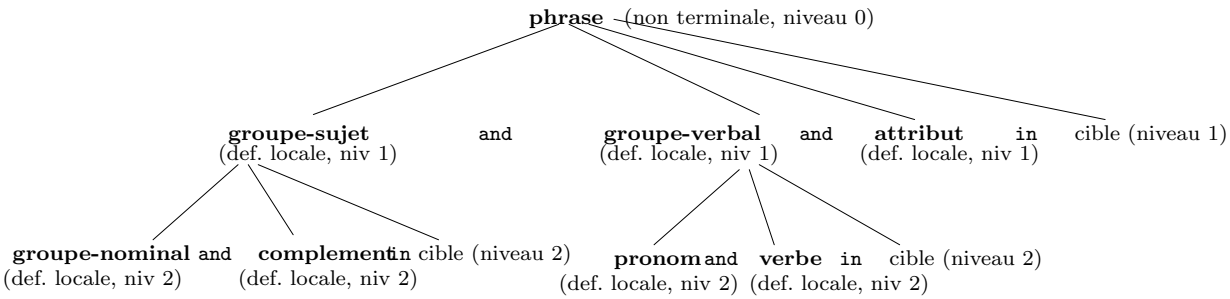


FIG. 3.5 – *Structure de la phrase*

```
# let phrase =
    let groupe_sujet=
        let groupe_nominal= "la maman " and complément="des poissons "
        in groupe_nominal^ complément
    and
       groupe_verbal=
        let pronom="elle "  and verbe="est "
        in pronom^ verbe
    and
       attribut = "bien gentille "
    in groupe_sujet^ groupe_verbal^ attribut ;;
phrase : string = "la maman des poissons elle est bien gentille "
```

Ici, *phrase* est au niveau 0 (comme en atteste la réponse de la machine), *groupe_sujet, groupe_verbal* et *attribut* sont au niveau 1, *groupe_nominal, complément, pronom* et *verbe* sont au niveau 2.

3.5 Expressions conditionnelles

Parmi les expressions que l'on peut construire, celles dont l'évaluation rend une valeur de type *booléen* s'appellent des *conditions*. Par exemple :

```
# 6>3 ;;
- : bool = true

# ('a'='A')  or (((9-4)  mod 2 =0) & (7.5/.3.<=.2.)) ;;
- : bool = false
```

Une expression conditionnelle permet de combiner deux expressions *de même type* à partir d'une *condition*. L'évaluation d'une expression condition-

nelle nécessite l'évaluation de la condition et l'évaluation de l'une des deux expressions.

La syntaxe d'une expression conditionnelle utilise les mots-clefs **if, then, else** :

expr_cond : **if** *condition* **then** *expression_1* **else** *expression_2*

La sémantique en est la suivante : *condition* est évaluée. Si son évaluation échoue, alors l'évaluation de *expr_cond* échoue. Sinon, si la valeur de *condition* est `true`, alors la valeur de *expr_cond* est le résultat de l'évaluation de *expression_1*, si la valeur de *condition* est *false*, alors la valeur de *expr_cond* est le résultat de l'évaluation de *expression_2*.

Exemples

```
(* dans un contexte où  x est lié à une valeur de type entier : *)
# if x>=0 then "positif" else "negatif";;
- : string = "negatif" (*  x est lié à une valeur négative! *)
```

Bien entendu, il est possible de combiner ces constructions entre elles :

```
(* dans un contexte où  x est lié à une valeur de type entier : *)
 # let modulo_10 = let y = if x<0 then (-x) else x in
                 if y<10 then y
                         else if y<100 then y mod 10
                                       else y mod 100;;
modulo_10 : int = 5 (* si  x est lié à la valeur -55 par exemple *)
```

L'intérêt des expressions conditionnelles apparaîtra surtout dans les définitions de fonctions - que nous allons voir au chapitre suivant.

Résumé syntaxique :

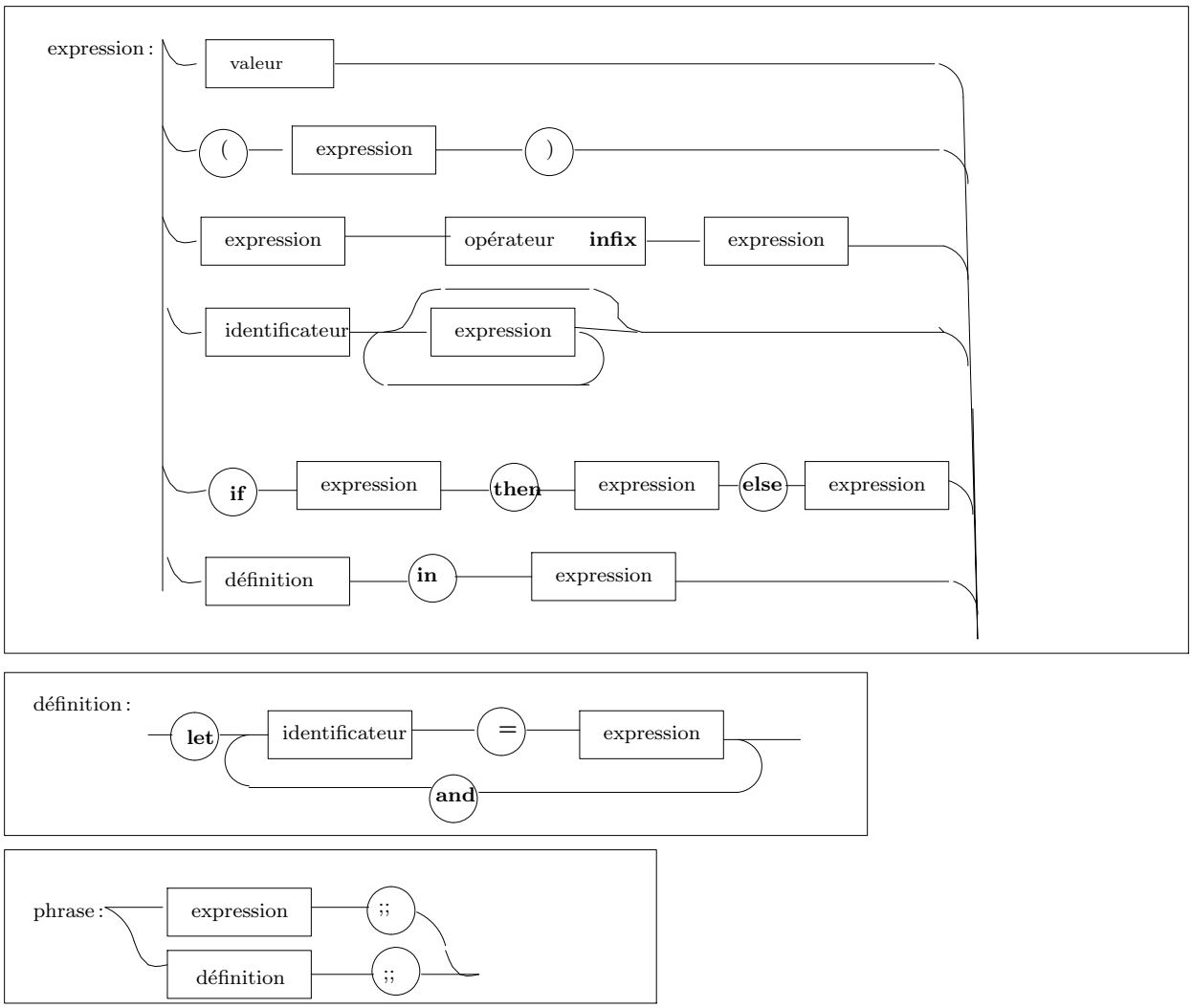


FIG. 3.6 – *Phrases CAML*

Chapitre 4

Fonctions

Dans ce chapitre, un nouveau constructeur de type et son constructeur de valeur associé, les constructeurs *fonctionnels*, sont abordés. Mais, outre l'aspect syntaxique de ces constructeurs, la notion de fonction est fondamentale, puisque c'est à travers elle que l'on aborde vraiment la programmation. En effet, nous allons voir que les fonctions sont des entités exprimant des algorithmes. Une fonction exprime en fait une correspondance entre des données et un résultat, cette correspondance étant définie par une expression dans laquelle les données ne sont pas toutes liées à une valeur du contexte (de telles données s'appellent *arguments formels*). Le *processus d'exécution* de telles expressions, dans un contexte où les arguments formels sont remplacés par des valeurs spécifiques (arguments *effectifs*) constitue alors le processus d'exécution de l'algorithme : aux valeurs des arguments effectifs correspond le résultat de l'évaluation de l'expression définissant la fonction.

Une fonction constitue donc un moyen d'abstraction d'expression, lorsque la valeur de certains composants de l'expression ne doit pas - ou ne peut pas - être spécifiée au moment où l'on conçoit l'expression.

4.1 Fonctions et algorithmes

4.1.1 Un exemple introductif

Prenons comme exemple le "problème" du calcul du prix toutes taxes comprises (prix ttc) d'un article soumis à la tva de taux 18.6%, dont on connaît le prix hors taxe (prix h.t.). La solution à ce problème peut être décrite par l'algorithme suivant : *ajouter le prix h.t. à 0.186 fois le prix h.t.*. Ainsi, l'expression ci-dessous exprime le prix ttc d'un article de prix h.t. égal à FF.167 :

```
# 167. +. 0.186*.167. ;;
 -:float= 198.062
```

Si maintenant on veut le prix ttc d'un article à FF.7.85 h.t., il faut écrire la phrase expression suivante :

```
# 7.85 +. 0.186*.7.85 ;;
 -:float = 9.3101
```

et de même chaque fois que l'on veut faire calculer à la machine CAML un prix ttc. Or, on remarque tout de suite que les deux expressions ci-dessus ne diffèrent que par la valeur du prix h.t., à savoir le flottant 167.0 dans le premier cas, le flottant 7.85 dans le deuxième. En fait, chacune de ces expressions exprime l'*application* de l'algorithme de calcul du prix ttc *à une valeur particulière*. C'est pourquoi la possibilité d'*abstraire* cette valeur particulière du contexte, en exprimant *l'algorithme* lui-même plutôt que son application à une valeur particulière, constitue un outil indispensable pour faire de la machine CAML autre chose qu'une simple calculette[1].

Dans cet exemple, l'algorithme de calcul du prix ttc, décrit informellement plus haut, peut être exprimé, dans le langage CAML, en définissant un nom (`pttc_186` dans notre exemple) suivi d'un identificateur (`x` dans notre exemple). Cet identificateur fait abstraction (ou *représente*) la valeur du prix h.t. `x` s'appelle un *argument formel* de la fonction . La phrase ci-dessous exprime une telle définition :

```
# let pttc_186 x =  x +. 0.186*.x ;;
 pttc_186:float ->  float =<fun>
```

On peut alors *appliquer* cette fonction (c'est-à-dire lancer un processus d'exécution de l'algorithme correspondant) à toute valeur particulière de type `float`, ou même à toute expression pourvu qu'elle soit évaluable en une valeur de type `float` :

```
# pttc_186 167. ;;  (* argument effectif : 167.0 *)
 -:float = 198.062

# pttc_186 167 ;;  (* argument effectif : 167 *)
 > Toplevel input:
>pttc_186 167;;
>          ^^^
> Expression of type int
> cannot be used with type float

# pttc_186 7.85 ;;  (* argument effectif : 7.85 *)
 -:float = 9.3101
```

1. une calculette est une machine munie d'un ensemble de fonctions défini une fois pour toutes.

```
# pttc_186 (34.0/.5.4 +.78.65) ;;  (* argument effectif 34.0/.5.4 +.78.65 *)
 -:float = 100.746307407
```

4.1.2 Définition

Une valeur de type *fonction*, encore appelée *valeur fonctionnelle*, est une règle de correspondance qui associe à chaque valeur d'un type t_1 une valeur d'un type t_2. Une telle valeur exprime donc la notion *mathématique* de fonction. En mathématique on écrira, par exemple :

$$\begin{aligned} \textit{successeur} \quad &: \quad \mathcal{N} \rightarrow \mathcal{N} \\ &\qquad n \mapsto n+1 \end{aligned}$$

et on dira : "*successeur est la fonction qui, à tout nombre entier désigné par n, fait correspondre le nombre entier calculé par l'expression n+1*". La "valeur" de cette fonction est donc la règle de correspondance exprimée par l'écriture $n \mapsto n+1$. L'identificateur *successeur* constitue le *nom* de cette fonction.

En CAML la définition d'une fonction est composée d'un *nom* (identificateur) suivi d'un (ou plusieurs) *arguments*, et d'une *expression de définition*:

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

Si l'on examine la réponse de la machine CAML on constate que :

1. l'indication du type de valeur utilise le *constructeur de type* ->. Ici, `int->int` nous indique qu'il s'agit d'une valeur appartenant au type des fonctions "entiers vers entiers".
2. la machine ne nous indique pas explicitement la *valeur* puisque celle-ci exprime en fait un algorithme (règle de correspondance). L'indication <*fun*> indique seulement qu'il s'agit d'une valeur fonctionnelle.

Les constructions de valeurs et de types peuvent être combinées entre elles, par exemple en construisant un *couple de fonctions* :

```
# (successeur, pttc_186) ;;
 -:(int-> int)*(float-> float) = <fun>, <fun>
```

ou en construisant une fonction à *résultat de type couple* :

```
# let succ_et_pred n = (n + 1, n - 1) ;;
 -:int-> (int*int) = <fun>
```

De même, une fonction peut avoir un résultat de type fonctionnel (voir le paragraphe 4.3),

ou un argument de type fonctionnel :

```
# let accroissement_un f =  (f 1) - (f 0);;
 accroissement_un:(int ->  int) ->  int
```

Cette fonction fait correspondre, à une fonction entiers vers entiers, la différence entre les deux valeurs f(1) et f(0):

```
# accroissement_un successeur;;
 -:int = 1

# let f n = 3*n-2 in accroissement_un f;;
 -:int=3
```

De nombreux exemples de combinaison de types seront ainsi rencontrés dans les chapitres suivants.

4.2 Contexte local d'une fonction

La *syntaxe* générale d'une *définition de fonction* est donc la suivante :

```
let ident_fonction ident_arg =  expression_def
```

où `ident_fonction` et `ident_arg` sont des identificateurs, dénotant respectivement la fonction et *l'argument formel* de cette fonction, et `expression_def` une expression appelée *expression de définition de la fonction*. Une telle définition introduit dans le contexte une *valeur fonctionnelle* ayant un *type fonctionnel*

Contrairement aux types vus jusqu'à présent (types simples, types t-uples), une valeur de type fonctionnel exprime un *algorithme*, et elle contient un contexte, appelé *contexte local de la fonction.* En effet, dans la phrase de définition ci-dessus, l'identificateur `ident_arg` est *lié* dans l'expression de définition, mais son type et sa valeur ne sont pas connus a priori. Tout se passe comme si la définition locale suivante était effectuée (où `?` signifie que la valeur n'est pas connue):

```
let ident_arg=?  in expression_def
```

Ainsi, le contexte local de la valeur fonctionnelle nommée `successeur` et définie par

```
let successeur n =  n + 1;;
```

comporte la liaison locale `n ~?`. Autrement dit, l'argument formel `n` est lié dans l'expression de définition `n+1`. C'est la raison pour laquelle une telle expression est correcte dans une phrase, car dans l'expression de définition `n+1` tous les identificateurs sont liés. Le fait que la valeur de `n` ne soit pas connue n'est

pas important car l'évaluation de la valeur fonctionnelle nommée `successeur` n'implique pas l'évaluation de l'expression de définition `n+1`. Par contre, le fait que `n` soit lié montre que cette dernière expression *pourra* être effectivement évaluée lorsque l'identificateur `n` sera associé à une valeur effective, dans une expression d'application de fonction par exemple (ce point sera revu au chapitre 5, section 5.3)

Remarque: la discussion qui précède montre que l'identification d'un argument formel n'a aucune importance : les deux phrases

```
let successeur n = n + 1;;
```
et
```
let successeur p =  p + 1;;
```
définissent exactement la même valeur fonctionnelle [2]. Dans la première expression, l'argument formel `n` est lié dans l'expression de définition `n + 1`. Dans la deuxième expression, l'argument formel `p` est lié dans l'expression de définition `p + 1`. Par contre, la phrase

```
let f n =  p + 1
```
définit une autre valeur fonctionnelle, car l'identificateur `p` n'est pas lié dans le contexte local `n ~?` de la valeur fonctionnelle. Plusieurs situations sont alors possibles :

1. Si `p` est déjà lié à une valeur de type `int` dans le contexte lorsque la définition de la valeur fonctionnelle `f` est rencontrée, alors `f` dénote une fonction *constante* : la correspondance qui, à toute valeur de `n`, associe la valeur entière de l'expression de définition `p + 1`, indépendante de `n`.
2. Si `p` est déjà liée dans le contexte, mais à une valeur d'un type autre que `int`, l'expression de définition `p+1` est incohérente, donc inévaluable.
3. Si `p` n'est pas liée dans le contexte, l'expression de définition `p+1` n'est pas évaluable.

4.3 Fonctions à plusieurs arguments

4.3.1 Fonctions à deux arguments

Reprenons l'exemple du calcul du prix ttc. Nous venons de voir comment faire abstraction du prix h.t. dans le calcul du prix ttc *lorsque le taux de tva est fixé à la valeur particulière* 18.6%. Supposons maintenant que nous voulions exprimer le même calcul, mais avec un taux de tva différent, par exemple 5.5%.

2. ce phénomène n'est pas unique : on le rencontre aussi, par exemple dans le cadre des fractions où deux notations telles que $\frac{1}{2}$ et $\frac{4}{8}$ dénotent la même valeur.

Nous pouvons définir une autre fonction, à savoir la fonction de calcul du prix ttc des articles soumis à la tva 5.5% :

```
# let pttc_55 x =  x +. 0.055*.x;;
 pttc_55:float ->  float = <fun>
```

Ainsi, à la valeur de type `float 0.186` correspond une fonction de type `float ->float` (identifiée par `pttc_186`), à la valeur de type `float 0.055` correspond une autre fonction de type `float ->float` (identifiée par `pttc_55`). Plus généralement, à chaque valeur `taux` de type `float`, représentant un taux de tva, on peut faire correspondre une fonction de type `float ->float`, exprimant l'algorithme de calcul du prix ttc des articles soumis à la tva de taux `taux`. En faisant abstraction des valeurs particulières de taux de tva, on définit une fonction qui, à une valeur de type `float` (le taux de tva) fait correspondre une fonction de type `float ->float` (la fonction de calcul du prix ttc pour ce taux de tva). Une telle fonction est donc une entité de type `float ->(float ->float)`. Il s'agit là d'un exemple de fonction dont le résultat est de type fonctionnel.

La machine CAML va confirmer cette analyse, comme le montre la définition de la fonction `pttc` ci-dessous :

```
# let pttc taux x =  x +. taux*.x;;
 pttc:float ->  float ->  float = <fun>
```

Dans sa réponse, la machine CAML ne met pas de parenthèses dans l'expression de type de `pttc`: le constructeur de type `->` est *associatif à droite*, ce qui signifie tout simplement que le parenthésage suivant est implicite :

`float ->float ->float` signifie `float ->(float ->float)`

et, plus généralement :

`t1 ->t2 ->...->tn` signifie `t1 ->(t2 ->(...->tn) ...)`.

Les fonctions `pttc_186` ou `pttc_55` définies plus haut peuvent être définies comme des *applications* de la fonction `pttc` à des valeurs *particulières* de l'argument formel `taux` :

```
# let pttc_186 = pttc 0.186;;
 pttc_186:float-> float = <fun>

# pttc_186 167.;;
 -:float = 198.062
```

résultat que l'on pourrait obtenir directement sans passer par l'identificateur intermédiaire `pttc_186` :

```
# pttc 0.186 167.;;  (* signifiant (pttc 0.186) 167.  *)
 -:float = 198.062
```

```
# let pttc_55 = pttc 0.055 ;;
 pttc_55:float -> float = <fun>

# pttc_55 167. ;;
 -:float= 176.185
```

La fonction `pttc` que nous venons de définir est bien une fonction à un argument (de type `float`), dont le résultat est une fonction, elle même à un argument de type `float` et à résultat de type `float`. En fait, la fonction `pttc` peut aussi être vue, conceptuellement, comme une fonction à *deux* arguments de type `float` et à résultat de type `float`, puisque l'expression finale exprimant l'algorithme de calcul : `x +. taux*.x`, dépend de *deux* "paramètres", à savoir le taux de tva `taux` et le prix h.t. `x`. Lorsque l'on "fixe" le *premier* des deux arguments (en l'occurrence le taux de tva `taux`), on obtient une fonction à *un* argument, c'est-à-dire dans laquelle l'expression finale exprimant le résultat ne dépend plus que d'*un* paramètre, le prix hors taxe `x`.

Cette vision de la fonction `pttc` comme fonction à deux arguments est manifeste dans une expression comme `pttc 0.186 167.` où, pour obtenir un résultat du type primitif `float`, il faut former une expression comportant l'identificateur de la fonction suivi de deux expressions (deux arguments effectifs). Toutefois, comme nous venons de le voir, il est possible de ne fournir qu'*un seul* argument effectif, et dans ce cas, le résultat est une fonction.

4.3.2 Fonctions à t arguments ou à un argument t-uple

Les constructions fonctionnelles du paragraphe 4.3 peuvent évidemment être généralisées à un nombre quelconque d'arguments. Par exemple, la fonction ci-dessous délivre le maximum de 3 nombres réels, à partir du maximum de deux nombres réels :

```
# let max2 x y =   if
 x>=.y  then x  else y ;;
 max2: float -> float -> float = <fun>

# let max3 x y z = max2 (max2 x y) z ;;
 max3:float -> float -> float -> float
```

et on pourrait tout aussi bien obtenir le maximum de 4 nombres :

```
# let max4  x y  z  t =  max2 (max2 x y) (max2 z t) ;;
 max4:float -> float -> float -> float -> float = <fun>
```

Une autre manière de définir les fonctions `max2, max3, max4` serait de les considérer comme des fonctions à un seul argument, de type respectif *couple* (2-uple), *triplet* (3-uple), *quadruplet* (4-uple) :

```
# let maxi2 (x, y) =  if x>=.y then
 x else y ;;
 maxi2:(float*float) ->  float =<fun>

# let maxi3 (x, y, z) =  maxi2 ((maxi2(x, y), z) ;;
 maxi3:(float*float*float) ->  float

# let maxi4  (x, y, z, t) = maxi2((maxi2(x,y)), maxi2(z,t)) ;;
 maxi4:(float*float*float*float) ->  float
```

La différence entre les deux fonctions `max2` et `maxi2`, par exemple, ne réside pas dans l'algorithme lui-même (l'expression de définition est identique) mais *essentiellement dans le typage*. La première, `max2`, est une fonction à un argument de type `float` et à résultat *fonctionnel* - pouvant aussi être vue comme une fonction à *deux* arguments de type `float` et à résultat de type `float`, tandis que la seconde, `maxi2`, est une fonction à *un seul* argument de type couple `float*float`, et à résultat non fonctionnel. Cette différence de typage se fait évidemment sentir dans la forme des expressions utilisant ces fonctions, comme on le voit en comparant les expressions de définition des deux fonctions `max3` et `maxi3` ou `max4` et `maxi4`.

Nous n'insisterons pas ici sur les subtilités "conceptuelles" de ces deux formes, signalant seulement que l'on peut facilement définir l'une des deux fonctions à partir de l'autre :

```
(* max2 étant définie : *)
# let maxi2 (x, y) =  max2 x y ;;
 maxi2:(float*float) ->  float = <fun>
```

```
(* maxi2 étant définie : *)
# let max2 x y =  maxi2(x, y) ;;
 max2:float ->  float ->  float = <fun>
```

Pour la "culture", indiquons que la forme à résultat fonctionnel (comme `max2`) est appelée la *forme curryfiée* de la forme à résultat non fonctionnel (comme `maxi2`), l'adjectif *curryfié* étant dérivé du nom du logicien Haskell CURRY, qui a étudié l'équivalence de ces formes fonctionnelles.

Nous avons finalement le schéma syntaxique suivant (figure 4.1 complétant la partie *définition* du schéma de la figure 3.6.

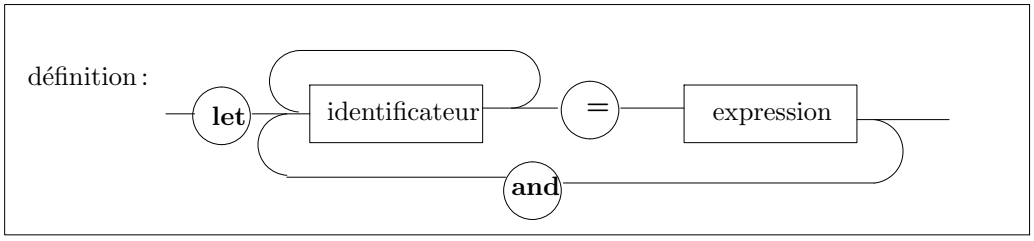


FIG. 4.1 – *Phrase définition*

4.4 Valeurs fonctionnelles anonymes

Aux paragraphes précédents, nous avons défini des valeurs fonctionnelles *nommées*, avec une syntaxe de la forme :

let *ident arg_1 arg_2 ... arg_n = expression*

Il existe aussi un *constructeur de valeur*, permettant de dénoter directement une valeur fonctionnelle (c'est à dire sans lui donner de nom). Nous avons déjà signalé cette possibilité pour d'autres types, par exemple les types t-uples :

`(1,'1',''1'')` est une notation de valeur 3-uple (de type `int*char*string`), qui utilise des notations de valeur de base et le constructeur de valeur `,`.

`[1; 2; 3]` est une notation de valeur de liste (de type `int list`), qui utilise des notations de valeur de base et les constructeurs de valeur `[];`. Dans ces deux exemples, les valeurs ainsi construites sont anonymes.

En ce qui concerne les valeurs fonctionnelles, le constructeur de valeur s'appelle **function** et utilise le symbole ->:

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

Si l'on examine la réponse de la machine CAML on constate qu'il n'y a pas de nom (la phrase est une expression, composée d'une simple valeur).

Une valeur fonctionnelle à n arguments pourra alors être dénotée comme une valeur fonctionnelle à un argument, et dont l'argument est lui-même une valeur fonctionnelle à $n-1$ arguments :

```
# function x ->   function y ->
      if x>=.y  then  x  else y;;
 - : float ->  float ->  float = <fun>
```

L'utilisation de valeurs fonctionnelles anonymes peut s'avérer utile lorsque l'on veut utiliser une telle valeur comme argument effectif dans une expression d'application de fonction ayant des arguments de type fonctionnel. Par exemple, la fonction `map` (prédéfinie) prend en argument une fonction `f` et une liste `l`, et rend la liste obtenue en appliquant la fonction `f` à tous les éléments de la liste `l`. Un exemple d'utilisation de la fonction `map` est le suivant : produire, à partir d'une liste de chaînes de caractères non vides, la liste des premiers caractères de chaque chaîne. La fonction `f` est alors la fonction qui, à une chaîne non vide `ch` fait correspondre le premier caractère de cette chaîne. Si une telle fonction n'existe pas encore dans le contexte, on peut utiliser directement la valeur fonctionnelle correspondante, ce qui donne l'expression:

```
map (function ch ->nth_char ch 0) l
```

Exemple:

```
let premier_char l = map (function ch -> nth_char ch 0) l;;
 premier_char: string list -> char list = <fun>

premier_char [``Diplome''; ``Enseignement''; ``Superieur''; ``Specialise''];;
 - : char list = [`D`; `E`; `S`; `S`]
```

La notation de valeur fonctionnelle peut être utilisée pour définir une fonction (nommée), selon la syntaxe standard d'une phrase de définition :

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

Mais il est plus simple d'utiliser la syntaxe déjà introduite au début de ce chapitre, qui constitue en fait un "raccourci" d'écriture :

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

Cette simplification est encore plus manifeste pour les fonctions à "plusieurs" arguments :

```
#  let pttc = function taux -> function x -> x +. taux*.x;;
 pttc:float ->  float ->  float =<fun>
```

que l'on aurait pu aussi bien définir par la forme syntaxique "mixte" :

```
#  let pttc taux =  function x ->  x +. taux*.x;;
 pttc:float ->  float ->  float =<fun>
```

ou par la forme syntaxique concise:

```
# let pttc taux x =  x +. taux*.x;;
 pttc:float -> float -> float =<fun>
```

4.5 La construction match ... with : définitions par cas

4.5.1 Un exemple

Nous avons vu plusieurs exemples de fonctions dont l'expression de définition est conditionnelle. Une expression conditionnelle est en fait un cas particulier d'expression plus générale, appelée expression filtrée. Prenons tout de suite un exemple : soit la fonction `litt_of_num` qui, étant donné un entier (compris entre 1 et 12), délivre la chaîne exprimant le mois correspondant, en toutes lettres. Une première solution aurait l'allure suivante :

```
let litt_of_num n =
    if n=1  then "janvier"
    else if n=2  then "fevrier"
    else if n=3  then "mars"
      ...
    else if n=12  then "decembre"
    else "pas de treizième mois...";;
```

La construction **match ... with** permet une définition plus concise, analogue à la construction *par cas* que l'on trouve dans beaucoup de langages de programmation :

```
let litt_of_num n =  match n  with
   1 -> "janvier"
  | 2 -> "fevrier"
  | 3 -> "mars"
  | 4 -> "avril"
  | 5 -> "mai"
  | 6 -> "juin"
  | 7 -> "juillet"
  | 8 -> "aout"
  | 9 -> "septembre"
  | 10-> "octobre"
  | 11-> "novembre"
  | 12-> "decembre"
```

```
  | _-> "pas de treizieme mois";;
litt_of_num:int -> string = <fun>
```

La sémantique de la construction **match ... with** est simple : les valeurs possibles de l'argument entier de la fonction sont "filtrées" en fonction d'une séquence de valeurs possibles. Chaque cas est séparé du suivant par le symbole `|` (alternative). Dans l'exemple précédent, les valeurs possibles sont :

1. les *notations de valeurs entières* (constantes 1, 2, ..., 12),
2. le symbole _ (souligné)

Les premières ne "filtrent" que la valeur qu'elles dénotent (1 ne filtre que la valeur 1), tandis que le symbole _ filtre n'importe quelle valeur. Ainsi, la définition de la fonction s'interprète comme suit : lors de l'évaluation de `litt_of_num n`, si `n` est compris entre 1 et 12, la constante dénotant la valeur `n`) permet de trouver le cas correspondant. Sinon, aucun des 12 premiers cas ne convient, et on arrive au dernier (symbole _). Celui-ci convient, puisqu'il filtre tout. Cela correspond donc au **else** final, qui exprime le *"dans tous les autres cas..."*.

Souvent, les définitions par cas sont plus claires, elles peuvent aussi être plus concises. En fait, la notion de filtre ne se limite pas à une simple définition par cas avec des valeurs de type simple. C'est une notion beaucoup plus riche, mais que l'on ne retrouve pas dans les autres langages qui seront étudiés dans nos cours de programmation (pascal, Eiffel, C, etc.). Pour cette raison, nous limiterons ici son utilisation à des cas simples comme dans l'exemple ci-dessus. Les lecteurs intéressés peuvent se reporter aux compléments sur le langage CAML, présentés en Annexe (voir A.1pour une présentation plus complète de la notion de filtre).

Chapitre 5

Typage et évaluation

5.1 Expressions "application de fonctions"

Nous avons vu, au chapitre 3, la notion d'expression et, parmi les expressions, celles qui sont de la forme *application d'entités fonctionnelles*. Mais, parmi ces dernières, seules des expressions contenant des entités fonctionnelles de nature "opérateurs" :*arithmétiques* (`+, -, *, /`) , de *comparaison* (`<, >, <=, >=, =, <>`), *logiques* (`not, &, or`), de *concaténation* (`^`), etc. ont été définies.

Par exemple, une expression telle que `5+3` est une *application de l'entité fonctionnelle* `+` *au x deux expressions* `5` et `3`. Ces deux expressions constituent les *arguments effectifs* de la fonction `+` dans l'expression `5+3`. Les expressions d'application de telles entités fonctionnelles obéissent à une syntaxe particulière, puisque les *arguments effectifs* des opérateurs figurent de part et d'autre de l'identificateur d'opérateur : on dit alors qu'il s'agit d'une *notation infixe* ou *symétrique* (voir troisième ligne du schéma syntaxique de la figure 3.6).

Les fonctions qui ne rentrent pas dans la catégorie des "opérateurs" peuvent aussi être utilisées dans des expressions de nature *application d'entités fonctionnelles*. Dans ce cas, la syntaxe correspond à la notation *préfixe*, dans laquelle la fonction appliquée (dénotée directement par sa valeur ou par son identificateur) précède son (ou ses) argument(s) effectif(s). La phrase ci-dessous, par exemple, est une expression de cette nature :

```
# ( function n ->  n+1) 4 ; ;
 - :int=5
```

Il s'agit de l'expression appliquant la valeur fonctionnelle **function** `n ->n+1` à l'expression `4` ou encore, si la valeur fonctionnelle a été nommée `successeur` :

```
# successeur 4 ; ;
 - :int=5
```

Dans chacun de ces deux exemples, l'*argument effectif* de la fonction dans l'expression est l'expression `4`. Mais, comme l'indique le schéma syntaxique des expressions (quatrième ligne de la figure 3.6) un argument effectif de fonction dans une expression peut être une expression de n'importe quelle nature. Considérons par exemple, la phrase ci- dessous :

```
# successeur (( function x ->  x*x) 3) ; ;
 - :int = 10
```

C'est une expression appliquant la fonction `successeur`, déjà définie, à l'argument effectif d'expression `(function x ->x*x) 3` qui, elle-même, est l'expression d'application de la fonction (anonyme) **function** `x ->x*x` à l'argument effectif d'expression `3`.

Cette dernière fonction aurait pu être nommée, soit par une définition au niveau 0, soit par une définition locale comme ci- dessous :

```
#  let carre x = x*x  in successeur(carre 3) ; ;
 - :int=10
```

Notons que tout opérateur peut être transformé en une fonction "préfixe" grâce au mot-clef **prefix**:

```
# +;;
 Syntax error (* la machine voit cette phrase comme une expression mal formée *)

#  prefix +;;
 - :int -> int -> int = <fun>
# 1+1 ;;
 - :int = 2
# prefix + 1 1 ;;
 - :int = 2
```

Les deux sections qui suivent rappellent et complètent les règles de typage et le fonctionnement des processus d'évaluation des expressions.

5.2 Typage des expressions

Dans ce paragraphe, nous rappelons et complétons les règles permettant de déterminer la *cohérence* et le *type* d'une expression, déjà abordées, au chapitre 3. Rappelons que tous les identificateurs intervenant dans une expression doivent être présents dans le contexte, avec une *valeur* et un *type* (*a priori* inconnus dans le cas d'un argument formel).

5.2.1 Expression dont tous les composants ont un type connu

- L'expression est une *valeur* : son type est celui auquel appartient la valeur,
- L'expression est un *identificateur* lié dans le contexte : son type est celui indiqué dans la première liaison de cet identificateur trouvée dans le contexte,
- L'expression est une application de fonction : soit `f e1 e2 ...ep` une expression, où `f` est une expression de type t_1 -> t_2 -> ... -> t_n -> t_{res}, e_1 une expression de type t_1, ...,e_p une expression de type t_p, et $p \leq n$. Alors, l'expression `f e1 e2 ...ep` est de type t_{p+1} ->...-> t_n -> t_{res}.
- L'expression est conditionnelle : **if** *expr* **then** *e1* **else** *e2*; si *expr* a le type *booléen* et les deux expressions *e1* et *e2* ont le *même* type *t* alors l'expression a le type *t*
- dans les autres cas, il y a erreur de typage (signalée par le compilateur)

Exemples La fonction *char_for_read : char ->string* est prédéfinie et transforme un caractère en la chaîne constituée de ce caractère. L'expression *char_for_read 'a'* est l'application de l'expression *char_for_read* de type *char ->string* à l'expression 'a', de type *char*. Elle est donc de type *string* :

```
# char_for_read 'a' ;;
 -:string = "a"
```

Nous avons déjà rencontré, au chapitre précédent, les deux fonctions prédéfinies *string_length: string ->int* et *nth_char: string ->int ->char*.

Déterminons le type de l'expression

let *ch="azerty"* **in** *char_for_read (nth_char ch (string_length ch -1))*. Elle est de la forme

let *ch = "azerty"* **in** *e1*, avec

e1 = char_for_read e2,

e2 = nth_char e3 e4,

e3 = ch,

e4=string_length e3 -1.

Comme *ch* est lié à la valeur *"azerty"*, on a :

e3 : string puis *string_length : string ->int, e3 : string* d'où

e4 = string_length e3 - 1 : int; comme

nth_char: string ->int ->char, e3: string, e4: int on a *e2 = nth_char e3 e4 : char*; enfin, comme

char_for_read : char ->string, e2 : char, on a *e1 = char_for_read e2 : string*.

```
# let ch="azerty" in char_for_read(nth_char ch (string_length ch -1));;
 -:string : "y"
```

5.2.2 Typage d'une valeur fonctionnelle : synthèse de type

Lorsqu'on exprime une valeur fonctionnelle, l'expression fournie à la machine n'indique nulle part quel est le type des arguments ni le type du résultat de la fonction. Le mécanisme décrit à la section précédente ne s'applique donc plus, puisque certains composants de l'expression, à savoir les identificateurs d'arguments formels, ont un type *a priori* inconnu. Dans ce cas, c'est la machine CAML qui détermine le type de la valeur fonctionnelle ainsi exprimée.

Considérons par exemple l'expression de définition de la fonction `max2`, donnée au chapitre précédent :

let `max2 x y =` **if** `x>=.y` **then** `x` **else** `y;;`

La phrase de définition ne comporte aucune indication explicite sur le *type* des deux arguments, ni sur celui du résultat. Or la machine nous fournit le type fonctionnel de `max2` dans sa réponse, en l'occurrence le fait que les deux arguments doivent être de type `float` et que le résultat est de type `float`. La machine CAML a donc effectué un *calcul de type*, appelé encore *synthèse de type*.

Sans entrer dans les détails, indiquons que l'algorithme de synthèse de type utilisé par la machine est basé sur les règles de typage vues à la section précédente. C'est à partir de l'expression de définition et des informations de types déjà présentes dans le contexte que la machine CAML essaye de déterminer le type que doit avoir chaque argument formel pour que cette expression soit cohérente. L'expression de définition de la fonction `max2` contient les deux identificateurs `x` et `y`, liés aux deux arguments formels et dont le type est *a priori* inconnu; l'expression contient aussi la valeur fonctionnelle `>=.`, connue de la machine CAML comme opérateur infixe s'appliquant à deux arguments de type `float`. Par conséquent, la cohérence de l'expression de définition contraint `x` et `y` à être de type `float`. De plus, cette contrainte n'est contredite par aucune autre dans l'expression de définition. Enfin, le résultat de la fonction étant soit `x` soit `y`, il n'y a pas de conflit puisque tous deux sont de même type (`float`). Ainsi, la machine peut conclure et fournir sa réponse :

max2: float ->float ->float = <fun>

Donnons d'autres exemples. La fonction prédéfinie

`string_length: string ->int` délivre la longueur de son argument, et la fonction prédéfinie

`nth_char: string ->int ->char` est telle que `nth_char ch n =` *caractère de rang* `n` *de la chaîne* `ch` (le premier est de rang 0) :

```
# string_length "bonjour";;
 - : int = 7

# nth_char "bonjour" 1;;
 - : char = 'o'
```

(les deux identificateurs `string_length` et `nth_char` sont automatiquement introduits dans le contexte à l'ouverture d'une session).

Définissons une fonction qui délivre le dernier caractère d'une chaîne non vide. Puisque le premier caractère est de rang 0, le dernier est de rang `l-1`, où `l` est la longueur de la chaîne. La première version donnée ci-dessous ne fait aucun test sur la *vacuité* de la chaîne, cette précondition étant supposée vérifiée.

```
# let dernier ch = nth_char ch (string_length ch -1);;
 dernier:string -> char = <fun>
```

Détaillons le calcul des types. La valeur `dernier` est suivie d'*un* argument formel, elle est donc de type `X ->Y`, où `X` et `Y` sont des *inconnues de types*. Les équations suivantes doivent être vérifiées :

(1) `ch : X`

(2) `nth_char ch (string_length ch -1) : Y`

Examen de l'expression de définition: le contexte indique `nth_char: string ->int ->char`. L'identificateur `nth_char` est suivi de l'expression réduite à l'identificateur `ch`, puis de l'expression `string_length ch -1`. Par conséquent, la cohérence de l'expression n'est possible que si les deux équations suivantes sont vérifiées :

(3) `ch : string`

(4) `string_length ch -1 : int`

(1) et (3) impliquent `X = string`. De plus, le contexte indique `string_length : string ->int` ce qui, avec l'équation (3) implique `string_length ch : int` d'où `string_length ch -1 : int`; l'équation (4) est donc vérifiée. Reportant ces informations dans l'équation (2), on trouve que `nth_char ch (string_length ch -1) : char =Y`. La détermination des types `X` et `Y` est donc possible et complète, d'où la réponse de la machine : *dernier : string ->char*.

5.2.3 Polymorphisme de type

Certaines expressions de définitions de fonctions sont écrites de telle sorte que leur type ne peut pas être déterminé par le mécanisme de synthèse de type. Considérons par exemple le cas de la fonction *identité* qui, à toute valeur fait correspondre la même valeur:

```
# let identite x = x;;
```

identite: quelle va être la réponse ?

L'algorithme de synthèse de type pose que :

`identité : X ->Y`

`x : X`

`x : Y`

La seule contrainte qui en résulte est donc `X = Y`, ce qui est insuffisant pour déterminer `X` (ou `Y`). La machine va donc répondre en notant le type indéterminé `'a`:

identite : 'a -> 'a = <fun>

On dit que `'a` est un *paramètre de type*, et que la fonction `identite` est de *type fonctionnel polymorphe*.

Essayons maintenant cette fonction avec différents types d'arguments:

```
# identite 3;;
- : int = 3   (*  'a est spécialisé par  int *)

# identite true;;
- : bool = true   (*  'a est spécialisé par  bool *)

# identite "identite";;
- : string = "identite"   (*  'a est spécialisé par  string *)

# identite nth_char;;
- : string -> int -> char = <fun>   (*  'a  est spécialisé par le type fonctionnel
                                        string -> int -> char *)
```

Dans la suite, notamment avec les types construits *t-uples* et *listes* nous rencontrerons d'autres fonctions polymorphes. Par exemple, les deux fonctions suivantes (prédéfinies) permettant d'extraire respectivement la première et la seconde composante d'un doublet, sont polymorphes :

```
#  let fst (x,y) = x;;
fst : 'a -> 'b -> 'a = <fun>

#  let snd (x,y) = y;;
snd : 'a -> 'b -> 'b = <fun>

# fst ('a', true);;
- : char = 'a'

# snd ("hello", ["bonjour"; "ciao"]);;
- : string list = ["bonjour"; "ciao"]
```

5.2.4 Première approche des mécanismes d'exceptions

Reprenons l'exemple de la fonction *dernier*, étudié ci-dessus (§5.2.2). Supposons maintenant que l'on veuille écrire une version contrôlant la vacuité de la chaîne *ch*, *dernier* étant déjà définie comme nous venons de le faire :

```
# let dernier_avec_controle ch =
    if ch <>"" then dernier ch
               else "erreur : chaine vide";;
 > Toplevel input:
>   if ch<>"" then dernier ch

>
 ^^^^^^^
 > Expression of type string - > char
 > cannot be used with type string - > string
```

Ici, le typage de l'expression de définition – et donc de l'identificateur *dernier_avec_controle* – échoue. En effet, *dernier_avec_controle : X ->Y*, avec les deux équations :

ch : X

expr_def : Y, où *expr_def* est l'expression de définition.

Dans cette expression conditionnelle, la clause **then** est formée de l'expression *dernier ch*; le contexte indique que *dernier : string ->char*, et donc, *ch : string*, d'où *X = string*, et *dernier ch : char*. Ceci implique

X = string

Y = char (5)

La clause **else** est formée de l'expression *"erreur : chaine vide"* qui est une notation de valeur de type *string*. Ceci implique

Y = string (6)

Les deux équations (5) et (6) sont donc incompatibles, et le système d'équations de type n'a pas de solution.

Pour remédier à cette situation, il faudrait donc que l'expression dans la clause **else** soit aussi de type *char*. Mais alors, quelle valeur de caractère délivrer? Logiquement, aucune valeur ne convient, car il est *absurde* de vouloir définir le *dernier caractère d'une chaîne n'ayant aucun caractère*! On pourrait convenir à l'avance d'une valeur "bidon" qui serait rendue lorsque la chaîne est vide, comme dans l'exemple ci-dessous:

```
# let dernier_avec_controle ch =
    if ch <>"" then dernier ch
               else '$';;
 dernier_avec_controle:char -> char=<fun>
```

Mais c'est une *très mauvaise solution*, à plusieurs titres :

1. elle oblige tous les utilisateurs potentiels de cette fonction à se mettre d'accord sur une telle valeur (ici *'$'*)
2. elle oblige ensuite les utilisateurs à programmer un test pour capter le cas où l'application de la fonction donnerait la valeur "bidon" *'$'*
3. enfin, les deux expressions `dernier ""` et `dernier "cout : 25$"` donneront le même résultat *'$'*!

Malheureusement, *beaucoup trop* de programmes utilisent ce genre d'artifices, qui reviennent à traiter un cas d'*exception* comme un cas *normal* et donc à créer des confusions parfois difficiles à débrouiller par la suite, et contraire à la qualité des programmes. Les langages "modernes", et notamment CAML, offrent une solution claire à ce type de problème, grâce aux mécanismes d'*exceptions*. Pour l'instant, notons que l'on peut utiliser la fonction prédéfinie `failwith`, qui prend un argument de type `string`, et rend un résultat d'un type spécial, appelé `exception`, *compatible* avec n'importe que autre type :

```
# let dernier_avec_controle ch =
    if ch <>"" then dernier ch
               else failwith "erreur : chaine vide";;
 dernier_avec_controle:string -> char=<fun>
```

Le typage de la valeur fonctionnelle `dernier_avec_controle` a donc réussi, sans utiliser de valeur "bidon", et de plus, toute tentative d'utilisation de cette fonction avec une chaîne vide va faire échouer le processus d'évaluation avec indication de la cause d'échec, ce qui est bien l'effet souhaité :

```
# dernier_avec_controle "";;
 Uncaught exception: Failure "erreur : chaine vide"

# dernier_avec_controle "cout : 25$";;
 -:char = '$'
```

5.2.5 Exemple récapitulatif : tête et reste d'une liste non vide.

Nous voulons définir deux fonctions, délivrant respectivement la tête et le reste d'une liste quelconque. Il est clair que de telles fonctions doivent être polymorphes, puisque leur définition est indépendante du type de base de la liste. De plus, elles ne sont pas définies sur les listes vides, donc elles doivent contenir des cas d'exceptions. Enfin, leur définition n'est possible que par filtrage, permettant l'identification des identificateurs du filtre avec les valeurs

des arguments[1] . On obtient :

```
# let tete = function
    [] -> failwith("liste vide")
  | x :: r -> x;;
tete : 'a list -> 'a = <fun>

# let reste = function
    [] -> failwith("liste vide")
  | x :: r -> r;;
tete : 'a list -> 'a list = <fun>
```

A noter que les deux fonctions prédéfinies `hd` (pour *head*) et `tl` (pour *tail*) sont synonymes des deux fonctions que nous venons de définir.

5.3 Processus d'évaluation des expressions

Dans cette partie nous rappelons et complétons le fonctionnement du processus d'évaluation d'une expression, basé sur le *modèle par substitution* déjà abordé dans la sections 3.2.2. Une expression générale est de la forme:

$$f e_1 e_2 \ldots e_p$$

où f est une expression du type t_1 -> t_2 -> ... -> t_n -> t_{res}, e_1 est une expression de type t_1, ..., e_p est un e expression de type t_p et $p \leq n$.

Dans un contexte où l'évaluation est possible (tous les identificateurs sont liés et le typage est correct) le processus est le suivant:

Soit $\mathcal{C}$ le contexte existant au début de l'évaluation.

1. Les expressions $e_1, \ldots, e_p$ sont évaluées dans le contexte $\mathcal{C}$ (l'ordre dans lequel elles sont évaluées est indifférent : ces expressions sont logiquement indépendantes). Soit $v_1, \ldots, v_p$ la désignation des valeurs obtenues. Le contexte $\mathcal{C}$ est le même qu'au début du processus.
2. L'expression f est évaluée dans le contexte $\mathcal{C}$. Le résultat de cette évaluation est une valeur fonctionnelle de la forme
   function arg_1 ->**function** arg_2 -> ... **function** arg_n ->*expr_def*
   et elle contient un contexte local
   $[arg_1\tilde{}?; arg_2\tilde{}?; \ldots; arg_n\tilde{}?]$.
3. L'expression de définition *expr_def* est transformée de la manière suivante:
   (a) Tous les identificateurs apparaissant dans *expr_def* sont liés dans l e contexte local de f ou dans le contexte $\mathcal{C}$.

1. Si vous n'êtes pas convaincu(e) du bien-fondé de la définition par filtrage, essayez donc de définir ces fonctions avec des conditionnelles (**if ... then ... else**) . Si vous trouvez, envoyez un mail à `helary@irisa.fr`

(b) Les identificateurs autres que $arg_1, \ldots, arg_p, arg_{p+1}, \ldots, arg_n$ sont remplacés par leur valeur dans $\mathcal{C}$.

(c) Les identificateurs $arg_1, \ldots, arg_p$ (arguments *formels*) sont remplacés par les valeurs respectives $v_1, \ldots, v_p$ (arguments *effectifs*) : c'est la *substitution* des arguments effectifs aux arguments formels (remplacement de valeurs inconnues par des valeurs effectives).

(d) Les identificateurs $arg_{p+1}, \ldots, arg_n$ ne sont pas remplacés.

On obtient ainsi l'expression *expr_def_2*

4. Le résultat final est la valeur fonctionnelle **function** arg_{p+1} -> ... **function** arg_n -> *expr_def_2* , de type t_{p+1} -> ... -> t_n -> t_{res} , avec le contexte local $[arg_{p+1}\tilde{}?; \ldots; arg_n\tilde{}?]$.
5. Les liaisons arg_1 ~ v_1 , ... , arg_p ~ v_p sont détruites

Remarque Si $p = n$, tous les arguments formels sont remplacés par des valeurs effectives, le résultat est une valeur de type t_{res} , évaluation de *expr_def* dans le contexte $[arg_1\tilde{}v_1?; \ldots; arg_n\tilde{}v_n; \mathcal{C}]$.

Exemples

I. Évaluation de l'expression `successeur 4` dans le contexte $\mathcal{C}$ = [`successeur` ~**function** `n ->n+1`] .

Cette expression est de la forme fe_1 , avec:
f de type `int ->int` e_1 est de type `int`

1. L'évaluation de e_1 donne la valeur $v_1 = 4$.
2. L'évaluation de `f` donne la valeur **function** `n ->n+1` avec le contexte local [`n ~?`]
3. L'expression de définition `n+1` est transformée comme suit:
   (a) Il n'y a pas d'identificateur autre que `arg` 1 = `n`
   (b) La substitution de l'argument effectif v_1 à l'argument `n` donne l'expression `4+1`
   (c) Ici, $p = n = 1$
4. Le résultat final est donc la valeur `5`, de type t_{res} =`int`.
5. La liaison `n ~4` est détruite.

II. Évaluation de l'expression `pas ((`**function** `x ->x*x) 3)` dans le contexte $\mathcal{C}$ = [`pas` ~**function** `x ->x+k ; k ~2`; ...] .

Cette expression est de la forme fe_1 , avec:
f de type `int ->int` et le contexte local [`x ~?`]
e_1 est l'expression `(`**function** `x ->x*x) 3`

1. Évaluation de e_1 . Cette expression est de la forme ge_2 avec g du type `int ->int` et un contexte local [`x ~?`] , e_2 du type `int`.
   (a) L'évaluation de e_2 donne la valeur $v_2 = 3$
   (b) L'évaluation de g donne la valeur **function** `x ->x*x`

   (c) Son expression de définition `x*x` est évaluée dans le contexte [`x ~3`; ...] , ce qui donne à e_1 la valeur $v_1 = 9$.
   (d) La liaison `x ~3` est détruite
2. L'évaluation de f donne la valeur **function** `x ->x+k` et le contexte local [`x ~?`]
3. Son expression de définition `x+k` est évaluée dans le contexte [`x ~9`; `pas ~ ...` ; `k ~2`; ...] , d'où le résultat de valeur 11.
4. La liaison `x ~9` est détruite.

III. Évaluation de l'expression `max2 5. 3.5` dans le contexte $\mathcal{C}$ = [`max2 ~` *cf. définition donnée page 43*; ...] .

Cette expression est de la forme fe_1e_2 , avec:
f de type `float ->float ->float` e_1 et e_2 sont de type `float`

1. e_1 est évaluée $v_1 = 5$.
2. e_2 est évaluée $v_2 = 3.5$
3. L'évaluation de f donne la valeur **function** `x ->` **function** `y ->expr_def`, où `expr_def` est l'expression conditionnelle

   if `x>=.y` **then** `x` **else** `y`

   et le contexte local [`x ~?`; `y ~?`]
4. L'expression de définition est évaluée dans le contexte [`x ~5.` ; `y ~3.5`] ce qui donne la valeur résultat `5.`
5. Les liaisons `x ~5.` et `y ~3.5` sont détruites

IV. Évaluation de l'expression `max2 5.` dans le même contexte que l'exemple précédent.

Cette expression est de la forme fe_1 , avec:
f de type `float ->float ->float` e_1 est de type `float`

1. e_1 est évaluée $v_1 = 5$.
2. L'évaluation de f donne la valeur **function** `x ->` **function** `y ->expr_def`, où `expr_def` est l'expression conditionnelle

   if `x>=.y` **then** `x` **else** `y`

   et le contexte local [`x ~?`; `y ~?`]
3. L'expression de définition est évaluée dans le contexte [`x ~5.` ; ...] ce qui donne l'expression `expr _def_2`:

   function `y ->`**if** `5.>=.y` **then** `5.` **else** `y`

   qui est une valeur fonctionnelle de type `float ->float`, avec le contexte local [`y ~?`] .

4. La liaison `x ~5.` est détruite.

Chapitre 6

Récursion

6.1 Introduction

La récursion est un mode d'expression permettant d'établir des définitions faisant référence à elles-mêmes. Par exemple, la définition suivante de la notion d'ancêtre est récursive : *un ancêtre de x est soit un parent de x, soit un ancêtre d'un parent de x.* Cette définition fait donc référence non seulement à la notion de parent (qui doit être déjà définie), mais aussi à la notion d'ancêtre, en cours de définition. Deux remarques s'imposent immédiatement à propos d'une telle définition :

1. Sa concision (certains diront ... abstraction !). A ce titre, elle s'oppose à des styles plus "opérationnels" de définition, comme par exemple : *un ancêtre de x est une personne telle qu'il existe une chaîne de parents remontant de x jusqu'à cette personne.*
2. Elle porte en elle un *schéma d'algorithme* permettant de traiter des problèmes relatifs aux ancêtres, comme par exemple : étant donné un ensemble de personnes muni de relations de parenté, et deux personnes x et y de cet ensemble, y est-elle ancêtre de x? Ou encore : étant donné un ensemble de personnes muni de relations de parenté, et une personne x , établir l'ensemble des ancêtres de x. En effet, dans un cas comme dans l'autre, un algorithme basé sur la définition récursive consisterait à considérer d'abord le sous-ensemble des parents de x, puis à recommencer le traitement à partir de chacun des parents.

Le point 1. ci-dessus met l'accent sur la concision d'une définition récursive, et le point 2. sur son applicabilité. Les deux aspects sont importants, car une définition trop concise pourrait être inapplicable, comme le montre l'exemple trivial suivant définissant la notion de menteur : *Un menteur est un menteur.* En effet, cette définition fait référence uniquement à elle-même, et donc suppose la notion *complètement* définie avant d'être définie... L'exemple précédent relatif à la notion d'ancêtre ne souffrait pas du même défaut, car la définition faisait référence à une *autre* définition (celle de parents), en ne supposant la

notion d'ancêtre déjà définie que *partiellement* (sur le sous-ensemble des parents). Cet aspect essentiel de la récursion, pour l'instant très informel, sera bien sûr l'objet d'une étude détaillée dans la partie plus technique de ce chapitre.

Terminons cette introduction en montrant, sur quelques exemples, l'étendue du domaine d'application de la récursion, que ce soit pour exprimer des définitions, des calculs, des actions, des évaluations, etc.

Définition d' une action. *Répéter* n fois une action $\mathcal{A}$: si $n > 0$ alors effectuer l'action $\mathcal{A}$ puis *répéter* $n-1$ fois l'action $\mathcal{A}$.

Définition d'une évaluation (cf. chapitre 5.) *Évaluer* l'expression $fe_1e_2 \ldots e_p$: *évaluer* les expressions e_1,$e_2, \ldots$,e_p puis appliquer f aux valeurs obtenues.

Définition d'une structure. *Séquence* de longueur n: si $n = 0$ alors séquence vide, sinon élément suivi d'une *séquence* de longueur $n-1$.

Définition d'un calcul. *Factorielle* du nombre entier n: si n=0 alors 1 sinon $n\times$ *factorielle* de $(n-1)$.

Définition d'une courbe fractale. Tracer la *fractale* de la génération n : si $n = 0$ alors tracer le motif de base, sinon remplacer chacun des fragments de la *fractale* de génération $n-1$ par le motif de base, convenablement dimensionné et orienté.

6.2 Construction de définitions récursives

Les exemples précédents (sauf celui du menteur!) montrent tous qu'une définition récursive est obtenue à partir de définitions *partielles* de la même notion et de définitions déjà connues d'autres notions, que l'on sait combiner pour obtenir la définition complète. Il s'agit en fait d'une expression du principe général *"diviser pour régner"*: dans tous les cas, il s'agit de *diviser* l'univers de la définition en sous-univers sur lesquels la définition est supposée déjà établie, puis de "recomposer" les morceaux pour obtenir la définition complète (*régner*). Tout l'art de la récursion consiste alors à trouver une division à partir de laquelle l'expression sur chacune des parties est simple (décomposition récursive), et telle que la recomposition des morceaux s'exprime aisément. Nous allons d'abord examiner quelques exemples de calculs récursifs. Ensuite, nous donnerons quelques techniques générales menant à de "bonnes" décompositions récursives.

6.2.1 Exemples de définitions récursives de calculs

Somme des valeurs des chiffres d'un entier décimal. Il s'agit de calculer la valeur entière obtenue en additionnant les valeurs de chacun des chiffres de la notation décimale d'un entier n; par exemple, avec $n = 5$ le résultat vaut 5; avec $n = 65$ le résultat vaut 11; avec $n = 765$, le résultat vaut 18, etc. Si le nombre n a plus d'un chiffre, il s'écrit sous forme d'une séquence de chiffres décimaux : $c_k c_{k-1} \ldots c_1 c_0$. Décomposons cette séquence en deux sous-séquences : $c_k c_{k-1} \ldots c_\ell$ et $c_{\ell-1} \ldots c_0$ (avec $k \geq \ell \geq 1$) dénotant respectivement deux entiers n_1 et n_2. A partir du résultat du calcul pour ces deux sous-séquences, dénotés respectivement s_1 et s_2, il est facile de recomposer le résultat cherché, qui vaut évidemment $s_1 + s_2$.

Par exemple, avec $n = 1374286$, décomposé en $n_1 = 137$ et $n_2 = 4286$, on a :

$somme\_des\_chiffres(n_1) = 11$, $somme\_des\_chiffres(n_2) = 20$, d'où $somme\_des\_chiffres(n) = 11 + 20 = 31$.

Il y a de multiples façons d'effectuer la décomposition d'une séquence en deux sous-séquences consécutives. Dans le cas présent, le choix doit être effectué de manière à faciliter les évaluations intermédiaires, à savoir : les valeurs entières n_1 et n_2 doivent être faciles à obtenir à partir des deux sous-séquences. L'arithmétique élémentaire nous suggère une décomposition simple : la *séquence privée du dernier chiffre* d'une part (valeur correspondante : quotient de la division par 10), le *dernier chiffre* d'autre part (valeur correspondante : reste de la division par 10). De plus, le résultat du calcul pour un nombre à un chiffre est immédiat : c'est le nombre lui-même.

Basé sur cette décomposition, on obtient la définition récursive suivante :

```
somme_des_chiffres de n = si n a un seul chiffre
                          alors n
                          sinon (somme_des_chiffres de n/10) + n mod 10
                      fsi
```

Nous en verrons l'expression CAML dans les sections suivantes.

Présence d'un élément dans une séquence Étant donnés une séquence σ d'éléments (de type quelconque) et une valeur v de ce type, déterminer si cette valeur est présente dans la séquence. Par exemple, le caractère 'a' est présent dans la séquence (chaîne) de caractères "La disparition" mais le caractère 'e' en est absent... Décomposons cette séquence en deux sous-séquences consécutives σ_1 et σ_2. A partir du résultat du calcul pour ces deux sous-séquences, dénotés respectivement b_1 et b_2, il est facile de recomposer le résultat cherché, qui vaut évidemment b_1 **ou** b_2 (la valeur est présente dans σ si et seulement si elle est dans σ_1 ou dans σ_2). Dans cet exemple aussi, il y a de multiples

manières d'effectuer la décomposition en deux sous-séquences. Tout dépend de la structure de cette séquence (c'est-à-dire du mode d'accès à ses éléments : séquentiel, direct, etc. et de propriétés plus spécifiques concernant par exemple l'ordre dans lequel les éléments sont rangés, etc.).

Pour fixer les idées, examinons le cas d'une chaîne de caractères. En général, il est facile d'en extraire le *premier caractère* d'une part, et la chaîne privée du premier caractère (le *reste*) d'autre part (en CAML, par exemple, les fonctions correspondantes sont faciles à définir). Désignons respectivement par *premier* et *sauf_premier* les deux fonctions correspondantes. On remarque de plus que, dans le cas d'une chaîne réduite à un caractère, le résultat est immédiat : il vaut *vrai* si et seulement si ce caractère est le caractère cherché! D'autre part, il faut aussi tenir compte du cas de la chaîne vide, pour laquelle les deux fonctions *premier* et *sauf_premier* ne sont pas applicables; or dans ce cas aussi le résultat est immédiat : il vaut *faux* (aucun caractère n'est présent dans une chaîne vide!).

Basé sur cette décomposition, on obtient la définition récursive suivante :

```
presence de c dans ch = si ch est vide
                        alors faux
                        sinon si premier de ch = c
                              alors vrai (* dans ce cas, inutile d'aller voir plus loin *)
                              sinon presence de c dans reste de ch
                                    (* dans ce cas, le caractère cherché
                                       ne peut être que dans le reste *)
```

Puissance (méthode rapide) . Le calcul consiste à calculer la puissance n d'un nombre a (n est entier non négatif, a est quelconque). On sait que, si n est décomposé en la somme d'entiers : $n = n_1 + \ldots + n_k$, alors $a^n = a^{n_1} \times \ldots \times a^{n_k}$. De plus, si ces entiers sont non nuls, ils sont chacun strictement plus petits que n. Parmi les multiples manières d'exprimer n comme somme d' entiers non nuls, considérons celle qui est basée sur l'égalité de la division par deux : soit $n/2$ et $n \ mod \ 2$ le quotient et le reste de la division de n par 2. On a évidemment : $n = n/2 + n/2 + n \ mod \ 2$. Par exemple, 4575 = 2287+2287+1, tandis que 64482=32241+32241+0. Si l'on remarque par ailleurs que $n \ mod \ 2$ est égal soit à 0 (si n est pair), soit à 1 (si n est impair), et que $a^0 = 1$, $a^1 = a$, on obtient la définition récursive suivante :

```
puissance n de a =
    si n=0
       alors 1
       sinon si n=1
                alors a
                sinon soit x=puissance (n/2) de a dans
```

```
                    si n mod 2 =0
                       alors  x ×  x ×  1
                       sinon  x ×  x ×  a
                    fsi
            fsi
    fsi
```

Cette méthode est qualifiée de rapide, car à chaque étape la puissance à calculer est *divisée* par deux. Il y a donc au plus $log_2 n$ étapes . Examinons par exemple le déroulement sur l'exemple suivant : calcul de 2^{25}. La définition "opérationnelle " classique engendrerait la multiplication de 2 par lui-même 24 fois! Par contre, le processus d'exécution qui découle de la définition récursive obtenue ci-dessus est le suivant :

$$
\begin{aligned}
2^{25} &= 2^{12} \times 2^{12} \times 2 \\
2^{12} &= 2^{6} \times 2^{6} \\
2^{6} &= 2^{3} \times 2^{3} \\
2^{3} &= 2^{1} \times 2^{1} \times 2 \\
2^{1} &= 2 \\
&= 2 \times 2 \times 2 = 8 \\
&= 8 \times 8 = 64 \\
&= 64 \times 64 = 4096 \\
&= 4096 \times 4096 \times 2 \\
&= 33554432
\end{aligned}
$$

soit seulement 6 multiplications.

6.3 Mise en œuvre en CAML : fonctions récursives

En CAML, les calculs sont exprimés par des fonctions (algorithmes). Les calculs récursifs sont donc exprimés par la définition de fonctions récursives , c'est-à-dire de fonctions qui font référence à elles-mêmes dans leur expression de définition. Cela suppose déjà que de telles fonctions puissent être *référencées*, c'est-à-dire identifiées par un nom. Par conséquent, une fonction récursive ne peut être définie que dans une phrase de définition commençant par le mot clef **let**. Examinons alors ce que donnerait la définition de la fonction *factorielle*. Sa définition récursive informelle:

```
factorielle n = si n=0
                    alors 1
                    sinon n × factorielle (n-1)
```

pourrait se traduire en CAML par:

```
# let factorielle n = if n=0
                      then 1
                      else n*(factorielle(n-1));;
 > Toplevel input:
>                  else n*(factorielle (n-1));;
>                       ^^^^^^^^^^^
> Variable factorielle is unbound.
```

La réponse de la machine montre clairement qu'une telle définition n'est pas acceptée! En effet, si la présence, dans l'expression de définition, de l'identificateur `n` ne pose pas de problème (il est lié à l'argument formel), il n'en va pas de même de l'identificateur `factorielle`. Le problème vient de ce que la phrase ci-dessus est une *définition* de cet identificateur, et la machine CAML voudrait que cet identificateur soit *déjà* lié dans le contexte. La construction **let** n'est donc pas adéquate. C'est pourquoi CAML propose la construction **let rec** adaptée à ce cas :

```
# let rec factorielle n = if n=0
                          then 1
                          else n*factorielle(n-1);;
factorielle : int -> int = <fun>
```

L'effet de la construction **let rec** est donc d'établir une liaison `factorielle ~?` dans le contexte, et ainsi de pouvoir accepter la présence de l'identificateur `factorielle` dans l'expression de définition. Cette liaison sera complètement résolue (c'est-à-dire le *type* et la *valeur* seront déterminés) lorsque la phrase aura été complètement traitée par le compilateur [1].

La *syntaxe* de définition d'une fonction récursive est donc la suivante :

let rec `id_fonction arg` $_1$ `...  arg` $_n$ `= expression_def`

où `id_fonction` apparaît dans `expression_def`, ou encore, sous forme non abrégée :

let rec `id_fonction =` **function** `id_arg -> expression_def`

Pour terminer cette section, nous donnons le texte CAML des définitions de fonctions récursives vues depuis le début de ce chapitre.

Somme des chiffres

```
# let rec somme_des_chiffres n =
      if n<10 then n
           else somme_des_chiffres (n/10) + n mod 10;;
somme_des_chiffres : int -> int =<fun>
```

1. Nous ne détaillerons pas, dans ce cas, l'algorithme de synthèse de type, analogue dans son principe à celui vu au chapitre précédent.

Présence d'un caractère dans une chaîne.

```
# let rec present c ch =
      if vide ch
          then false
          else if premier ch = c
                    then true
                    else present c (sauf_premier ch);;
present : char -> string -> bool =<fun>
```

(les fonctions `vide, premier, sauf_premier` doivent avoir été définies préalablement. Ceci est laissé en exercice au lecteur).

Puissance (méthode rapide).

```
let rec puiss_dich a n =
      if n=0
          then 1
          else  let x=puiss_dich a (n/2)  in
                                  if n mod 2=0
                                       then x * x
                                       else x*x*a;;
puiss_dich : int -> int -> int =<fun>
```

6.4 Méthodologie

Les trois exemples vus précédemment sont tous construits selon la même méthodologie, que nous synthétisons ci-dessous. Cette méthode comporte trois phases:

1. Spécification du calcul
2. Équation de définition
3. Cas d'arrêt

Une telle construction méthodique permet d'obtenir le texte de la fonction, que ce soit en CAML ou dans tout autre langage autorisant la récursion. Dans ce document, nous nous limitons au langage CAML comme support de programmation, mais la méthodologie de construction de fonctions récursives est évidemment utilisable lorsque l'on utilise d'autres langages, de style fonctionnel (LISP, Scheme, ...) ou impératif (Pascal, ...).

Un deuxième aspect, complémentaire de la construction, doit être examiné: c'est celui de la *correction*, qui doit être formellement prouvée. Cette preuve se compose de deux parties:

1. Sûreté

2. Vivacité

– La *sûreté* exprime la propriété suivante : *si l'exécution du calcul se termine, alors le résultat est conforme à la spécification.*
– La *vivacité* exprime la propriété suivante : *l'exécution se termine en un temps fini.*

6.4.1 Construction

Reprenons en détail les trois points de la méthode de construction.

1. Spécification : il s'agit de spécifier les données et les résultats (avec leurs types). Si l'on considère le calcul comme une fonction, cette phase consiste donc à typer cette fonction – au sens du typage des valeurs fonctionnelles vu dans les chapitres précédents.

2. Équation de définition : cette équation exprime le résultat du calcul pour une donnée d en fonction des résultats du même calcul sur des données d_1,d_2,...,d_k. C'est l'expression de la décomposition récursive résultant de l'approche *diviser pour régner*. Cette équation sert aussi à prouver la sûreté, en général par récurrence.

3. Cas d'arrêt : l'équation de définition ne s'applique pas à certaines valeurs particulières des données. Il faut alors donner une expression *directe* – c'est-à-dire non récursive – du résultat pour ces valeurs particulières. Ces cas, lorsqu'ils seront rencontrés par le processus d'exécution, constitueront des *cas d'arrêt* de la récursion. La présence d'au moins un cas où l'expression du résultat ne fait référence à aucun résultat du même calcul (cas d'arrêt) est donc *nécessaire* pour assurer la terminaison du processus de calcul. Toutefois, ceci *n'est pas suffisant* : il faut de plus s'assurer que, quelque soit la donnée d, un des cas d'arrêt sera atteint par le processus après un temps fini. Ceci fait l'objet de la preuve de vivacité.

6.4.2 Des exemples

Exemple 1 Étant donné une séquence d'éléments tous de même type, calculer la séquence miroir (c'est-à-dire obtenue en énumérant les éléments du dernier au premier).

1. **Typage** : `miroir : séquence de t ->séquence de t` (`t` désigne le type – quelconque – des éléments).

2. **Équation** : on décompose la séquence en deux parties : sa *tête* (séquence réduite au premier élément) et son *reste* (séquence privée du premier élément). En supposant les deux fonctions `tete : sequence de t ->sequence de t` et `reste : sequence de t ->sequence de t` disponibles, ainsi que la fonction `concatener : sequence de t × sequence de t ->sequence de t` qui concatène deux séquences en une troisième, il est facile de voir que le miroir d'une séquence `ch` est obtenu en concaténant le miroir du reste de `ch` avec la tete de `ch`. On obtient donc l'équation :

   ```
   miroir(ch) = concatener (miroir (reste ch)) (tete ch)
   ```

3. Cas d'arrêt Les deux fonctions `tete` et `reste` ne sont pas définies pour une séquence vide. L'équation ci-dessus n'est donc pas valable lorsque `ch` est la séquence vide. Il faut donc donner une expression directe du résultat dans ce cas. Ici, c'est trivial, puisque le miroir de la séquence vide est évidemment la séquence vide!

Code en CAML Nous prenons l'exemple d'une séquence de caractères, codée sous forme de chaîne (type `string`). Les fonctions `premier` et `sauf_premier` sont définies comme suit :

```
# let premier ch = sub_string ch 0 1 ;;
 premier : string -> string = <fun>

# let sauf_premier ch = sub_string ch 1 ((string_length ch) -1) ;;
 premier : string -> string =<fun>
```

La concaténation des chaînes est une fonction infixe prédéfinie, notée ^. On obtient donc :

```
# let rec miroir ch =
              if ch=""
                  then ch
                  else (miroir (sauf_premier ch)) ^ (premier ch) ;;
 miroir : string -> string = <fun>

# miroir "ONU" ;;
 - : string = "UNO"

# miroir "alavalellelavala" ;;  (* A Laval elle l'avala *)
 - : string = alavalellelavala  (* eh oui! c'est un palindrome *)
```

Preuve : sûreté . Par *récurrence* sur la longueur ℓ de la chaîne `ch`. En effet, l'expression définissant `miroir ch` contient l'expression `miroir (sauf_premier ch)`, montrant que le calcul avec un argument `ch` de longueur ℓ dépend du résultat avec un argument de longueur $\ell - 1$.

cas de base: $\ell = 0$. Cela correspond au cas où `ch=""`; le résultat est alors la valeur `""`, ce qui est correct.

induction: Supposons le résultat vrai pour les chaînes de longueur $\ell - 1$, et montrons qu'il est vrai pour les chaînes de longueur ℓ. Soit `ch` de longueur ℓ ($\ell \geq 1$), avec `ch` $= c_1 c_2 \ldots c_\ell$. D'après la définition,

`miroir ch` $=$ `concatener (miroir (sauf_premier ch)) (premier ch)`
$=$ `concatener (miroir` $c_2 \ldots c_\ell$`)` c_1
$=$ `concatener` $c_\ell \ldots c_2$ c_1 (hypothèse de récurrence)
$= c_\ell \ldots c_2 c_1$

Preuve : vivacité . La longueur de l'argument décroît strictement de 1 à chaque appel récursif, donc ℓ appels sont suffisants pour atteindre le cas d'arrêt $\ell = 0$.

Exemple 2. Calcul de la puissance entière positive ou nulle d'un nombre (méthode séquentielle). Nous donnons ici une construction aboutissant à un calcul incorrect; ceci est volontaire: une version correcte sera donnée ensuite.

Typage : `puissance : nombre ->entier ->nombre`.

Équation. On peut considérer que la puissance n de a est égale à la puissance $(n+1)$, divisée par a (ceci est conforme à la définition mathématique si $a \neq 0$). D'où :

```
puissance a n = (puissance a (n+1))/a
```

Cas d'arrêt L'équation est valable pour tout entier $n \geq 0$, mais n'est pas valable pour $a = 0$. Dans ce cas, le résultat est donné directement par `puissance 0 n = 0`

Texte en CAML Pour simplifier, nous supposerons que a est entier.

```
# let rec puissance a n =
            if a=0
                then 0
                else (puissance a (n+1))/a ;;
puissance : int -> int -> int = <fun>
```

Preuve : validité . Si $a = 0$, le résultat est évident.

Dans le cas où $a \neq 0$, par *récurrence* sur la valeur de n, puisque l'expression définissant `puissance a n` contient l'expression `puissance a (n+1)`.

cas de base: Première apparition du problème: il n'y a pas de valeur de n pour laquelle le résultat calculé par l'exécution du texte ci-dessus est évident (lorsque $a \neq 0$). Nous allons retrouver le même problème dans l'étude de la vivacité.

induction: Supposons le résultat vrai pour a $(n+1)$, et montrons qu'il est vrai pour a n. D'après la définition,

`puissance a n` = `(puissance a (n+1))/a`
= `(a` $\times$ `a` $\times \ldots \times$ `a)` $n+1$ `fois /a` (hypothèse de récurrence)
= `(a` $\times$ `a` $\times \ldots \times$ `a)` n `fois`

Toutefois, l'absence de validation dans un cas de base infirme la preuve par récurrence. La *sûreté* n'est donc pas établie.

vivacité : le cas d'arrêt est obtenu lorsque $a = 0$. Mais, la valeur de l'argument a demeure inchangée à chaque appel récursif, donc ce cas d'arrêt ne sera jamais atteint lorsque l'évaluation est menée avec un argument a de valeur non nulle!

Une tentative d'exécution par la machine CAML du texte donné ci-dessus, avec un argument a non nul, se solderait par un échec du processus (échec dû, dans ce cas, à l'incapacité du système sous-jacent à la machine CAML de gérer une suite "infinie" d'appels). Ceci, alors même que la machine CAML a parfaitement accepté la *définition* de fonction `puissance`, correcte du point de vue syntaxique et du point de vue typage.

Solution correcte: l'équation donnée ci-dessous conduit à une solution correcte :

`puissance a n = a` $\times$ `(puissance a (n-1))`

Cette équation ne s'applique pas lorsque $n = 0$, cas pour lequel le résultat vaut 1. On obtient donc le texte CAML :

```
# let rec puissance a n =
                  if n=0
                     then 1
                     else a*(puissance a (n-1));;
puissance : int -> int -> int = <fun>
```

Les preuves de sûreté et de vivacité ne posent pas de problème: elles sont basées d'une part sur la récurrence par rapport à n (avec le cas de base $n = 0$ et l'induction *résultat correct pour* $n-1$ *implique résultat correct pour* n), et d'autre part sur la décroissance stricte, à chaque appel récursif, de la valeur de l'argument n.

```
# puissance 2 10;;
 -: int = 1024
```

Multiplication russe (voir page 4)

Typage : `mult_russe : ent ->ent ->ent`

Équation : d'après ce qui a été vu page 4 :

```
mult_russe a b = (mult_russe (a/2) (b*2)) + (a mod 2)*b
```

Cas d'arrêt : L'équation précédente est valable pour tout entier non négatif. Toutefois, il est nécessaire de prévoir un cas d'arrêt, que l'on sache traiter directement et que l'on soit sûr d'atteindre en un nombre fini d'applications. Or on remarque que, si `a=0` ou `b=0`, le résultat vaut `0`. Doit-on alors choisir comme cas d'arrêt la première ou la seconde condition? La réponse est fournie par la fonction d'évolution des données :

```
a b -> a/2 b*2
```

qui montre que `a` diminue, tandis que `b` augmente. C'est donc la condition `a=0` qui sera atteinte.

Texte en CAML

```
# let rec mult_russe a b =
    if a=0
       then 0
       else (mult_russe (a/2) (b*2))+(a mod 2)*b;;
 mult_russe : int -> int -> int = <fun>
```

Preuve : sûreté . On raisonne par *induction* sur `a`, en prouvant la propriété suivante : $(\forall a)\ (\forall b)$, `mult_russe a b` $= a \times b$.

cas de base : `a=0`. La propriété est vérifiée : $(\forall b)$, *mult_russe 0 b* $= 0 \times b = 0$.

induction : Montrons que si la propriété est vérifiée pour une valeur a', alors elle est vérifiée pour les valeurs $2 \times a'$ et $2 \times a' + 1$.

cas où $a' \neq 0$:

$$
\begin{array}{lrl}
(\forall b), mult\_russe\ (2 \times a')\ b & = & mult\_russe\ a'\ (b \times 2) \\
 & = & a' \times (b \times 2) \text{ (hypothèse d'induction)} \\
 & = & (2 \times a') \times b \\
(\forall b), mult\_russe\ (2 \times a' + 1)\ b & = & mult\_russe\ a'\ (b \times 2) + b \\
 & = & a' \times (b \times 2) + b \text{ (hypothèse d'induction)} \\
 & = & (2 \times a' + 1) \times b
\end{array}
$$

cas où $a' = 0$

$$\begin{array}{lcl}(\forall b), mult\_russe\ (2 \times 0)\ b & = & mult\_russe\ 0\ b \\ & = & 0 \\ & = & (2 \times 0) \times b \\ (\forall b), mult\_russe\ (2 \times 0 + 1)\ b & = & mult\_russe\ 1\ b \\ & = & mult\_russe\ 0\ (b \times 2) + b \text{ (hypothèse d'induction)} \\ & = & b \\ & = & (2 \times 0 + 1) \times b\end{array}$$

On en déduit que la propriété est vérifiée pour tout entier `a`, puisque tout entier est obtenu à partir de la valeur 0 par un nombre fini d'applications de la fonction $x \rightarrow 2 \times x$ ou de la fonction $x \rightarrow 2 \times x + 1$.

Preuve : vivacité. La valeur de l'argument `a` décroît strictement à chaque appel récursif (`a` → `a/2`). Donc, la valeur `a=0` sera atteinte après un nombre fini d'appels récursifs.

Chapitre 7

Traitements récursifs sur les listes

7.1 Introduction

Dans ce chapitre, la technique de construction de fonctions récursives vue au chapitre précédent est appliquée à divers traitements sur les listes. Rappelons que la structure de liste permet de représenter des collections de données caractérisées par les propriétés suivantes :

1. toutes les données appartiennent à un même type de base,
2. l'organisation et l'accès des données sont séquentiels,
3. la longueur d'une liste (nombre de données) n'est pas bornée *a priori*.

De nombreux traitements nécessitent d'effectuer un *parcours* exhaustif ou partiel de la structure, et d'appliquer un calcul sur chacun des éléments de liste visités lors de ce parcours. Étant donné la vision des listes sous forme de doublet `tete::reste`, de tels traitements s'expriment naturellement selon le schéma récursif suivant :

traitement sur x::r = **si** *parcours terminé*
alors *expression résultat*
sinon *calcul sur x combiné avec traitement sur r*
fsi

L'élément "visité" à chaque étape est donc l'élément de tête de la liste courante, auquel on applique le calcul spécifique, le parcours étant "matérialisé" par la relance du traitement sur le reste.

Pour exprimer ce schéma[1] nous aurons recours à deux fonctions d'accès prédéfinies, respectivement:

- `hd : 'a list -> 'a`, telle que : `hd l` délivre la valeur du premier élément (c'est-à-dire la tête) de la liste `l` *si cette liste est non vide*, et lève une exception (failure) sinon.
- `tl : 'a list -> 'a list`, telle que : `tl l` délivre une copie de la liste `l` privée de son premier élément (c'est-à-dire le reste de la liste `l`) *si cette liste est non vide*, et lève une exception (failure) sinon.

Pour tester si une liste est vide, on utilisera la fonction `vide : 'a list ->bool` définie ainsi :

```
let vide l = l=[] ;;
```

ou directement l'expression `l=[]`.

Enfin, la construction d'une liste se fera par la fonction `cons : 'a*'a list -> 'a list` définie ainsi :

```
let cons v l = v::l;;
```

ou directement par l'expression `v::l`

Deux des schémas les plus souvent rencontrés sont les suivants :

Parcours exhaustifs de liste:

```
let rec parcours l =
   if vide l
      then   valeur de base  (* correspond au "cas d'arrêt" liste vide *)
      else g (f (hd l)) (parcours (tl l)) ;;
```

parcours partiel de liste : (arrêt sur conditions)

```
let rec parcours_part l
   if vide l
       then  valeur de base  (* correspond au "cas d'arrêt" liste vide *)
       else if C (hd l)
             then  autre valeur de base  (* correspond à un autre cas d'arrêt *)
             else g (f (hd l)) (parcours_part (tl l)) ;;
```

Nous allons donner, dans les section suivantes, quelques exemples représentatifs de tels traitements, classés selon le type de résultat cherché : fonctions de type *liste de t -> t* ("accesseurs"), de type *liste de t -> liste de t'* ou *t -> liste de t'*

1. En CAML, l'outil privilégié pour exprimer ce schéma est celui de filtrage, les différents cas de filtrage étant définis par des filtres basés sur les constructeurs `[]` et `::`. Toutefois, pour des raisons déjà évoquées, nous préférons, dans ce cours, éviter d'avoir recours à cette technique spécifique.

("constructeurs") et enfin fonctions combinant ces cas. Dans chaque catégorie, nous montrerons aussi comment l'on peut concevoir des fonctions de traitement *génériques*, au sens où de telles fonctions expriment des traitements paramétrables non seulement par les données, mais aussi par d'autres fonctions.

7.2 Fonctions d'accès aux listes

7.2.1 Parcours exhaustifs

7.2.1.1 Longueur d'une liste

Il s'agit de calculer le nombre d'éléments d'une liste.

Spécification: *'a list ->int*

Équation de définition: *longueur x::r = longueur r +1*

Cas d'arrêt: l'équation précédente ne s'applique pas si la liste est vide. Dans ce cas, on a:
longueur []= 0

Code en CAML :

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

7.2.1.2 Somme des valeurs des éléments d'une liste d'entiers

Spécification: *int list ->int*

Équation de définition: *somme x::r = longueur r + x*

Cas d'arrêt: l'équation précédente ne s'applique pas si la liste est vide. Dans ce cas, on a:
somme []= 0

Code en CAML :

```
# let rec somme l =
             if vide l
```

```
                then  0
                else somme (tl l) + (hd l);;
somme : int list -> int = <fun>
```

On remarque déjà une grande similitude avec la fonction `longueur`. Poussons cette similitude encore plus loin avec l'exemple suivant:

7.2.1.3 Transformer une liste de caractères en chaîne

Il s'agit décrire la fonction `liste_vers_ch` telle que:
`liste_vers_ch` $[c_1; c_2; \ldots; c_n]$, où les c_i sont des caractères, ait pour valeur la chaîne $c_1 c_2 \ldots c_n$.

Spécification: *char list ->string*

Équation de définition: *liste_vers_ch x::r = (chaine x)^(liste_vers_ch r)*

Cas d'arrêt: l'équation précédente ne s'applique pas si la liste est vide. Dans ce cas, on a:
liste_vers_ch []= ""

Code en CAML :

```
# let rec liste_vers_ch l =
             if vide l
                then  ""
                else (char_for_read (hd l))^(liste_vers_ch (tl l));;
liste_vers_ch : char list -> string = <fun>
```

7.2.1.4 Fonction d'accumulation générique

Les trois exemples que nous venons d'écrire sont identiques dans leur structure, et ne diffèrent entre eux que par les éléments suivants :

1. la valeur de base lorsque la liste est vide, soit `e`
2. la fonction de calcul appliquée à la tête, soit `f`
3. la fonction d'accumulation, combinant le résultat du calcul sur la tête au résultat du traitement sur le reste, soit `g`

Soit `t` le type de base de la liste argument, et `t'` le type du résultat. Les trois fonctions que nous venons d'écrire sont toutes du type `t list ->t'`. De plus, `e` est de type `t'`, `f` est de type `t ->t'` et `g` est de type `t' ->t' ->t'`. Nous allons abstraire les trois fonctions précédentes par une fonction générique `accumuler`, ayant comme arguments non seulement une liste, mais aussi les entités `e, f, g`.

Spécification: *t' ->(t ->t') ->(t' ->t' ->t') ->(t list) ->t'*

Équation de définition:
accumuler e f g (x::r) = g (f x) (accumuler e f g r)

Cas d'arrêt: l'équation précédente ne s'applique pas si la liste est vide. Dans ce cas, on a:
accumuler e f g []= e

Code en CAML :

```
# let rec accumuler e f g l =
             if vide l
                then  e
                else  g (f (hd l)) (accumuler e f g (tl l));;
 accumuler : 'a -> ('b -> 'c) -> ('c -> 'a -> 'a) -> 'b list -> 'a = <fun>
```

Exemples d'utilisation Nous allons retrouver, à partir de cette fonction générique. les trois fonctions précédentes, ainsi qu'un autre exemple.

Longueur. Ici, `e=0`, `f` est la fonction constante `x ->1`, `g` est l'addition des entiers (notée `prefix +` en CAML). D'où :

```
# let longueur l =
  accumuler 0 (function x -> 1) ( prefix +) l ;;
 longueur : 'a list -> int = <fun>
```

Somme. Ici, `e=0`, `f` est l'identité, `g` est l'addition des entiers. D'où :

```
# let somme l =
  accumuler 0 (function x -> x) (prefix +) l ;;
 somme : int list -> int = <fun>
```

liste_vers_ch. Ici, `e = ""`, `f = char_for_read`, `g` est la concaténation des chaînes (notée `prefix ^` en CAML). D'où :

```
# let liste_vers_ch l = accumuler "" char_for_read (prefix ^) l ;;
 liste_vers_ch : char list -> string = <fun>
```

Minimum conditionnel. Pour terminer, écrivons une fonction qui, appliquée à une liste de doublets `float*bool` délivre le minimum des flottants pour lesquels le champ booléen correspondant vaut *true* (et, par convention, `1.e-300` si l'ensemble des couples de champ booléen égal à *true* est vide). Ici, `e=1.e-300`, `f` est la fonction qui délivre le premier champ (flottant) si le deuxième vaut

true, et 1.e-300 sinon, et `g` est la fonction *minimum* sur les flottants. On peut définir localement ces deux fonctions, et l'on obtient :

```
# let min_cond l =
       let f x = if snd x then fst x else 1.e-300  and
           g x y = if x<=.y then x else y in
      accumuler 1.e-300 f g l ;;
min_cond : (float * bool) list -> float = <fun>
```

7.2.2 Parcours partiels

Certains traitements ne nécessitent pas de parcourir systématiquement toute la liste, mais seulement une partie allant de la tête jusqu'au premier élément vérifiant une certaine condition (s'il existe). Cela ressemble au paradigme de la boulangerie : supposons que l'on soit à la recherche d'une boulangerie dans une rue donnée. La méthode consiste à parcourir la rue depuis l'une de ses extrémités, jusqu'à ce qu'une boulangerie soit trouvée (dans ce cas, il est inutile d'aller plus loin : le parcours s'arrête sur un *succès*) ou jusqu'à ce que l'on ait parcouru entièrement la rue sans trouver de boulangerie (le parcours s'arrête sur un *échec*).

7.2.2.1 Recherche de la présence d'une valeur dans une liste

Il s'agit d'écrire une fonction *membre* qui, étant donné une une valeur de type `t` et liste de `t`, rend *vrai* si et seulement si la valeur est présente dans la liste.

Spécification: *t ->liste de t ->bool*

Équation de définition: *membre v x::r = si x=v alors vrai sinon membre v r*

Cas d'arrêt: l'équation précédente ne s'applique pas si la liste est vide. Dans ce cas, on a:
membre v []= faux

Code en CAML :

```
# let rec membre v l =
             if vide l
                then  false
                else if hd l = v
                        then true
                        else membre v (tl l) ;;
```

membre : 'a -> 'a list -> bool = <fun>

7.2.2.2 Recherche générique

Il s'agit de généraliser la fonction précédente à la recherche de l'existence d'un élément vérifiant une condition donnée, et qui délivre le résultat d'un calcul sur cet élément, ou une valeur par défaut si aucun élément ne vérifie la condition. La fonction admet donc comme arguments, outre la liste à parcourir:

- la condition `C` (fonction de type `t ->bool`)
- la fonction exprimant le calcul à effectuer sur l'élément trouvé (fonction `f : t ->t'`)
- la valeur par défaut `e : t'`

Spécification: *(t ->bool) ->(t ->t') ->t' ->(liste de t) ->t'*

Équation de définition:
recherche C f e (x::r) = si C x alors f x sinon recherche C f e r

Cas d'arrêt: l'équation précédente ne s'applique pas si la liste est vide. Ce cas correspond à l'absence d'élément vérifiant la condition `C` , et donc : *recherche C f e []= e*

Code en CAML :

```
# let rec recherche C f e l =
             if vide l
                then  e
                else if C (hd l)
                        then f (hd l)
                        else recherche C f e (tl l);;
```

recherche : ('a -> bool) -> ('a -> 'b) -> 'b -> 'a list -> 'b = <fun>

A titre d'application, la fonction *membre* peut être définie par:

```
# let membre v l =
  recherche (function x -> x=v) (function x-> true) false l ;;
```

membre : 'a -> 'a list -> bool = <fun>

Autre d'exemple d'utilisation de la fonction générique *recherche*: on considère une liste de triplets `string*int*float`. Chaque triplet représente un article en stock, chaque article ayant comme attribut un nom (chaîne), une quantité disponible (entier) et un prix unitaire hors-taxes (flottant). On donne un nom d'article `ch` et une quantité désirée `n`, et on cherche s'il existe un article ayant ce nom et disponible en quantité suffisante (au moins `n` unités disponibles). Si c'est le cas, la réponse est constituée : de la chaîne "trouvé", du coût des `n`

articles demandés, et d'un nouvel article de même nom, même prix unitaire, et de quantité disponible diminuée de `n`. Si ce n'est pas le cas, on rend la chaîne "echec", le coût 0 et l'article ("",0,0).

Le type de la fonction est donc :

```
string ->int ->(string*int*float) list ->string*float*(string*int*float)
```

On l'obtient à partir de la fonction *recherche* avec:

- La condition `C` est la fonction qui, appliquée au triplet `(nom,qte,pu)` rend vrai si et seulement si `nom=ch` **et** `n<=qte`
- La fonction `f` est la fonction qui, appliquée au triplet `(nom,qte,pu)` rend le triplet `("trouvé", n*pu, (nom,qte-n,pu))`
- La valeur par défaut est le triplet `("echec", 0, ("",0,0))`

```
# let rech_stock ch n =
  let C (nom,qte,pu) = (nom=ch) & (qte>=n)  and
      f (nom,qte,pu) = ("trouve", (float_of_int n)*.pu,(nom,qte-n,pu))
  in recherche C f ("echec,0.,("",0,0.)) ;;
rech_stock : string -> int -> (string * int * float) list -> string * float * (string * int * float) = <fun>
```

7.3 Fonctions de constructions de listes

Il s'agit de fonctions qui, appliquées à une liste, délivrent une autre liste. Le principe général de raisonnement est le suivant: le résultat à exprimer étant une liste, il s'agit de définir sa *tête*, son *reste* puis de combiner les deux avec le constructeur `::`. Voyons quelques exemples.

7.3.1 Constructions simples

7.3.1.1 Ajout d'un élément en fin de liste

Étant donné une liste de valeurs de type `t` et une valeur de type `t`, construire la liste obtenue en ajoutant cette valeur en fin de liste.

Spécification: *liste de t ->t ->liste de t*

Équation de définition: *append (x::r) v* est une liste. Sa tête est x et son reste est obtenu en ajoutant la valeur v à la fin de r, d'où :

append (x::r) v = x::(append r v)

Cas d'arrêt: l'équation précédente ne s'applique pas si la liste est vide. Dans ce cas, on a:

append []v = v::[]

Code en CAML :

```
# let rec append l v =
             if vide l
                then  cons v l
                else cons (hd l) (append (tl l) v);;
 append : 'a list -> 'a -> 'a list = <fun>
```

7.3.1.2 Suppression

Étant donné une liste d'éléments de type `t` et un entier `n`, il s'agit de construire la liste obtenue en supprimant le n-ème élément de la liste (liste inchangée si `n` est supérieur au nombre d'éléments de la liste).

Le résultat étant une liste, il faut déterminer sa tête et son reste (en fonction de la liste argument et de `n`). Dans le cas particulier où $n = 1$, le traitement revient à supprimer la tête, donc le résultat est le reste de la liste argument. Dans le cas général (n¿1), la tête du résultat est la tête de la liste argument, et le reste du résultat est obtenu en supprimant le $(n-1)$-ème élément du reste de la liste argument.

Spécification: *int ->liste de t ->liste de t*

Équation de définition: *suppression n x::r = si n=1 alors r sinon x::(suppression (n-1) r)*

Cas d'arrêt: l'équation précédente ne s'applique pas si la liste est vide. Dans ce cas, on a:

suppression n []= []

Remarque : on obtiendra ce cas d'arrêt lorsque $n >$ longueur de la liste argument.

Code en CAML :

```
# let rec suppression n l =
             if vide l
                then l
                else if n=1
                        then (tl l)
                        else cons (hd l)(suppression (n-1) (tl l));;
 suppression : int -> 'a list -> 'a list = <fun>
```

7.3.2 Fonctions génériques

7.3.2.1 Appliquer à tous: map

Il s'agit de construire la liste obtenue en appliquant une fonction `f` à tous les éléments d'une liste. C'est donc une abstraction du traitement **faire pour tout ...**. Si `l` est une liste de type de base `t` et `f` une fonction de type `t ->t'`, on obtiendra une liste de type de base `t'`.

Spécification: *(t ->t') ->liste de t ->liste de t'*

Équation de définition: *map f (x::r) = (f x)::(map f r)*

Cas d'arrêt: l'équation précédente ne s'applique pas si la liste est vide. Dans ce cas, on a:
map f []= []

Code en CAML :

```
# let rec map fl =
             if vide l
               then l
               else cons (f (hd l)) (map f (tl l));;
```
map : ('a -> 'b) -> 'a list -> 'b list = <fun>

Remarque La fonction *map* est construite sur le même principe que la fonction générique *accumuler*, vue à la section précédente. De fait, il s'agit d'un cas particulier de *accumuler*, obtenu avec :

- `e` est la valeur [](de type `'b list`)
- `f` est la fonction donnée `f`
- `g` est la fonction `cons`

D'où une autre définition possible de *map*:

```
# let map f l =
     accumuler [] f cons l;;
```

Exemples d'utilisation: Dans une liste, remplacer toutes les occurrences de la valeur v1 par la valeur v2.

Ici, `f` est la fonction qui, à une valeur `x` fait correspondre `v2` si `x=v1` ou `x` sinon.

```
# let remplacer v1 v2 l =
   let f x = if x=v1 then v2 else x in map f l;;
```

remplacer : 'a -> 'a -> 'a list -> 'a list = <fun>

7.3.2.2 Parcours conditionnels

La fonction générique précédente engendrait un parcours exhaustif de toute une liste (**faire pour tout ...**). Selon le même principe, il est possible de programmer des parcours partiels, contrôlés par une condition `C` : **faire tant que, faire jusqu'a, faire si, ...**. A titre d'exemple, nous considérons la fonction générique qui applique une fonction `f` à tous les éléments d'une liste, tant que ceux ci vérifient une certaine condition `C`.

Soit `x::r` la forme de la liste argument. Le résultat est une liste, égale à:

- la liste vide si `x` ne vérifie pas la condition `C`,
- la liste ayant pour tête `f x`, et pour reste le résultat du traitement appliqué à `r`, si `x` vérifie la condition `C`

On a donc :

Spécification: *(t ->bool) ->(t ->t') ->liste de t ->liste de t'*

Équation de définition:

faire_tantque C f (x::r) = si C x alors (f x)::(faire_tantque C f r)
sinon []

Cas d'arrêt: l'équation précédente ne s'applique pas si la liste est vide. Dans ce cas, on a:

faire_tantque C f []= []

Code en CAML :

```
# let rec faire_tantque C f l=
             if vide l
                then l
                else if C (hd l)
                        then cons (f (hd l)) (faire_tantque C f (tl l))
                     else [] ;;
```

faire_ttque : ('a -> bool) -> ('a -> 'b) -> 'a list -> 'b list = <fun>

7.4 Fonctions de génération de listes

Il s'agit de fonctions qui, appliquées à une ou plusieurs valeurs d'un type quelconque, engendrent une autre liste. Le principe général de raisonnement est le même que dans la section précédente: le résultat à exprimer étant une liste, il s'agit de définir sa *tête*, son *reste* puis de combiner les deux avec le constructeur `::`.

7.4.1 Intervalle

Étant donné deux entiers a et b, construire la liste des entiers de l'intervalle [a,b]

Spécification *intervalle : int ->int ->int list*

Équation de définition *intervalle a b = a :: (intervalle (a+1) b)*

Cas particulier Si $a > b$, on a : *intervalle a b = []*

codage en CAML

```
# let rec intervalle a b =
              if a>b then [] else cons a (intervalle (a+1) b);;
 intervalle : int -> int -> int list = <fun>
```

7.4.2 Conversion chaîne ->liste

Étant donné une chaîne de caractères, construire la liste des caractères constituant cette chaîne.

Spécification *ch_versliste : string ->char list*

Équation de définition La tête du résultat *ch_vers_liste ch* est le premier caractère de *ch*, et le reste est le résultat de la fonction appliqué à *ch* privée du premier caractère:
ch_vers_liste ch = (premier ch) :: (ch_vers_liste (sauf_premier ch))

Cas particulier Lorsque *ch* est vide, l'équation ci-dessus ne s'applique pas. Dans ce cas, on a *ch_vers_liste "" = []*

codage en CAML (* rappel des fonctions `premier` et `sauf_premier` : *)

```
# let premier ch = nth_char ch 0  and
      sauf_premier ch = sub_string ch 1 (string_length ch -1);;
 premier : string -> char = <fun>
 sauf_premier : string -> string = <fun>

 (* définition de  ch_vers_liste  *)
# let rec ch_vers_liste ch =
        if ch = ""
```

```
          then []
          else cons (premier ch) (ch_vers_liste (sauf_premier ch));;
ch_vers_liste : string -> char list = <fun>
```

7.4.3 Valeurs successives d'une suite

Une suite de valeurs (d'un type quelconque) est définie par récurrence de la manière suivante :

- première valeur x_1 donnée
- $\forall i \geq 2$: $x_{i+1} = f(x_i)$, où f est une fonction donnée.

On veut écrire la fonction qui construit la liste des n premières valeurs de cette suite, étant donnés la valeur initiale x, la fonction f et le nombre de termes désirés n.

Spécification *suite : 'a ->('a ->'a) ->int ->'a list*

Équation de définition Le résultat de *suite x f n* est une liste à n éléments, $[x_1; x_2; \ldots; x_n]$. Sa tête est donc x_1 (donné) et son reste $[x_2; \ldots; x_n]$. Le reste est donc la liste obtenue en appliquant la fonction *suite* avec $x_2 = f(x_1)$ comme premier élément, f comme fonction de récurrence, et ayant $n - 1$ éléments. D'où :

suite x f n = x :: (suite (f x) f (n-1))

Cas particulier L'équation ci-dessus ne s'applique pas si $n = 0$. Dans ce cas, on a directement *suite x f 0 = []*

codage en CAML

```
# let rec suite x f n =
      if n< 0
        then failwith("n ne doit pas etre negatif")
        else if n=0
              then []
              else  cons x (suite (f x) f (n-1));;
suite : 'a -> ('a -> 'a) -> int -> 'a list
```

7.5 Autres situations

Les fonctions manipulant des listes, que ce soit comme arguments ou comme résultats, peuvent se rencontrer dans de nombreuses situations, mettant en jeu d'autres structures de données. A titre d'exemple nous étudions

deux fonctions génériques : l'une *sépare* une liste en un couple de listes, l'autre, au contraire, *fusionne* un couple de listes en une seule. Dans les deux cas, un critère général est passé en paramètre.

7.5.1 Séparation

Étant donné une liste de type quelconque, et une condition sur les valeurs des éléments de la liste, calculer deux listes, la première formée des valeurs vérifiant la condition, la seconde formée des autres valeurs.

Spécification *separer : ('a ->bool) ->'a list ->('a list)*('a list)*

Équation de définition Le résultat pour la liste *x::r* et la condition C est un couple de deux listes (ℓ_1,ℓ_2). Pour déterminer la tête et le reste de ℓ_1 et ℓ_2, il y a deux cas possibles :

1. Si x vérifie C alors x est en tête de ℓ_1, et le résultat de la séparation appliquée à r donne le couple ($reste\ de\ \ell_1$,ℓ_2)
2. Si x ne vérifie pas C, idem, en changeant les rôles de ℓ_1 et ℓ_2.

On a donc :

separer C x::r = si C x alors (x::fst (separer C r), snd (separer C r))
sinon (fst (separer C r), x::snd (separer C r))

Cas particulier L'équation ci-dessus ne s'applique pas à une liste vide. Dans ce cas, on a :
separer C []= ([], [])

Codage en CAML Pour éviter d'évaluer deux fois l'expression `separer C r` (chaque évaluation provoque une évaluation récursive, donc un parcours de la liste `r`) on introduit une définition locale.

```
# let rec separer C l =
        if vide l
            then (l, l)
            else let (a,b) = separer C (tl l) in
                        if C (hd l)
                           then (cons (hd l) a, b)
                           else (a, cons x b);;
separer : ('a -> bool) -> 'a list -> 'a list*'a list = <fun>
```

Application. Un couple (x,y) de deux entiers est ordonné si $x \leq y$. Écrire une fonction qui sépare une liste de couples en deux listes de couples, selon que

les couples sont ordonnés ou non. Pour cela, on utilise la fonction générique `séparer`, en précisant la condition C.

Spécification *f: (int*int) list ->(int*int) list * (int*int) list*

Codage CAML

```
# let f l = separer (function (x,y) -> x >= y) l ;;
 f: (int*int) list -> (int*int) list * (int*int) list = <fun>
```

7.5.2 Fusion

On considère deux listes la et lb d'un même type quelconque. Soit C une condition sur les valeurs des éléments de ces listes. Écrire une fonction qui fusionne ces deux listes en une seule selon le procédé suivant : si la (resp. lb) est vide, alors le résultat est lb (resp. la). Sinon, soit xa et xb les têtes respectives de la et lb. Si $C\ xa\ xb$ est vrai, alors xa est mis en tête du résultat, sinon xb est mis en tête du résultat. L'exécution est terminée lorsque les deux listes ont été entièrement parcourues.

Spécification
fusion : ('a ->'a ->bool) ->'a list ->'a list ->'a list

Équation de définition

fusion C xa::ra xb::rb = si C xa xb
alors xa::(fusion C ra (xb::rb))
sinon xb::(fusion C (xa::ra) rb))

Cas particuliers L'équation ci-dessus n'est pas valable si la ou lb sont vides. On a alors :
fusion [] lb = lb et *fusion la [] = la*

Codage CAML

```
# let rec fusion C la lb =
      if vide la
         then lb
         else if vide lb
                then la
                else let xa = hd la and xb = hd lb and ra = tl la and rb = tl lb in
                  if C xa xb
                    then cons xa (fusion C ra lb)
```

```
                    else cons xb (fusion C la rb);;
```
fusion_aux : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list= <fun>

Application: fusion de deux listes d'entiers, triées par valeurs croissantes. C'est une spécialisation de la fonction *fusion*, où C est la comparaison `<=`.

```
# let fus_tri la lb = fusion (prefix <=) la lb;;
```
fus_tri : int list -> int list -> int list = <fun>

7.6 Une optimisation: définitions récursives locales

7.6.1 Exemple de `fusion`

L'examen de la définition précédente montre qu'il est inutile que `fusion` possède l'argument `C`: en effet, cet argument ne change pas d'un appel récursif au suivant; par conséquent, l'identificateur de paramètre formel `C` est lié (à une valeur inconnue) dans le contexte de la définition de `fusion`. On peut donc se contenter de la définition (plus simple) suivante :

```
# let fusion C la lb =
     let rec fusion_aux l1 l2=
        if vide l1
          then l2
          else if vide l2
                then l1
                else let x1 = hd l1 and x2 = hd l2 and r1 = tl l1 and r2 = tl l2 in
                  if C x1 x2
                    then cons x1 (fusion_aux r1 l2)
                    else cons x2 (fusion_aux  l1 r2)
     in fusion_aux  la lb;;
```
fusion : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list= <fun>

La référence à `C` dans l'expression de définition de `fusion_aux` est donc une référence à l'argument formel de la fonction `fusion`, ce qui est correct.

Cette remarque peut être appliquée à toute fonction récursive ayant plusieurs arguments, lorsque certains des arguments ne changent pas d'un appel à l'autre: il est alors inutile de *passer* ces arguments à chaque appel récursif. Pour cela, on introduit des définitions locales de fonctions récursives auxiliaires. Notez bien que cette utilisation de définitions récursives locales n'est en rien indispensable. Elle peut rendre les définitions moins "lisibles" pour les programmeurs, mais elles offrent une amélioration au niveau de l'efficacité des processus d'exécution, en économisant des passages de paramètres

inutiles. C'est le cas notamment de nombre de fonctions génériques, ayant des arguments de type fonctionnel, non modifiés au cours des appels successifs

Ci-dessous, nous ré-écrivons selon cette technique les fonctions récursives vues au chapitres 6 et 7, auxquelles cette optimisation est applicable.

7.6.2 Ré-écriture d'autres exemples

7.6.2.1 Présence d'un caractère dans une chaîne.

```
let present c ch =
    let rec aux ch1
     if vide ch1
         then false
         else if premier ch1 = c
                then true
                else aux (reste ch1)
    in aux ch;;
```
present : char -> string -> bool =<fun>

7.6.2.2 Puissance (méthode rapide).

```
let puiss_dich a n =
   let rec puiss n =
        if n=0
           then 1
           else let x=puiss (n/2) in
                 if n mod 2=0
                      then x * x
                      else x*x*a
   in puiss n;;
```
puiss_dich : int -> int -> int =<fun>

7.6.2.3 Fonction d'accumulation générique

```
# let accumuler e f g l =
    let rec parcours l =
             if vide l
                then e
                else g (f (hd l)) (parcours (tl l))
    in parcours l;;
```
accumuler : 'a -> ('b -> 'c) -> ('c -> 'a -> 'a) -> 'b list -> 'a = <fun>

7.6.2.4 Recherche générique

```
# let recherche C f e l =
    let rec parcours l =
             if vide l
               then e
               else if C (hd l)
                         then f (hd l)
                         else parcours (tl l)
    in parcours l ;;
```
recherche : ('a -> bool) -> ('a -> 'b) -> 'b -> 'a list -> 'b = <fun>

7.6.2.5 Ajout d'un élément en fin de liste

```
# let append l v =
     let rec parcours l =
             if vide l
               then  ->  cons v l
               else cons (hd l) (parcours (tl l))
     in parcours l;;
```
append : 'a list -> 'a -> 'a list = <fun>

7.6.2.6 La fonction générique `map`

```
# let map f l =
     let rec parcours l =
             if vide l
                then l
                else cons (f (hd l)) (parcours (tl l))
     in parcours l;;
```
map : ('a -> 'b) -> 'a list -> 'b list = <fun>

7.6.2.7 Le parcours `faire_tantque`

```
# let faire_tantque C f l =
      let rec parcours l =
             if vide l
                then l
                else if C (hd l)
                        then cons (f (hd l)) (parcours (tl l))
                        else []
```

```
      in parcours l;;
```
faire_ttque : ('a -> bool) -> ('a -> 'b) -> 'a list -> 'b list = <fun>

7.6.2.8 Intervalle

```
# let intervalle a b =
      let rec aux inf =
              if inf>b
                  then []
                  else cons inf (aux (inf+1))
      in aux a;;
```
intervalle : int -> int -> int list = <fun>

7.6.2.9 Valeurs successives d'une suite

```
# let suite x f n =
     let rec aux prem n=
         if n=0
            then []
            else cons prem (aux (f prem) (n-1))
     in aux x n;;
```
suite : 'a -> 'a -> 'a -> int -> 'a list

7.6.2.10 Séparation

```
# let separer C l =
     let rec parcours l =
          if vide l
             then (l, l)
             else let (a,b) = parcours (tl l) in
                     if C (hd l l)
                          then (cons (hd l) a, b)
                          else (a, cons (hd l) b)
     in parcours l;;
```
separer : ('a -> bool) -> 'a list -> 'a list'a list =*
<fun>

Chapitre 8

Calculs itératifs : récursivité terminale

Dans ce chapitre, on aborde une autre méthodologie de construction de fonctions récursives moins immédiate pour le programmeur, en ce sens qu'elle s'éloigne de la définition récursive "naturelle" qui a guidé l'écriture des fonctions récursives tout au long des chapitres 6 et 7. Cependant, cette nouvelle approche est utile, pour au moins trois raisons :

1. elle peut améliorer l'efficacité des processus d'exécution,
2. elle permet parfois de résoudre plus simplement certains problèmes,
3. elle prépare aux méthodes de construction d'itération que l'on rencontre en programmation impérative.

Nous abordons successivement ces trois aspects.

8.1 Amélioration de l'efficacité

Dans ce paragraphe, nous considérons des exemples qui peuvent être exprimés par une définition récursive classique (ils ont été vus aux chapitres précédents), mais nous les abordons avec l'approche itérative, afin d'en améliorer l'efficacité. Le premier exemple : calcul de a^n, pour a entier et n entier non négatif, va être détaillé à l'extrême, afin d'expliquer en détail la méthode de construction utilisée. Les exemples suivants seront traités plus brièvement.

La définition récursive "naturelle $a^n = a.a^{n-1}$ (avec $a^0 = 1$) nous a conduit à la fonction suivante :

```
# let rec puiss a n =
     if n = 0
        then 1
        else a*(puiss a (n-1));;
puiss : int -> int -> int=<fun>
```

Examinons le déroulement du processus d'évaluation de l'expression *puiss 2 6*:

```
puiss 2 6 =
2*(puiss 2 5) =
2*2*(puiss 2 4) =
2*2*2*(puiss 2 3) =
2*2*2*2*(puiss 2 2) =
2*2*2*2*2*(puiss 2 1) =
2*2*2*2*2*2*(puiss 2 0) =
2*2*2*2*2*2*1 =
2*2*2*2*2*2 =
2*2*2*2*4 =
2*2*2*8 =
2*2*16 =
2*32 =
64
```

On constate que le nombre de valeurs intermédiaires à "stocker" est proportionnel à n. Cela est du au fait que, lors de l'évaluation de l'expression *puiss a k*, l'évaluation de l'appel récursif – ici l'appel à *puiss a (k-1)* – n'est pas la dernière opération: il faut ensuite multiplier le résultat de cet appel récursif par a. Et ceci pour chaque expression correspondant à une valeur de k allant de n à 1. On dit que la fonction *puiss* est à *récursivité non terminale.*

Essayons maintenant de construire une fonction *puiss_iter* basée sur la définition "en extension":

$$a^n = \underbrace{a * a * \cdots * a}_{n \text{ fois}}$$

Il s'agit d'effectuer les calculs successifs a, $a * a$, $a * a * a, \cdots$ en comptant le nombre de fois où l'on multiplie le résultat courant par a. Pour mettre en œuvre ce principe, on utilise un raisonnement basé sur la notion de *suites de valeurs*, régies par une équation d'évolution. Dans cet exemple, nous avons deux suites :

```
   - La suite ''compteur''              0,  1,  2,  ...  , k, ...
   - La suite des valeurs calculées     1,  a, a*a, ...  ,a*a*...*a,
...
```

Désignons la première par *compt* et la seconde par *acc*. On constate qu'à chaque étape du calcul, les valeurs des deux suites vérifient une relation, appelée pour cette raison **invariant du calcul**, qui est la suivante :

$$acc = \underbrace{a * \cdots * a}_{compt \text{ fois}} = a^{compt}$$

Le résultat du calcul sera donc exprimé par *acc* lorsque la valeur de *compt* sera égale à n. On a mis en évidence la **condition d'arrêt** :

$compt = n$, n étant une donnée

On constate de plus que l'on passe d'une valeur du couple *compt*, *acc* à la suivante par la transformation :

$$\begin{pmatrix} compt \\ acc \end{pmatrix} \longrightarrow \begin{pmatrix} compt + 1 \\ a * acc \end{pmatrix}$$

Ceci met en évidence la **loi d'évolution** ou **progression**.

Enfin, initialement, on peut prendre $compt = 0$ et $acc = 1$, qui sont des valeurs *connues* et *satisfaisant l'invariant* : $1 = a^0$. On a donc mis en évidence les **valeurs initiales**.

Ces quatre éléments : **invariant, condition d'arrêt, progression, valeurs initiales** sont constitutifs de toute construction *itérative* permettant d'exprimer la répétition d'un calcul "un certain nombre de fois". Il nous reste à expliquer comment exploiter ces éléments pour obtenir la définition correspondante, en CAML, de la fonction *puiss_iter*.

D'après la condition d'arrêt, un embryon de définition est le suivant:
`puiss_iter a n` = *valeur de* `acc` *lorsque* `compt=n`

Il faut donc trouver une expression donnant cette valeur de `acc`. Cela va être réalisé à l'aide d'une fonction exprimant la progression. Cette progression nous indique que :

```
      1 est la valeur de acc lorsque compt = 0
      a  ‘‘      ‘‘      acc    ‘‘   compt = 1
      .                   .           .     .
      .                   .           .     .
      .                   .           .     .
   si  x est la valeur de acc lorsque compt = k, alors
     a*x  ‘‘       ‘‘     acc     ’   compt = k+1
```

L'évolution des deux suites est donc engendrée par le processus d'exécution de la fonction, que nous désignons par `progression`, telle que :

`progression compt acc = progression (compt + 1) (a*acc)`

En CAML, cette fonction est définie récursivement, avec sa condition d'arrêt, dans un contexte où `a` et `n` sont liés :

```
let rec progression compt acc =
              if compt = n
                 then acc
                 else progression (compt+1) (a*acc)
```

Le processus d'évaluation de l'expression `progression 0 1` (les valeurs initiales respectives de `compt` et `acc`) va alors produire la suite des valeurs

`progression 0 1 = progression 1 a = progression 2` a^2 `= ...progression n` a^n.

Finalement, la définition de *puiss_iter* va englober la définition de la fonction *progression* comme définition locale, ce qui donne le texte final :

```
# let puiss_iter a n =
    let rec progression compt acc =
              if compt = n
                 then acc
                 else progression (compt+1) (a*acc)
    in progression 0 1 ;;
puiss_iter : int -> int -> int = <fun>
```

Déroulons maintenant le processus d'évaluation de l'expression *puiss_iter 2 6* :

```
puiss_iter 2 6 = progression 0 1
               = progression 1 2
               = progression 2 4
               = progression 3 8
               = progression 4 16
               = progression 5 32
               = progression 6 64
               = 64
```

On constate que le processus n'a besoin de stocker, à chaque étape, qu'un nombre *constant* de valeurs (ici, les valeurs des deux paramètres effectifs de la fonction *progression*). Cela est dû au fait que, dans l'évaluation de l'expression *progression compt acc*, l'évaluation de l'appel récursif – ici l'appel à *progression (compt+1) (a*acc)* – est la *dernière* opération. On dit que la fonction *progression* est à *récursivité terminale.*

Exemple 2 : renversement d'une liste

Étant donnée une liste ℓ d'éléments de type quelconque, définir une fonction *miroir* qui rend la liste des valeurs de ℓ dans l'ordre inverse : *[a; b; c; d] -> [d; c; b; a]*

Solution récursive directe

Appliquant la méthodologie du chapitre 7, on obtient immédiatement l'équation :

miroir x::r = mettre_en_queue (miroir r) x, où *mettre_en_queue* est la fonction qui, appliquée une liste et une valeur , met cette valeur en queue de liste.

La fonction *mettre_en_queue* elle-même est récursive.

Voici les textes en CAML :

```
# let rec mettre_en_queue l v =
      if vide l
```

```
        then cons v l
        else cons (hd l) (mettre_en_queue (tl l) v);;
mettre_en_queue : 'a list -> 'a -> 'a list

# let rec miroir l =
    if vide l
       then l
       else mettre_en_queue (miroir (tl l)) (hd l);;
miroir : 'a list -> 'a list
```

On peut ici réaliser la complexité de cette solution, puisque chaque évaluation de *miroir* provoque une évaluation de *mettre_en_queue*, qui nécessite elle-même un parcours exhaustif de sa liste argument. Le processus d'évaluation de l'expression *miroir [a;b;c]*, par exemple, est le suivant (on abrège *mettre_en_queue* par *meq*):

```
miroir [a; b; c] =
meq (miroir [b; c]) a =
meq (meq (miroir [c]) b) a =
meq (meq (meq (miroir []) c) b) a =
meq (meq (meq [] c) b) a =
meq (meq [c] b) a =
meq (c::(meq [] b) a =
meq (c::[b]) a =
c::(meq [b] a) =
c::(b::(meq [] a)) =
c::(b::[a]) =
c::[b; a] =
[c; b; a]
```

Solution itérative

On considère deux suites, toutes deux constituées de listes; la première, désignée par *arg* est la suite de listes obtenues en ôtant successivement les têtes à partir de la liste donnée, la seconde, notée *res*, est la suite de listes obtenues en ajoutant successivement en tête les valeurs retirées de la première, à partir de la liste vide. On a donc l'évolution suivante :

```
arg          res
-----------------
[a;b;c;d]    []
[b;c;d]      [a]
[c;d]        [b;a]
[d]          [c;b;a]
[]           [d;c;b;a]
```

Invariant: *miroir(res)@arg = l* où *l* est la liste donnée, et *@* exprime la concaténation des listes.

Condition d'arrêt et résultat: `arg=[]` résultat : `res`. En effet, on a alors, d'après l'invariant : `miroir(res) = l` ce qui est équivalent à `res = miroir(l)`.

Progression

$$\begin{pmatrix} arg = x :: r \\ res \end{pmatrix} \longrightarrow \begin{pmatrix} r \\ x :: res \end{pmatrix}$$

Valeurs initiales `arg=l, res = []`

Codage CAML

```
# let miroir_iter l =
    let rec progression arg res =
          if vide arg
             then res
             else progression (tl l) (cons (hd l) res)
    in progression l [] ;;
```

miroir_res : 'a list -> 'a list = <fun>

Déroulons le processus d'évaluation de l'expression `miroir_iter [a; b; c]` :

```
miroir_iter [a; b; c] =
progression [a; b; c] [] =
progression [b; c] a::[] =
progression [c] b::[a] =
progression [] c::[b; a] =
[c ; b; a]
```

8.2 Faciliter la résolution de certains problèmes

Dans ce paragraphe, nous examinons deux exemples de fonctions dont la définition récursive directe, même si elle reste possible, s'avère plus difficile à concevoir, pour le programmeur, qu'une définition itérative.

8.2.1 Rang

Il s'agit de trouver le *rang* du premier élément d'une liste de type quelconque vérifiant une condition donnée. La **spécification** de cette fonction est donc :

rang : ('a ->bool) -> 'a list ->int

Si aucun élément de la liste ne vérifie la condition, alors le résultat doit être égal à 0.

Quelles sont les difficultés d'une définition récursive directe? Apparemment, il est facile d'établir une équation :

rang C x ::r = **si** C x **alors** 1
sinon 1 + *rang C r*

Cette équation n'est pas valable lorsque la liste est vide. Dans ce cas, quelle définition donner? Dans une liste vide, aucun élément ne vérifie la condition, donc le résultat doit être 0. Selon cette analyse, le codage CAML sera le suivant :

```
# let rec rang C l =
           if vide l
              then 0
              else if C (hd l)
                      then 1
                      else 1+rang C (tl l);;
rang : ('a -> bool) -> 'a list -> int = <fun>
```

Cependant, cette définition n'est pas correcte, comme le montre le processus d'exécution ci-dessous, avec `C x = x<0` (rang du premier élément négatif dans une liste d'entiers):

```
rang C [2 ; 6 ; 1] =
1 + rang C [6 ; 1] =
1 + 1 + rang C [1] =
1 + 1 + 1 + rang C [] =
1 + 1 + 1 + 0 =
3
```

En fait, le problème vient de ce que la liste vide constitue une condition d'arrêt correspondant au cas où aucun élément ne vérifie la condition (cas de parcours exhaustif). Dans ce cas, l'erreur vient de ce que la définition proposée incrémente le résultat obtenu lors de chaque retour d'appel. Il faudrait donc disposer d'une information supplémentaire permettant d'effectuer un test, dans l'expression du résultat, selon qu'un appel sur la liste vide a eu lieu ou non. Plutôt que d'essayer de résoudre directement ce problème, nous allons examiner une solution itérative.

Soit $\ell = [x_1; x_2; \ldots; x_n]$ la liste argument, dans le cas où elle n'est pas vide. Le parcours de cette liste avec gestion du rang met en œuvre trois suites :

rang :	1	2	...	$k-1$	k	...
tete :	x_1	x_2	...	x_{k-1}	x_k	...
reste :	$[x_2; \ldots; x_n]$	$[x_3; \ldots; x_n]$	...	$[x_k; \ldots; x_n]$	$[x_{k+1}; \ldots; x_n]$	...

Invariant: $tete = x_{rang}$, $reste = [x_{rang+1}; \ldots; x_n]$, $1 \leq rang \leq n$

Arrêt: *C tete* (résultat *rang*) ou $reste = []$ (résultat 0)

Progression: $\begin{pmatrix} rang \\ tete \\ reste = x :: r \end{pmatrix} \longrightarrow \begin{pmatrix} rang + 1 \\ x \\ r \end{pmatrix}$

Valeurs initiales: $rang = 1$, $tete = x_1$, $reste = [x_2; \ldots; x_n]$

Codage CAML

```
# let rang_iter C l =
     if vide l
        then 0
        else
          let rec progression rang tete reste =
                 if vide reste
                      then 0
                      else if C (hd reste)
                              then rang
                              else progression (rang+1) (hd reste) (tl reste)
        in progression 1 x1 r1 ;;
rang_iter : ('a -> bool) -> 'a list -> int = <fun>
```

Exemples de processus d'exécution :

```
rang_iter (function x -> x<0) [5 ; 7; 4 ; 8] =
progression 1 5 [7 ; 4 ; 8] =
progression 2 7 [4 ; 8] =
progression 3 4 [8] =
progression 4 8 [] =
0

rang_iter (function x -> x<0) [5 ; 7; 4 ; -1 ; 8] =
progression 1 5 [7 ; 4 ; -1 ; 8] =
progression 2 7 [4 ; -1 ; 8] =
progression 3 4 [-1 ; 8] =
progression 4 (-1) [8] =
4
```

8.2.2 Problème du Comité

Un comité est composé membres, chaque membre étant représentant d'un parti **A** ou d'un parti **B**. Il y a, initialement, nA (resp. nB) membres du parti **A** (resp **B**). Le renouvellement du comité, annuel, s'effectue par remplacement aléatoire d'un des membres, selon l'algorithme suivant : un premier tirage au sort détermine le parti du membre sortant, puis un deuxième tirage (indépendant du premier) détermine le parti du membre entrant. Un des

membres du parti sortant est alors remplacé par un nouveau membre du parti entrant (le nombre total reste constant, soit N). Le problème consiste à simuler l'évolution de ce comité jusqu'à ce qu'un des deux partis soit éliminé; le résultat est alors le couple (nom du parti éliminé, nombre d'années).

Les deux tirages sont effectués selon une loi pondérée par le nombre respectif de représentants des deux partis, avant remplacement. Autrement dit, le résultat de chacun des tirages est une variable aléatoire Y pouvant être égale à **A** ou **B**, avec les probabilités

$$\mathbf{P}(Y = \mathbf{A}) = \frac{x_A}{x_A + x_B},\ \mathbf{P}(Y = \mathbf{B}) = \frac{x_B}{x_A + x_B}$$

où x_A et x_B désignent respectivement le nombre de membres des partis **A** et **B** au moment du tirage. En CAML, on dispose de la fonction prédéfinie `random__int : int ->int` qui délivre un entier tiré au hasard selon la loi uniforme entre 0 et son argument. Le tirage peut donc être simulé par l'exécution de la fonction définie ci-dessous :

```
# let tirage xA xB = let x=random__int (xA+xB-1) in
                          if x<xA then 'A' else 'B';;
 tirage : int -> int -> char = <fun>
```

L'évolution du comité est modélisée à l'aide de trois suites :

xA	=	nombre de membres du parti **A**
xB	=	nombre de membres du parti **B**
compt	=	nombre d'années

Invariant:

xA	=	nombre de membres du parti **A** l'année *compt*
xB	=	nombre de membres du parti **B** l'année *compt*
$xA + xB$	=	N

Condition d'arrêt : $xA = 0$ (résultat ('A', *compt*) **ou** $xB = 0$ (résultat ('B',*compt*)).

Progression : soit $c1$ et $c2$ les résultats des deux tirages au sort successifs. Selon les valeurs du couple ($c1$,$c2$) (il y en a quatre possibles) on incrémente (resp. décrémente) de 1 les valeurs de xA et xB; dans tous les cas, *compt* augmente de 1.

Valeurs initiales : $xA = nA$, $xB = nB$,$compt = 0$

Codage en CAML

```
# let comite nA nB =
   let rec progression compt xA xB =
```

```
      if xA = 0
         then  ('A', compt)
         else if xB = 0
                 then ('B', compt)
                 else
                   let c1=tirage xA xB and c2=tirage xA xB in
                   match (c1, c2) with
                     if c1='A' & c2='A' or c1='B' & c2= 'B'
                        then progression (compt+1) xA xB
                        else if c1='A' & c2='B'
                                then progression (compt+1) (xA-1) (xB+1)
                                else (* c1='B', c2='A' *)
                                     progression (compt+1) (xA+1) (xB-1)
   in progression 0 nA nB;;
comite : int -> int -> int*char = <fun>
```

Remarque : ce programme est un exemple qui se prêterait bien à l'utilisation des filtres. Bien que nous n'ayons pas souhaité retenir ce style de programmation dans ce cours, on montre ci-dessous ce que ça donnerait (ceci illustre la concision de style en CAML; de plus, le programme est plus compréhensible).

```
# let comite nA nB =
    let rec progression compt = function
       (0, _) -> ('A', compt)
      | (_,0) -> ('B', compt)
      | (xA,xB) ->
           let c1=tirage xA xB and c2=tirage xA xB in
             match (c1, c2) with
               ('A','A') | ('B','B') -> progression (compt+1) (xA,xB)
                         ('A', 'B') -> progression (compt+1) ((xA-1), (xB+1))
                         ('B', 'A') -> progression (compt+1) ((xA+1), (xB-
1))
   in progression 0 (nA, nB);;
 > Toplevel input:
>.....match (c1,c2) with
>           ('a', 'a') — ('b','b') -> progression (compt+1) (xa, xb)
>         —('a','b') -> progression (compt+1) ((xa-1),(xb+1))
>         —('b','a') -> progression (compt+1) ((xa+1),(xb-1))
> Warning: pattern matching is not exhaustive

comite : int -> int -> int*char = <fun>
```

(le message *warning* n'est qu'un avertisssement, indiquant que tous les cas de valeurs de couples `(c1,c2)` n'ont pas été prévus. Si on veut éviter ce message, il suffit de rajouter un cas `| _ -> failwith "tirage imprevu!"`).

On peut provoquer un bel affichage du résultat, en définissant la fonction :

```
# let aff_comite nA nB = let (c,n) = comite nA nB in
"le parti "^(char_for_read c)^" s'eteint au bout de"
^(string_of_int n)^" annees";;
 aff_comite : int -> int -> string =<fun>
```

8.3 Simulation d'itérations

8.3.1 Boucle

Dans les cours de programmation impérative, on introduit la notion d'itération, qui peut être schématisée de la manière suivante : Soit x une valeur initiale. Soit *arret* un test vérifiant si x convient. Si c'est le cas, le résultat vaut x; sinon, soit *nouveau* une fonction qui, appliquée à x donne une nouvelle valeur. On remplace x par $nouveau(x)$ et on recommence.

Ce schéma peut être exprimé par une fonction *iterer*, ayant comme arguments la valeur *init:'a*, la condition d'arret *arret:'a ->bool* et la fonction de transformation *nouveau:'a -> 'a*. Remarquez que l'on retrouve les éléments **valeurs initiales, condition d'arrêt** et **progression** de la méthodologie itérative. On en déduit facilement la fonction générique *itérer* :

```
# let iterer init arret nouveau =
    let rec progression x = if arret x
                            then x
                            else progression (nouveau x)
    in progression init;;
 iterer : 'a -> ('a -> bool) -> ('a -> 'a) -> 'a =
<fun>
```

Application: calcul approché de la racine carrée On montre, en analyse numérique, que, si x est une valeur approchée de $\sqrt{a}$, où a est un nombre positif ou nul, alors $\frac{1}{2}\left(x + \frac{a}{x}\right)$ est une approximation plus précise de $\sqrt{a}$. On donne le nombre a, et un réel ε, et on veut calculer une valeur x telle que $\mid x * x - a \mid \leq \varepsilon$ (x est une approximation à ε près de $\sqrt{a}$). On peut prendre a comme approximation initiale.

Il suffit de spécialiser la fonction générique `iterer` en choisissant convenablement les arguments.

```
# let rac_carree a epsil =
    let arret x = abs_float(x*.x -.a)<=.epsil and
        nouveau x = (x+a/.x)/.2. in
```

```
    iterer a arret nouveau ;;
rac_carree : float -> float -> float = <fun>
```

8.3.2 Généralisation : calculs sur les suites

Soit x_0,x_1,...,x_k,... une suite de valeurs de type quelconque, telle que :

- le terme initial x_0 a une valeur donnée,
- le terme de rang k dépend du terme de rang $k-1$ et de la valeur entière k : $x_k = f(x_{k-1},k)$ où f est une fonction donnée

Soit enfin C une condition définie sur les couples (x,k) où x est une valeur du même type que les éléments de la suite, et k est entier.

On veut calculer la *valeur* x et le *rang* k du premier terme de la suite tel que la condition $C(x,k)$ soit vérifiée.

Soit *val_et_rang* le nom de la fonction Son type est donc :

*'a ->('a ->int ->'a) ->('a ->int ->bool) ->'a*int* (les arguments sont, dans l'ordre, x_0, f, C).

Les deux suites mises en jeu dans ce calcul sont :

- Le compteur *rang*, prenant les valeurs successives :
  0,1,...,$k-1$,k,...
- La suite *val*, prenant les valeurs successives :
  x_0,x_1,...,x_{k-1},x_k,...

Invariant : $val = x_{rang}$
Condition d'arrêt : $C(val,rang)$; le résultat est alors le couple $(val,rang)$

Progression :

$$\begin{pmatrix} rang \\ val \end{pmatrix} \longrightarrow \begin{pmatrix} rang+1 \\ f(val,rang) \end{pmatrix}$$

Valeurs initiales : $rang = 0$,$val = x_0$

Codage en CAML :

```
# let val_et_rang x0 f C =
    let rec progression rang val =
       if C val rang
            then (val,rang)
            else progression (rang+1) (f val rang)
    in progression 0 x0 ;;
val_et_rang : 'a -> ('a -> int -> 'a) -> ('a -> int -> bool) -> 'a * int = <fun>
```

Exemples d'applications

Capital Soit un capital a, placé au taux annuel t%. Au bout de combien d'années ce capital aura-t-il dépassé une valeur S donnée, et quelle sera alors sa valeur?

Les valeurs successives du capital constituent une suite, de valeur initiale a et évoluant selon la loi $x \longrightarrow x + x * \frac{t}{100}$.

La condition d'arrêt est $x \geq S$.

On spécialise donc la fonction générique `val_et_rang` en définissant correctement les paramètres effectifs :

```
# let capital a t S =
    let f v r = v+.v*.t/.100. (* f ne dépend pas de r*)
    and C v r = v>=.S  (* C ne dépend pas de r*)
    in val_et_rang a f C;;
capital : float -> float -> float -> float*int =<fun>
```

Quand est-ce que mon capital de 10000F, placé à 4.5%, aura doublé?

```
# let capital_double a t = capital a t (2*.a) in
    capital_double 10000. 4.5;;
- : float * int = 20223.7015303, 16
```

Chapitre 9

Vers la programmation impérative

9.1 Modification du contexte : affectation

9.1.1 Modèle par substitution et modèle à état modifiable

Jusqu'à présent, les programmes que nous avons écrit nous ont permis d'exprimer des algorithmes, dont l'exécution se contentait d'*évaluer* des expressions, en fonction du contexte. Nous avons constaté que l'évaluation d'une expression est basée sur la *lecture* du contexte, mais ne provoque aucune modification des valeurs contenues dans celui-ci. Le modèle utilisé s'appelle *modèle d'évaluation par substitution* car, effectivement, le seul mécanisme mis en jeu lors d'évaluations d'expressions est celui qui consiste à remplacer chaque identificateur présent dans l'expression par sa valeur correspondante dans le contexte (selon la règle de recherche dans le contexte, expliquée notamment aux chapitres 3 et 5). Comme il faut bien pouvoir construire un contexte (installer de nouvelles liaisons) ou le détruire (supprimer des liaisons), les opérations adéquates sont fournies à cet effet:

- L'installation d'une nouvelle liaison est réalisée explicitement par la construction **let** `id = expr` (voire **let rec** ...),
- La suppression d'une liaison est implicite, lors de fins d'évaluations pour les liaisons locales, lors de la fin de la session pour les liaisons globales.

Cependant, et c'est là le trait essentiel du modèle par substitution, aucune opération ne permet de *modifier* une liaison en place.

Il existe un autre modèle, dit *modèle à état modifiable* (ou plus brièvement *modèle à état*, ou *modèle à environnement*, ou *modèle à effets*) dans lequel de telles opérations sont possibles. Bien que le langage CAML fournisse les outils permettant d'exprimer de telles opérations, nous allons – pour des raisons de simplicité – décrire le modèle à état dans un formalisme *qui n'est pas celui*

du langage CAML. En effet, CAML fournit un mécanisme de modification indirecte, exprimant le concept de *références*. Ce concept sera étudié dans les cours de programmation impérative (avec d'autres langages). Dans un premier temps, nous décrivons un mécanisme de modification directe, plus simple à appréhender.

9.1.2 Affectation

Dans ce qui suit, nous considérons une machine "CAML-étendue", hypothétique, qui, outre les fonctionnalité de la machine CAML que nous connaissons, possède une opération supplémentaire. Nous introduisons un nouveau symbole qui, répétons le, **n'appartient pas au langage CAML**. Ce symbole, ←, exprime une *action*, ayant pour *effet* de modifier la valeur liée à un identificateur du contexte. Ainsi, l'exécution de la phrase:

```
x ←expr
```

effectue deux choses :

1. elle évalue l'expression `expr` dans le contexte courant : soit `val` la valeur obtenue (en supposant que l'évaluation réussisse);
2. dans la liaison identifiée par l'identificateur `x`, la valeur liée à `x` est effacée et remplacée par la nouvelle valeur `val`.

et la machine CAML-étendue affiche l'information relative à la liaison modifiée.

Noter que le mécanisme d'évaluation de l'expression, comme le procédé de recherche dans le contexte, sont identiques à ceux que nous avons vus jusqu'à présent : mécanisme standard du modèle par substitution pour l'évaluation, et recherche dans le contexte de la plus *récente* liaison d'identificateur `x`.

Une opération telle que celle que nous venons de décrire s'appelle une *affectation*. Pour qu'elle réussisse, il faut donc plusieurs conditions (les deux premières sont celles du modèle par substitution : nous ne faisons que les rappeler; la troisième est nouvelle, spécifique de l'affectation) :

1. celles conditionnant le succès de l'évaluation de l'expression `expr` (identificateurs liés dans le contexte et cohérence de type),
2. celle conditionnant le succès de la recherche de l'identificateur `x` dans le contexte,
3. le *type* du résultat de l'évaluation et le type de `x` dans le contexte doivent être les mêmes (rappelons que le champ *type* d'une liaison décrit les types de valeurs qui peuvent être liés à `x`).

Exemple Dans le contexte `[x~56 ; y~'0' ; x~"toto"]` on effectue l'action:

```
x ←int_of_char y - x.
```

Le contexte obtenu est alors le suivant (sachant que `int_of_char '0' = 48`) :

```
[x~-8 ; y~'a' ; x~"toto"]
```

(la machine "CAML-étendue" affiche la réponse *x : int = -8*)

On constate que l'exécution de cette opération a eu un *effet* sur le contexte en place. C'est pourquoi une telle exécution s'appelle une *action*, et la phrase qui la décrit dans le langage s'appelle une *instruction* (au sens administratif ou militaire du terme) : on donne un ordre, ou instruction, au processus.

Nous retenons les définitions suivantes, permettant de bien distinguer les notions relatives au programme (notions statiques) de celles relatives aux processus et contextes d'exécution (notions dynamiques) :

- On appelle **effet** (sur un identificateur présent dans un contexte) le remplacement d'une valeur liée à cet identificateur par une nouvelle valeur.
- Une **action** est une opération effectuée par un processus d'exécution, produisant un *effet*.
- Une **instruction** est une construction de langage permettant d'exprimer une *action*.

9.1.3 Les dangers de l'affectation

Avec cette nouvelle opération, la simplicité et la sécurité du modèle par substitution sont quelque peu perturbés, comme vont le montrer les exemples ci-après. Considérons les deux définitions de fonctions suivantes, la première étant une définition classique en CAML, la deuxième une définition faisant intervenir une instruction d'affectation (syntaxiquement incorrecte en CAML, du moins sous cette forme, rappelons le encore!) :

```
# let soustraire x y = y-x;;
 soustraire : int -> int -> int = <fun>

# let soustraire_mut x y = y ← y-x;;
 soustraire_mut : int -> int -> int = <fun>
```

Essayons maintenant deux applications successives de la fonction `soustraire`, puis deux applications successives de la fonction `soustraire_mut`:

```
let x=100 and y=500;;
x : int = 100
y : int = 500

# soustraire x y;;
- : int = 400

# soustraire x y;;
- : int = 400

# y;;
```

```
- : int = 500

# soustraire_mut x y ;;
  y : int = 400

# soustraire_mut x y ;;
 y : int = 300

y ;;
 - : int = 300
```

Ainsi, l'application de la fonction `soustraire` ne modifie pas le contexte, et deux applications successives avec les mêmes arguments donnent les mêmes résultats. Par contre, chaque application de la fonction `soustraire_mut` modifie le contexte (en l'occurrence la valeur liée à `y`) et cet effet se répercute lors d'appels successifs, puisque deux appels successifs avec les mêmes arguments formels donne deux résultats différents. Dans ce dernier cas, on s'aperçoit que des applications successives de la fonction ont un effet *cumulatif* ou *effet mémoire*, source de résultats surprenants si le phénomène n'est pas bien maîtrisé *par le programmeur*. Mais il y a pire :

```
# let y=500 in (soustraire_mut 100 y) + y ;;
 - : int = ???
```

Le résultat de l'évaluation, ici, **dépend de l'ordre** dans lequel les deux arguments de l'opérateur `+` vont être évalués. En effet, si l'opérande gauche est évalué *avant* le droit, on obtiendra `400 + 400` c'est-à-dire `800`, tandis que si l'opérande droit est évalué *avant* le gauche, on obtiendra `500 + 400` c'est-à-dire `900`. *La commutativité de l'addition n'est donc plus assurée*! Cet exemple illustre parfaitement le danger qu'il peut y avoir à définir des fonctions produisant des effets, lorsque ces fonctions sont appelées lors d'évaluations d'expressions: les effets en question peuvent être "pervers".

9.1.4 Composition d'actions : la séquentialité

Les modèles offrant la possibilité d'actions (telles que l'affectation) doivent aussi fournir les moyens de composer entre elles plusieurs actions, au moyen d'opérateurs de composition. Bien que l'étude détaillée de ceux-ci relève d'un cours d'initiation à la programmation selon l'approche impérative (qui fera l'objet d'un autre document), nous allons introduire ici le plus élémentaire d'entre eux, à savoir l'opérateur de *séquentialité*. Notez que cet opérateur existe dans le langage CAML, indépendamment de l'existence d'instructions produisant des actions (telles que l'instruction d'affectation), mais son utilisation ne présente pas d'intérêt lorsqu'on se limite au modèle d'évaluation par substitution.

L'opérateur de séquentialité est désigné au moyen du symbole `;` (point-virgule), et il possède la signification suivante : soit I_1 `et` I_2 deux instructions, telles que l'exécution de I_1 le contexte $\mathcal{C}_0$ produit le contexte $\mathcal{C}_1$ et la valeur v_1, et l'exécution de I_2 le contexte $\mathcal{C}_1$ produit le contexte $\mathcal{C}_2$ et la valeur v_2. Alors I_1 `;` I_2 est l'instruction dont l'exécution dans le contexte $\mathcal{C}_0$ produit le contexte $\mathcal{C}_2$ et la valeur v_2:

$$\mathcal{C}_0 \xrightarrow[v_1]{I_1} \mathcal{C}_1 \xrightarrow[v_2]{I_2} \mathcal{C}_2 \qquad \equiv \qquad \mathcal{C}_0 \xrightarrow[v_2]{I_1;I_2} \mathcal{C}_2$$

L'opérateur de séquentialité est associatif, c'est-à-dire : I_1 `;` `(`I_2 `;` I_3`)` est équivalent (produit le même contexte et la même valeur) que `(`I_1 `;` I_2`)` `;` I_3 si bien que l'on peut tout simplement écrire I_1 `;` I_2 `;` I_3.

Exemple: dans le contexte `[x~1 ; y~2]`, la séquence suivante :
`x` ←`x + y ; y` ←`x - y ; x` ←`x - y` va produire le contexte
`[x~2 ; y~1]`

L'opérateur de séquentialité peut aussi être utilisé pour composer des opérations d'évaluation sans effet. Dans ce cas, le contexte n'est pas modifié, et la valeur obtenue est le résultat de la dernière évaluation. Les deux exemples ci-dessous sont *légaux* en CAML (bien que leur intérêt, ici, ne soit pas évident!) :

```
# let x=1 and y=2 in x ; y ;;
 - : int = 2
# let x=1 and y=2 in y ; x ;;
 - : int = 1
```

En fait, la séquentialité n'a d'intérêt que lorsqu'elle compose des opérations dont au moins l'une est une action. Mais, ce faisant, les dangers déjà mentionnés à propos des opérations à effet – telles que l'affectation – sont encore plus grand, comme le montre l'exemple suivant [1] :

```
# let ajouter_mut  x y = x ← x+y ;;
 ajouter_mut : int -> int -> int = <fun>
# let soustraire_mut x y = y ← y-x ;;
 soustraire_mut : int -> int -> int = <fun>
# let x=100 and y=500 ;;
 x : int = 100
 y : int = 500
# ajouter_mut  x y  ;  soustraire_mut x y ;;
 - : int = -100
# (x, y) ;;
 - : int*int = 600, -100
```

1. rappel : qui n'est pas correct en CAML.

```
# let x=100 and y=500;;
 x : int = 100
 y : int = 500
# soustraire_mut  x y  ;  ajouter_mut x y ;;
 - : int = 500
# (x, y);;
 - : int*int = 500, 400
```

9.2 Fonctions et procédures : les entrées-sorties

9.2.1 Fonctions pures et procédures pures

Dans le modèle par substitution, un algorithme est exprimé sous forme d'une expression, qui peut être abstraite sous la forme d'une fonction, c'est-à-dire d'une expression dont l'évaluation est paramétrée. Un appel de fonction provoque alors, à l'exécution, l'évaluation de cette expression (expression de définition de la fonction) dans laquelle certains paramètres formels – identificateurs – ont été remplacés par des valeurs. Une telle opération ne produit pas d'effet, mais rend un résultat: on parle alors de *fonction pure*.

Au contraire, nous avons vu que le modèle à état permet d'introduire des instructions. L'exécution d'une instruction est une action, qui produit des effets, mais peut aussi rendre un résultat. Dans un tel modèle, un algorithme peut être exprimé comme un enchaînement d'instructions, qui peut être abstrait sous la forme de ce que l'on appelle une *procédure*, c'est-à-dire un enchaînement d'instruction éventuellement paramétré. Un appel de procédure provoque alors, à l'exécution, l'enchaînement des actions correspondant aux instructions, c'est-à-dire une action globale, produisant des effets. Si ce processus ne rend pas de valeur, alors on parle de *procédure pure*.

La plupart des langages de programmation mettent en œuvre les deux modèles par substitution et à états, et offrent les notions de fonctions et de procédures (groupées sous divers noms génériques comme *sous-programme* ou *routine* ou *méthodes*, etc.). Idéalement, seuls les concepts de *fonction pure* et *procédure pure* devraient être disponibles, c'est-à-dire qu'une fonction ne devrait pas permettre de programmer des actions, et une procédure ne devrait pas rendre de résultat. Malheureusement, cette discipline reste souvent de la responsabilité des programmeurs, car de nombreux langages n'imposent pas ce type de restrictions à l'usage des fonctions ou des procédures. En particulier, la notion hybride de fonction "à effets" (ou, ce qui est équivalent – sauf peut-être syntaxiquement dans certains langages – de procédure rendant des résultats) est une source de dangers, surtout si elles sont utilisées dans des expressions englobantes, comme l'ont montré quelques-uns des exemples précédents.

Arrivé à ce point de notre réflexion, il serait tentant de vouloir se passer

du modèle à états, donc de la notion d'effet – et partant de celle de procédure – de manière à ne garder que le concept de fonction pure. C'est ce que nous avons essayé de montrer tout au long de ce cours, et il semble que nous n'y avons pas trop mal réussi: cette approche, s'appuyant sur un sous-ensemble du langage CAML [2] a montré sa puissance d'expression, grâce à des concepts tels que *fonctionnalité*, *récursion*, *généricité*, etc. [3]

9.2.2 Entrées/sorties en CAML

Il y a toutefois un domaine dans lequel le modèle par substitution est insuffisant, c'est celui de l'échange de flots de données avec d'autres environnements, mis en jeu dans toutes sortes de communications, et dont la forme la plus immédiate est constituée des mécanismes d'*entrées/sorties* – appelés aussi *lecture/écriture*. Il est en effet indispensable, dans de nombreuses applications, d'aller chercher les données – et de porter les résultats – à l'extérieur du contexte d'exécution (par exemple sur des fichiers, depuis des capteurs, vers des afficheurs, etc.). Ceci implique un transfert de données depuis/vers des environnements extérieurs. Les langages offrent tous des mécanismes de communication primitifs permettant d'exprimer de tels transferts. Nous allons examiner ceux offerts par le langage CAML car, outre qu'ils complètent utilement les possibilités d'utilisation de ce langage, ils sont conceptuellement significatifs des problèmes posés par leur utilisation au niveau de la programmation.

9.2.2.1 Notion de canal

La communication entre le contexte d'un processus et les environnements extérieurs se fait à travers des dispositifs appelés *canaux*. Chaque canal permet la communication dans un sens (entrée ou sortie) depuis ou vers un environnement externe (un fichier, le clavier, l'écran, etc.). En CAML, la notion de canal est abstraite par deux types de données, le type `in_channel` pour les canaux entrants (c'est-à-dire extérieur ->contexte), et le type `out_channel` pour les canaux sortants (contexte ->extérieur). Les valeurs de type canaux sont en fait des *descripteurs de fichiers*, dont la structure exacte importe peu au programmeur [4]. Les différentes opérations applicables aux valeurs de ce type sont définies soit comme fonctions, soit comme procédures. Nous allons examiner rapidement les plus courantes, d'abord pour les canaux de sortie, puis pour les canaux d'entrée.

2. mais elle aurait aussi bien pu s'appuyer sur d'autres langages dits "fonctionnels", comme LISP ou son dialecte SCHEME par exemple.

3. et encore, nous n'avons pas présenté toutes les possibilités, notamment en matière de structuration des données!

4. et d'ailleurs, elle leur est cachée : il s'agit d'un exemple de *type de données abstrait*, concept que l'on retrouvera dans les cours plus avancés, notamment au cœur de la programmation par objets

9.2.2.2 Opérations sur le type *out_channel* : écritures sur fichiers

On définit une valeur de type *out_channel* en appliquant la fonction :

open_out : string ->out_channel

Ici, l'argument de type *string* est le nom d'un fichier dans le système d'exploitation sous-jacent (ms-dos^{TM}, ms-widows^{TM}, unix^{TM}, ...). Le résultat est une valeur de type *out _channel*; comme toute valeur, celle-ci peut être sauvegardée dans le contexte :

```
# let fr = open_out "toto.res";;
 fr : out_channel = < abstract >
```

Toute entité de type *out_channel* est donc "associée" à un fichier, sur lequel on peut "écrire" des valeurs de types imprimables, essentiellement les types *char, string, int, float*. Les "fonctions" d'écriture correspondantes sont les suivantes :

output_char : out_channel ->char ->unit

output_string : out_channel ->string ->unit

et, pour tout autre type imprimable : *output_value : out_channel ->'a ->unit*

Nous rencontrons pour la première fois le type *unit*. Il s'agit d'un type dont l'ensemble des valeurs est vide (en fait, pour des raisons syntaxiques, il comporte une seule valeur, notée *()*, signifiant "aucune valeur"). Mais alors, à quoi sert-il? En fait, la "fonction" *output_char*, par exemple, est une procédure pure: son exécution avec un canal *fr* et un caractère *c* donnés, a seulement pour effet de *déposer* la valeur liée à *c* dans le canal lié à *fr*, c'est-à-dire de modifier la valeur liée à *fr* (l'état du canal est modifié). Par contre, cette exécution ne délivre aucune valeur. Au niveau syntaxique, CAML ne connaît que la notion de fonction, de type fonctionnel *t ->t'*. Les procédures sont donc des valeurs "fonctionnelles" à type de résultat *unit*, c'est-à-dire des "fonctions" ne délivrant pas de valeurs.

```
# output_char fr 'c';;
 - : unit = ()
 (* ci-dessous, une définition sans aucun intérêt... *)
# let valeur_stupide = output_char fr '@';;
 valeur_stupide : unit = ()
# valeur_stupide;;
 - : unit = ()
```

Enfin, la fermeture d'un fichier est effectuée par la "fonction" (procédure pure)

close_out : output_channel ->unit

Attention, à l'issue de cette opération, l'identificateur est toujours dans le contexte. On ne sait pas alors quel est l'effet d'une écriture dans ce canal (probablement sans effet).

A titre d'exemple, nous donnons la définition de la fonction (procédure pure) dont l'exécution réalise la sauvegarde d'une liste de chaînes dans un fichier :

```
# let sauvegarde fr lc =
    let rec parcours l =
        if vide l
           then close_in fr
           else begin output_string fr ((hd l)^"\n"); parcours (tl l) end
     in parcours lc;;
sauvegarde : out_channel -> string list -> unit = <fun>
```

Le symbole `"\n"` désigne la valeur de chaîne ne comportant que le caractère de contrôle "passage à la ligne suivante". Les mots-clefs **begin ... end** délimitent l'expression (ici, séquence d'instructions, de type `unit`) à "évaluer" dans le cas de filtrage considéré.

9.2.2.3 Opérations sur le type `in_channel` : lectures sur fichiers

Ces opérations sont assez symétriques des opérations d'écriture. Nous avons :

ouverture :

`open_in : string ->in_channel`

fermeture :

`close_in : in_channel ->unit`

lecture d'un caractère :

`input_char : in_channel ->char`

lecture d'une chaîne (suite de caractères jusqu'à la prochaine marque de fin de ligne) :

`input_line : in_channel ->string`

Ces deux dernières fonctions rendent effectivement un résultat (ce ne sont pas des procédures pures) mais leur exécution produit aussi un effet sur leur premier argument canal d'entrée : un caractère (resp. séquence de caractères) est ôté du canal (ce ne sont pas des fonctions pures)

De manière symétrique, nous donnons la définition d'une fonction récursive permettant de constituer une liste de chaînes à partir des lignes d'un fichier (via un canal d'entrée). Pour cela, il faut déjà définir une fonction `eof : in_channel ->bool`, qui teste si la fin de fichier est atteinte, c'est-à-dire si tout le fichier a été parcouru. Ceci est établi en comparant la position courante dans le fichier (donnée par la fonction `pos_in : in_channel ->int`) avec la longueur du fichier (donnée par la fonction `in_channel_length : in_channel ->int`) . Puis, pour faire progresser la lecture dans le fichier, on définit localement la chaîne correspondant à la lecture de la prochaine ligne du fichier (ceci modifie le fichier), constituant la tête du résultat, puis on définit localement le reste du résultat en rappelant la fonction sur le fichier (on profite de l'effet effectué précédemment sur le fichier), et enfin on construit l'expression résultat. Cette manière de procéder, pas très "propre" – car utilisant implicitement l'effet de

la fonction de lecture sur le canal d'entrée – est la seule, si l'on veut définir la fonction récursivement; en effet, on ne sait pas "décomposer" la donnée, c'est-à-dire le canal d'entrée, en deux parties. Habituellement, ce genre de fonctions est programmé en utilisant des enchaînements itératifs d'actions.

```
# let eof fd = (pos_in fd = in_channel_length fd);;
 eof: in_channel -> bool = <fun>
# let rec file_to_list fd =
    if eof fd then []
           else let x=input_line fd in
                                let r = file_to_list fd in x::r;;
 file_to_list: in_channel -> string list = <fun>
```

9.2.2.4 Les canaux standard

Il s'agit des canaux d'entrée/sortie correspondant au clavier, à l'écran et au périphérique sur lequel sont envoyés les messages d'erreur (habituellement l'écran). Ces canaux sont désignés par des identificateurs prédéfinis et automatiquement ouverts:

`std_in : in_channel` correspond au clavier
`std_out : out_channel` correspond à l'écran
`std_err : out_channel` correspond au périphérique recevant les messages d'erreur (par défaut, l'écran).

Quelques fonctions, dont les significations sont évidentes:

`print_char : char -> unit`
`print_string : string -> unit`
`print_int : int -> unit`
`print_float : float -> unit`
`print_endline : string -> unit` Affiche la chaîne, suivie d'un passage à la ligne.
`print_newline : unit -> unit` Provoque un passage à la ligne.

Cette dernière fonction a un argument de type `unit`, c'est-à-dire qu'elle n'a pas d'argument. Pour l'appeler, il faut toutefois fournir un argument : c'est la valeur notée `()`, qui est la seule de type `unit`:

```
# print_newline ();;
 - : unit = ()
```

tandis que si on omet l'argument, on obtient la valeur fonctionnelle elle-même:

```
# print_newline;;
 - : unit -> unit = <fun>
```

C'est le même phénomène avec la fonction `quit : unit ->unit`, ce qui vous permet ENFIN de comprendre pourquoi, lorsqu'on veut terminer une session CAML, il faut taper la phrase `quit ()` et non pas `quit`.

A propos, ce cours est maintenant fini, vous pouvez donc taper `quit ()`.

Bibliographie

[AH-DG] T. Accart-Hardin, V. Donzeau-Gouge. *Concepts et outils de programmation.* InterEditions, 1992.

[AS] H. Abelson, G.J. Sussman, J. Sussman. *Structure et interprétation des programmes informatiques.* InterEditions, 1989.

[FH] A. Forêt, D. Herman. *Méthodes et outils de l'informatique - Approche fonctionnelle.* Cours C89, IFSIC, Université de Rennes, 1991.

[LW] X. Leroy, P. Weiss. *Manuel de référence du langage CAML.* InterEditions 1993.

[inria] http://pauillac.inria.fr/caml/index-fra.html

Annexe A

Compléments sur CAML

A.1 Expressions de filtres : définitions par cas

A.1.1 Un exemple

Nous avons vu plusieurs exemples de fonctions dont l'expression de définition est conditionnelle. Comme nous l'avions déjà signalé, une expression conditionnelle est en fait un cas particulier d'expression plus générale, appelée expression filtrée. Prenons tout de suite un exemple : soit la fonction *litt_of_num* qui, étant donné un entier (compris entre 1 et 12), délivre la chaîne exprimant le mois correspondant, en toutes lettres. Une première solution aurait l'allure suivante :

```
let litt_of_num = function n ->
    if n=1  then "janvier"
    else if n=2  then "fevrier"
    else if n=3  then "mars"
      ...
    else if n=12  then "decembre"
    else "pas de treizième mois...";;
```

Avec une construction par filtrage :

```
let litt_of_num = function
   1 -> "janvier"
  | 2 -> "fevrier"
  | 3 -> "mars"
  | 4 -> "avril"
  | 5 -> "mai"
  | 6 -> "juin"
  | 7 -> "juillet"
```

```
  | 8 -> "aout"
  | 9 -> "septembre"
  | 10-> "octobre"
  | 11-> "novembre"
  | 12-> "decembre"
  | _-> "pas de treizieme mois";;
litt_of_num:int -> string = <fun>
```

Ici, les valeurs possibles de l'argument entier de la fonction sont "analysées" en fonction d'une séquence de *filtres*. Chaque filtre est séparé du suivant par le symbole `|` (alternative). Dans l'exemple précédent, les filtres utilisées sont :

1. les *notations de valeurs entières* (constantes 1, 2, ..., 12),
2. le symbole _ (souligné)

Les premiers ne filtrent que la valeur qu'elles dénotent (1 ne filtre que la valeur 1), tandis que le second filtre n'importe quelle valeur. Ainsi, la définition de la fonction s'interprète comme suit : lors de l'évaluation de `litt_of_num n`, si `n` est compris entre 1 et 12, le filtre correspondant (la constante dénotant la valeur `n`) permet de trouver le cas correspondant. Sinon, aucun des 12 premiers filtres ne convient, et c'est le dernier qui sera sélectionné (puisqu'il filtre tout). Ce dernier cas correspond donc au **else** final, qui exprime le *"dans tous les autres cas..."*.

Souvent, les définitions par filtrage sont plus claires, elles peuvent aussi être plus concises, et nous verrons ultérieurement (chapitre sur les listes) qu'elles sont indispensables.

A.1.2 Syntaxe

Un filtre est une expression, pouvant contenir des constantes (notations de valeurs), des identificateurs ayant tous des noms différents, des constructeurs de valeurs (autres que **function**), et le symbole _ (souligné).

Exemples

1. Le filtre `(x,y)` est construit avec les deux identificateurs `x` et `y`, et le constructeur de valeurs `,`.
2. Le filtre `(true,_)` est construit avec la constante `true`, le symbole _, et le constructeur de valeurs `,`
3. Le filtre `('a',a,5)` est construit avec les constantes `'a'` et `5`, l'identificateur `a` et le constructeur de valeurs `,`
4. Le filtre `x::r` est construit avec les identificateurs `x` et `r` et le constructeur de valeur `[]`.
5. Le filtre `(1,_)::(2,_)::r` est construit avec les constantes 1, 2, l'identificateur `r`, le symbole _ et les constructeurs de valeurs `, ::`
6. L'expression `x+3` n'est pas un filtre.

A.1.3 Sémantique

Une valeur v est *filtrée* par un filtre F si elle est construite *de la même façon que* F, c'est-à-dire si toutes les constantes et tous les constructeurs de F sont présents dans v avec la même disposition. Le filtrage de v par le filtre F établit une liaison (temporaire) entre chaque identificateur apparaissant dans F et l'élément de valeur correspondant apparaissant dans v.

Exemples

1. `(5,3)` est filtrée par `(x,y)`. Ce filtrage établit les liaisons `x ~ 5` et `y ~ 3`.
2. `("bonjour",2)` est filtré par `(x,y)`. Ce filtrage établit les liaisons `x ~ "bonjour"` et `y ~ 2`.
3. `(true, 'a')` est filtré par `(true,_)`. Ce filtrage n'établit aucune liaison.
4. `(false,2)` **n'est pas** filtré par `(true,_)` car la constante `true` ne filtre que la valeur `true` (et donc ne filtre pas la valeur `false`).
5. Le filtre `('a',a,5)` filtre exactement l'ensemble des triplets de la forme `('a',v,5)`, où `v` dénote une valeur de type quelconque. Ce filtrage établit la liaison `a ~ v`. Par exemple, `('a',` **function** `x ->x+1, 5)` est filtré et la liaison `a ~` **function** `x ->x+1` est établie.
6. toute liste non vide `[`v_1 `;` v_2 `; ...;`v_n`]` est filtrée par le filtre `x::r`. Ce filtrage établit les liaisons `x ~` v_1 et `r ~ [`v_2 `; ...;` v_n`]`.
7. Le filtre `(1,_)::(2,_)::r` filtre toutes les listes ayant au moins deux éléments, commençant par les deux éléments de la forme `[(1,x); (2,y); ...]` (listes appartenant aux types `int*t list`, où `t` est un type quelconque). Le filtrage d'une liste `[(1,`x_1`); (2,`x_2`);` v_3 `; ...;` v_n`]` établit la liaison `r ~ [`v_3`;...;`v_n`]`

A.1.4 Utilisation de filtres dans les définitions de fonctions

Exemple 1 Donnons la définition de la fonction `sialsin` qui, appliquée à trois valeurs de type booléen `a, b, c` délivre la valeur de la proposition **si** `a` **alors** `b` **sinon** `c`.

```
let sialsin = function
  (true, true, _) -> true
| (false, _, true) -> true
| (_,_,_) -> false;;
sialsin:bool -> bool -> bool -> bool = <fun>
```

Exemple 2 Donnons une définition de la fonction qui, appliquée à une liste, délivre la liste obtenue en permutant les deux premiers éléments (liste inchangée si elle a moins de deux éléments):

```
# let permute = function
    x::y::r -> y::x::r
  |     s   ->  s     ;;
permute: 'a list -> 'a list = <fun>
```

Remarque 1: Le deuxième cas de filtrage ne capte que les listes ayant 0 ou 1 élément, puisque toutes celles qui ont au moins deux éléments sont filtrées par le premier cas. Si les deux cas de filtrage étaient intervertis, la définition serait fausse car le premier filtre (`s`) capterait *toutes* les listes, et donc la fonction serait l'identité...! D'où l'importance de l'ordre dans lequel on présente les filtres.

Remarque 2 L'indication de type `'a list` rendue par la machine sera expliquée au chapitre suivant. Retenons que le symbole `'a` représente un type quelconque (paramètre de type).

A.1.5 La construction match ... with

Avec la forme syntaxique abrégée (§4.4), les définitions de fonctions par filtrage restent possibles, grâce 'a la construction **match ... with** ayant la signification suivante :

```
match expression with
  filtre1 -> expr1
| filtre2 -> expr2
| ...
| filtrek -> exprk
```

est une expression, évaluée de la manière suivante : `expression` est évaluée. Soit v sa valeur. Soit `filtrei` le premier filtre de la séquence filtrant v. La valeur de `expression` est celle de `expri`, évaluée dans le contexte enrichi par les liaisons établies lors du filtrage. Ces liaisons sont détruites à la fin du processus d'évaluation de l'expression **match ... with**.

Pour illustrer cette construction, donnons la définition par filtrage de la fonction `num_of_litt` qui, à chaque mois donné sous forme de chaîne fait correspondre son numéro entre 1 et 12. Le filtrage se fait sur le couple du premier et du dernier caractère de la chaîne.

```
let num_of_litt ch =
  let p = nth_char ch 0 and
      d = nth_char ch (string_length ch -1) in
  match (p, d) with
  ('j','r') -> 1
| ('f',_) -> 2
| ('m','s') -> 3
```

```
|  ('a','l') -> 4
|  ('m','i') -> 5
|  ('j','n') -> 6
|  ('j','t') -> 7
|  ('a','t') -> 8
|  ('s','e') -> 9
|  ('o',_) -> 10
|  ('n',_) -> 11
|   _      -> 12;;
num_of_litt:string -> int = <fun>
```

Nous avons finalement le schéma syntaxique suivant (figure A.1) complétant la partie *expression* du schéma de la figure 3.6.

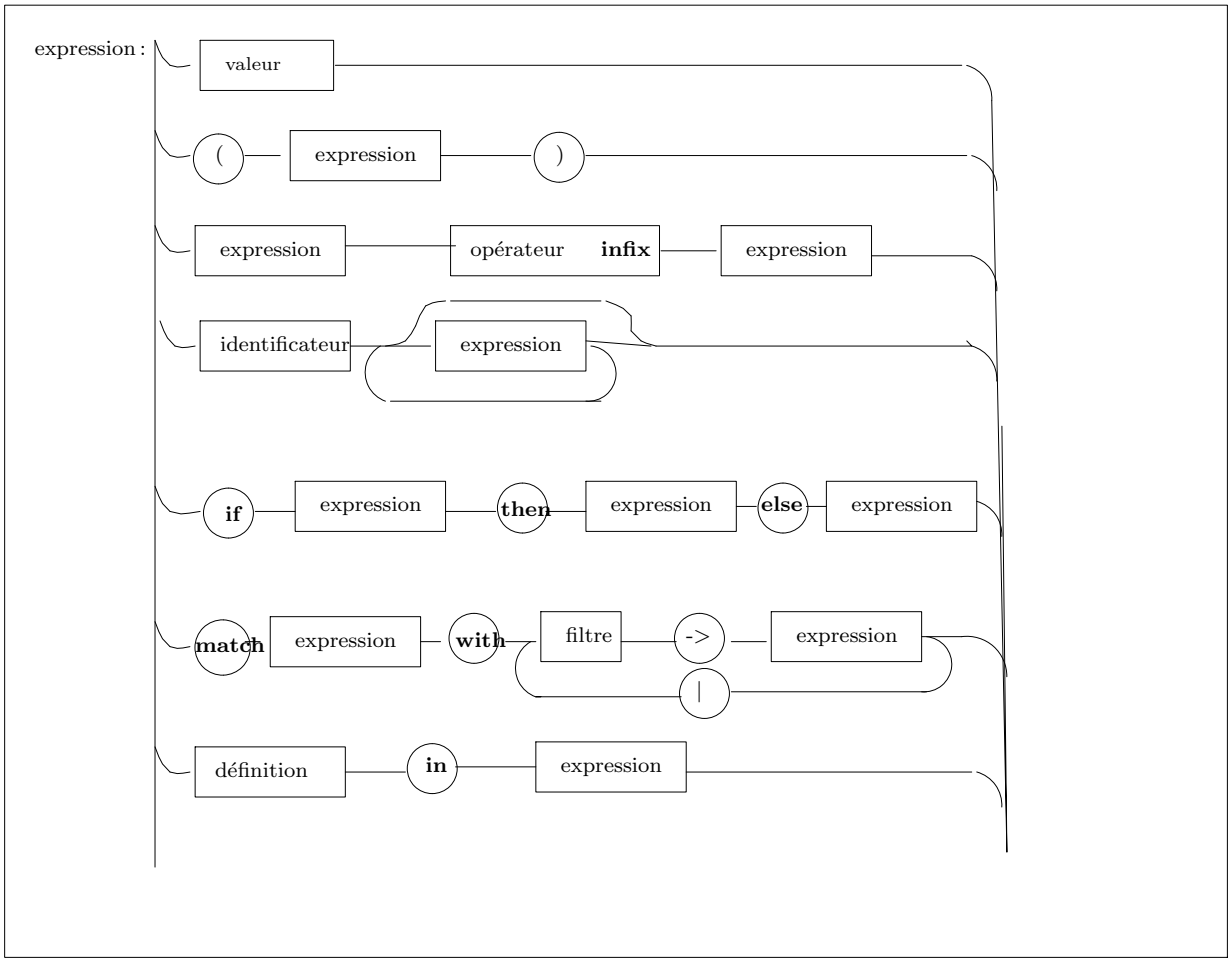


FIG. A.1 – *Expression*

A.2 Extraits de la bibliothèque de base

Extraits du document *en anglais* accessible en ligne sur Internet (URL `http://pauillac.inria.fr/caml/man-caml/index.html`

A.2.1 bool: boolean operations

```
prefix & : bool -> bool -> bool
prefix && : bool -> bool -> bool
prefix or : bool -> bool -> bool
prefix || : bool -> bool -> bool
```

The boolean **and** is written `e1 & e2` or `e1 && e2`. The boolean **or** is written `e1 or e2` or `e1 ||e2`. Both constructs are sequential, left-to-right: `e2` is evaluated only if needed. Actually, `e1 & e2` is equivalent to **if** `e1` **then** `e2` **else** `false`, and `e1 or e2` is equivalent to **if** `e1` **then** `true` **else** `e2`.

```
prefix not : bool -> bool
```

The boolean negation.

```
string_of_bool : bool -> string
```

Return a string representing the given boolean.

A.2.2 string: string operations

```
string_length : string -> int
```

Return the length (number of characters) of the given string.

```
nth_char : string -> int -> char
```

`nth_char s n` returns character number `n` in string `s`. The first character is character number 0. The last character is character number `string_length s - 1`. Raise `Invalid_argument "nth_char"` if `n` is ouside the range 0 to `(string_length s - 1)`. You can also write `s.[n]` instead of `nth_char s n`.

```
set_nth_char : string -> int -> char -> unit
```

`set_nth_char s n c` modifies string `s` in place, replacing the character number `n` by `c`. Raise `Invalid_argument "set_nth_char"` if `n` is ouside the range 0 to `(string_length s - 1)`. You can also write `s.[n] <- c` instead of `set_nth_char s n c`.

```
prefix ^ : string -> string -> string
```

`s1 ^s2` returns a fresh string containing the concatenation of the strings `s1` and`s2`.

```
sub_string : string -> int -> int -> string
```

`sub_string s start len` returns a fresh string of length `len`, containing the characters number `start` to `start + len - 1` of string `s`. Raise `Invalid_argument "sub_string"` if `start` and `len` do not designate a valid substring of `s`; that is, if `start < 0`, or `len < 0`, or `start + len > string_length s`.

```
create_string : int -> string
```

`create_string n` returns a fresh string of length `n`. The string initially contains arbitrary characters.

```
make_string : int -> char -> string
```

`make_string n c` returns a fresh string of length `n`, filled with the character `c`.

```
fill_string : string -> int -> int -> char -> unit
```

`fill_string s start len c` modifies string `s` in place, replacing the characters number `start` to `start + len - 1` by `c`. Raise `Invalid_argument "fill_string"` if `start` and `len` do not designate a valid substring of `s`.

```
blit_string : string -> int -> string -> int -> int -> unit
```

`blit_string s1 o1 s2 o2 len` copies `len` characters from string `s1`, starting at character number `o1`, to string `s2`, starting at character number `o2`. It works correctly even if `s1` and `s2` are the same string, and the source and destination chunks overlap. Raise `Invalid_argument "blit_string"` if `o1` and `len` do not designate a valid substring of `s1`, or if `o2` and `len` do not designate a valid substring of `s2`.

```
replace_string : string -> string -> int -> unit
```

`replace_string dest src start` copies all characters from the string `src` into the string `dst`, starting at character number `start` in `dst`. Raise `Invalid_argument "replace_string"` if copying would overflow string `dest`.

```
eq_string : string -> string -> bool
neq_string : string -> string -> bool
le_string : string -> string -> bool
lt_string : string -> string -> bool
ge_string : string -> string -> bool
gt_string : string -> string -> bool
```

Comparison functions (lexicographic ordering) between strings.

`compare_strings : string -> string -> int`

General comparison between strings. `compare_strings s1 s2` returns 0 if `s1` and `s2` are equal, or else -2 if `s1` is a prefix of `s2`, or 2 if `s2` is a prefix of `s1`, or else -1 if `s1` is lexicographically before `s2`, or 1 if `s2` is lexicographically before `s1`.

`string_for_read : string -> string`

Return a copy of the argument, with special characters represented by escape sequences, following the lexical conventions of Caml Light.

A.2.3 conversions with int, float, char, string

`int_of_char : char -> int`

Return the ASCII code of the argument.

`char_of_int : int -> char`

Return the character with the given ASCII code. Raise `Invalid_argument "char_of_int"` if the argument is outside the range 0–255.

`char_for_read : char -> string`

Return a string representing the given character, with special characters escaped following the lexical conventions of Caml Light.

`string_of_int : int -> string`

Convert the given integer to its decimal representation.

`int_of_string : string -> int`

Convert the given string to an integer, in decimal (by default) or in hexadecimal, octal or binary if the string begins with 0x, 0o or 0b. Raise `Failure "int_of_string"` if the given string is not a valid representation of an integer.

`int_of_float : float -> int`

Truncate the given float to an integer . The result is unspecified if it falls outside the range of representable integers.

`float_of_int : int -> float`

Convert an integer to floating-point.

`string_of_float : float -> string`

Convert the given float to its decimal representation.

`float_of_string : string -> float`

Convert the given string to a float, in decimal. The result is unspecified if the given string is not a valid representation of a float.

A.2.4 Some operations on numbers

```
abs : int -> int
```

Return the absolute value of the argument.

```
abs_float : float -> float
```

Return the absolute value of the argument.

```
prefix ** : float -> float -> float
prefix **. : float -> float -> float
power : float -> float -> float
```

Exponentiation.

```
exp : float -> float
log : float -> float
sqrt : float -> float
sin : float -> float
cos : float -> float
tan : float -> float
asin : float -> float
acos : float -> float
atan : float -> float
atan2 : float -> float -> float
```

Usual transcendental functions on floating-point numbers.

A.3 Types construits

Les types construits sont définis par l'utilisateur. Ils peuvent être identifiés grâce à une phrase de *définition de type* selon la syntaxe suivante :

```
# type ident_type = expression_definition_type ;;
 type ident_type defined.
```

A.3.1 Types produit

Comme les t-uples, les types produits sont des produits cartésiens, mais ils permettent en outre de nommer les composants, qu'on appelle des *champs*.

Exemple :

```
# type individu = {Nom : string ; Prenom : string ; Secu : int} ;;
 type individu defined.
```

Le type `individu` comporte trois champs. L'acces aux différents champs répond à la syntaxe suivante : si `moi` est une variable de type `individu`, les expressions `moi.Nom`, `moi.Prenom` et `moi.Secu` ont les valeurs respectives des trois champs de la variable. Une variable de ce type peut être définie :

```
# let lui={Nom = "Lapointe" ; Prenom = ''Bobby'' ; Secu = 123042} ;;
 lui : individu = {Nom = "Lapointe" ; Prenom = ''Bobby" ; Secu = 123042}
# lui.Nom ;;
 - : string = ''Lapointe"
```

A.3.2 Types énumérés

Les valeurs possibles d'un type énuméré sont dénotées par des identificateurs définis par le programmeur, appelés *constructeurs de valeurs*.

Exemple:

```
# type couleur = Blanc | Jaune | Orange | Rouge | Vert | Bleu | Violet ;;
 type couleur defined.
# Blanc ;;
 - : couleur = Blanc
# let l_Orient_est = Rouge ;;
 l_Orient_est : couleur = Rouge
```

Les types énumérés sont des cas particuliers de types *somme* (ou *union*), ici l'union de valeurs constantes (énumération de valeurs).

A.3.3 Types somme

Mathématiquement parlant, il s'agit de l'*union* d'ensembles disjoints (somme disjointe). L'ensemble des valeurs d'un type somme est l'union des ensembles de valeurs des types composants.

Exemple :

```
# type nombre = Ent of int | Re of float ;;
 type nombre defined.
```

Une valeur du type nombre est *soit* du type `int` *soit* du type `float`. Les identificateurs `Ent` et `Re` sont des *constructeurs de valeurs* obligatoires pour dénoter des valeurs de type `nombre` :

```
# let un_entier = Ent 123 ;;
 un_entier : nombre = Ent 123
# let un_reel = Re 123. ;;
 un_reel : nombre = Re 123.
# Re 99 ;;
 > Toplevel input:
>Re 99;;
>   ^^
> Expression of type int
> cannot be used with type float
```

Attention, le type `nombre` est *différent* du type `int` comme du type `float` :

```
# Ent 1 + Ent 2 ;;
 > Toplevel input:
>E 1 + E 2 ;;
> ^^^
> Expression of type nombre
> cannot be used with type int
```

A.3.3.1 Types paramétrés

Les types produit et somme sont construits à partir de types composants. Certains de ces composants (voire tous) peuvent être des *paramètres de type*:

Exemple :

```
# type 'a prod = {Prem : 'a ; Sec : int} ;;
 type 'a prod defined.
# let x={Prem=Blanc ; Sec=12} ;;
 x : couleur prod = {Prem=Blanc ; Sec=12}

# type 'a som = Prem of int | Sec of 'a ;;
 type 'a som defined
# let y = Sec (Re 3.1416) ;;
 y : nombre som = Re 3.1416
```

A.3.4 Types récursifs

Les types paramétrés rendent possible la définition de types *récursifs*. Par exemple, une définition possible du type *séquence* est la suivante:

```
# type 'a sequence = Vide | Cons of 'a * 'a sequence ;;
 type sequence defined.
```

Les valeurs possibles du type `sequence` sont donc : *soit* la constante dénotée `Vide`, *soit* un doublet, composé d'une valeur de type `'a` et d'une séquence d'éléments de type `'a`.

A.3.5 Utilisation des types construits : filtrage

Les constructeurs de valeurs utilisés dans les notations de valeur de type produit, somme, etc. peuvent servir à constituer de expressions de filtre.

Exemple 1

```
# type sexe = Homme | Femme ;;
 type sexe defined.
# type exploitant = {Nom : string ; S : sexe ; Age : int} ;;
 type exploitant defined.
# type culture = Haricot | Mais | Ble | Betterave ;;
 type culture defined.
# type statut = Jachere | Cultive of culture ;;
 type statut defined.
# type un_lot = {Cadastre : int ; Proprio : quidam ; Etat : statut} ;;
 type un_lot defined.

 (* un lot est-il cultivé avec la culture c ? *)
# let cultive_avec c = function
       {Cadastre =_ ; Proprio =_ ; Etat = Cultive x} -> x=c
     | _ -> false;;
 cultive_avec : culture -> un_lot -> bool = <fun>

 (* un lot est-il exploité par une femme? *)
# let expl_femme = function
       {Cadastre = _ ; Proprio.S = Femme ; statut=_} -> true
     | _ -> false ;;
 expl_femme : un_lot -> bool=<fun>

 (* Calculer le nombre de lots cultivés en Maïs par des hommes
```

dans une exploitation (liste de lots) *)

```
# let rec nb_mais_homme = function
    []  -> 0
|  {Cadastre=_ ; Proprio={Nom=_ ; S=H ; age=_} ; Etat= Cultive Mais}::r -> 1 + nb_mais_homme r ;;
 nb_mais_homme : un_lot list -> int = <fun>
```

Exemple 2 : définir l'addition de deux nombres (avec le type nombre vu plus haut).

```
# let add x y = match (x, y) with
     (Ent i,Ent j) -> Ent (i+j)
   |  (Ent i, Re r)  -> Re (float_of_int i +. r)
   |  (Re r, Ent i)  -> Re (float_of_int i +. r)
   |  (Re r, Re s) ->  Re (r+.s) ;;
 add : nombre -> nombre -> nombre = <fun>

# add (Re 1.5) (Ent 2) ;;
 - : nombre = Re 3.5

# add (Ent 3) (Ent 2) ;;
 - : nombre = Ent 5

# add 1.5 2.5 ;;
 > Toplevel input:
>add 1.5 2.5 ;;
>     ^^^
> Expression of type float
> cannot be used with type nombre
```

Index