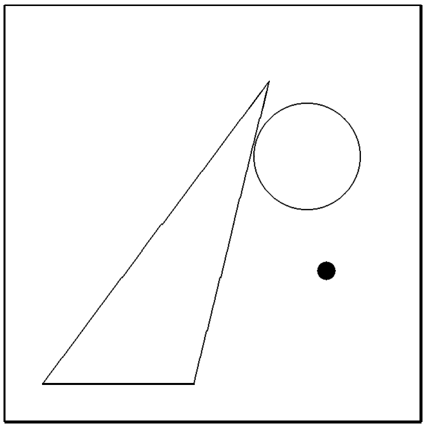
Le filtrage
une application de (et pour) Lisp

Christian Queinnec
Internet : `Christian.Queinnec@polytechnique.fr`



9 Mai 1995

Table des matières

1 Préface

L'univers n'est pas un chaos; nous y observons la récurrence de formes typiques auxquelles nous donnons des noms.

RENÉ THOM, *Stabilité structurelle et morphogenèse*

Cette monographie est entièrement dédiée au *filtrage* et plus particulièrement au filtrage en Lisp. Cette importante technique constitue l'un des tours de main les plus courants de Lisp et de ses multiples dialectes mais c'est aussi une technique fondamentale de l'intelligence artificielle. Les rudiments du filtrage sont exposés dans de nombreux manuels comme [Winston & Horn 81, pp 219–231], [Queinnec 82, pp 126–129], [Kiremitdjian & Roy 85, pp 251–263], [Wertz 85, pp 124–126,172–190], [Abelson & Sussman 85, pp 367–372], [Kessler 88, pp 440–447] ou [Winston & Horn 88, pp 353–366] ... Le filtrage y apparaît souvent sous la forme plaisante de Doctor ou Eliza, deux programmes simulant un psychiatre aux prises avec des patients, tout pareillement programmeurs et paranoïaques.

Le filtrage, appliqué à la spécification d'interprètes comme Lisp ou Conniver, a été mis à l'honneur, en France, par Patrick Greussay dans sa thèse [Greussay 77]. Il fonde aussi la manière d'invoquer les fonctions de Plasma dont l'étude est toujours vivante à Toulouse [Arcangeli & Pomian 90]. Outre-Atlantique, un compilateur de filtres écrit par Larry Masinter [Masinter 78] figurait aussi et déjà dans InterLisp dès 1972. On peut enfin souligner l'importance grandissante de la définition par filtrage dans les langages fonctionnels tels que CAML [Weis 89], Miranda™ [Turner 85] ou plus récemment encore, Haskell [Hudak & Wadler 90].

Deux choses toutefois manquent à ces descriptions — elles sont sommaires et omettent souvent les segments nommés, — elles sont fermées et ne peuvent être simplement étendues à de nouveaux filtres. Nous avons donc pris le parti de combler ces vides et d'expliciter la genèse du filtrage tout en mettant en évidence la nécessité interne de certaines "catastrophes" de structure programmatique qui surviennent au fil des contraintes. Nous avons aussi décidé de pousser jusqu'à la compilation des filtres, c'est-à-dire leur traduction en des programmes Lisp spécialisés (et, bien sûr, compilables). Nous estimons en effet que cette approche du phénomène de compilation est particulièrement pédagogique puisque langage de départ et d'arrivée sont simplement appréhendables sans que la transformation de l'un vers l'autre ne soit triviale. Enfin pour nous faire plaisir et mettre sous la dent du lecteur intéressé autre chose qu'une bouillie prémâchée (et probablement de ce fait insipide) les techniques de compilation de filtres qui seront exposées au dernier chapitre sont originales et neuves [Queinnec 90].

Cette monographie a été conçue comme une "promenade" ou "aventure". En raison du manque de moyens techniques qui auraient permis de flâner autour des programmes tout en les interrogeant, peut-être à la façon d'un hypertexte, sur leur théorie sous-jacente, leurs propriétés, les raisons de leur genèse ou les subtilités locales auxquelles ils se sont conformés, nous avons dû adopter la structure linéaire classique. Le fil conducteur de l'ouvrage est la présentation d'un *moteur à filtrer* qui sera successivement modifié à mesure qu'on lui demandera de traiter des filtres de plus en plus complexes. Tout en variantes autour d'un thème fuyant, l'ouvrage s'appuie sur de nombreux travaux en continuelle expansion comme en témoigne la bibliographie en fin d'ouvrage.

Le but primordial est de démythifier un certain nombre d'outils usuels de la programmation en Lisp, d'une part car ils ne sont pas inaccessibles au commun des "lispeurs" et d'autre part parce qu'ils constituent d'excellents exemples de programmes, un tantinet complexes, mettant en valeur quelques-uns des plus fermes fondements de l'art de la programmation en Lisp. Sont donc illustrés, tout au long de cette étude sur le filtrage, et en vrac :

- les techniques d'interprétation,
- les problèmes de compilation,
- les rapports entre interprètes et compilateurs,
- la nature des bibliothèques d'exécution,
- la programmation déclarative,
- la réutilisation de code,
- la programmation par retour-arrière,
- les continuations,
- les langages intermédiaires,
- la génération de code,
- la frontière entre ex- et im- plicite
- et bien d'autres choses encore,...

Les premiers points énoncés soulignent bien le parallèle qu'il y a entre filtreur et évaluateur, filtrage et évaluation, filtre et programme ... Les problèmes sont parents sauf que ceux des évaluateurs sont d'une complexité bien plus élevée.

Les programmes ici présentés sont écrits en (ce qui sera fort probablement) le Lisp normalisé par l'Organisation Internationale de Standardisation ou ISO. Ils peuvent être portés sans problème vers Le-Lisp[1], COMMON LISPou Scheme. Le langage choisi n'a cependant nullement été restreint et l'on trouvera des emplois non triviaux des traits des Lisps modernes comme valeur multiple, objet fonctionnel de première classe, structure, objet, fonction générique (CLOS [Bobrow *et al.* 88]) et, bien sûr, liaison lexicale. *Common Lisp, the Language* de Guy L. Steele, Jr. [Steele 90], est, à l'heure de parution de cette monographie, une bonne approximation du dialecte ici employé.

Cet ouvrage doit beaucoup à un grand nombre de personnes qui par leurs conseils, écrits, remarques, amitiés, relectures ou encouragements ont contribué à son éclosion. Que soient ici remerciés Josy Baron, Jean-Pierre Briot, Nathalie Bucsek, Jérôme Chailloux, Nicole Cheutin, Pierre Cointe, Charles Consel, Olivier Danvy, Vincent Delacour, Alain Deutsch, Marc Gengler, Nicolas Graube, Christian Jullien, Cyprienne Lancray-Javal, Eugen Neidl, Claire N***, Greg Nuyens, Jean-François Perrot, Emmanuel Saint-James, Bernard Serpette, Nitsan Séniak et Martine Wiznitzer.

La composition a grandement profité des remarquables TeX et LaTeX. Loués soient leur progéniteur Donald E. Knuth et Leslie Lamport à qui l'auteur est redevable de quelques peines et de nombre de joies.

Résumé

Que cette programmation souhaitée "exemplaire" (mais certainement encore améliorable sur le triple plan de l'esthétique, de la fonctionnalité et de l'efficacité) puisse inciter le lecteur à acquérir de bonnes habitudes programmatrices ainsi que de solides vertus structurantes.

[1] Le-Lisp est une marque déposée de l'Institut National de Recherche en Informatique et Automatique (INRIA).

Prolégomènes 2

Je ne peux m'en délivrer qu'en allant jusqu'au bout de cette histoire. Mais que trouverai-je en fin de parcours ? Vous n'êtes pas de celles qui ferment une histoire. Vous seriez plutôt de celles qui la laissent ouverte en vue d'en faire un conte infini. Votre histoire est une suite de portes qui s'ouvrent sur des territoires blancs et des labyrinthes qui tournent ...

Tahar Ben Jelloun, *La nuit sacrée*

La plupart des langages informatiques connaissent la notion de "fonction", "procédure" ou "sous-programme" qui permet de condenser en une entité, souvent pourvue d'un nom, une séquence d'instructions susceptible d'être plusieurs fois employée. Outre les réductions de taille de code qu'elle permet d'obtenir et les effets bénéfiques pour la maintenance qu'autorise le confinement en un unique endroit de certains algorithmes, deux qualités majeures la caractérisent :

- elle est appelable, c'est-à-dire que l'on peut explicitement l'autoriser à dérouler ses calculs,
- elle peut acquérir des informations spécifiques lors de son appel.

Une fonction peut aussi retourner de l'information ou opérer, par effet de bord, des modifications sur son environnement d'appel mais ces propriétés ne sont pas essentielles ici.

De multiples mécanismes ont été imaginés pour ce transfert d'informations entre appelant et appelé. En terme d'évaluation, existent les appels par nom, par valeur, par nécessité ... En terme d'implantation physique, se trouvent les passages par référence, par valeur, par variable ... Enfin en terme d'appariement variables $\leftrightarrow$ arguments, se rencontrent la liaison terme à terme, la liaison déstructurante, le filtrage à la PLASMA [Hewitt 78], le filtrage typé à la ML [Weis 89] ou l'unification à la PROLOG [Giannesini *et al.* 85]. Toutes ces variantes reviennent à nommer, grâce à un ensemble structuré de noms, un ensemble non moins structuré de valeurs.

2.1 La liaison terme à terme

En Lisp ou en Scheme, les fonctions acquièrent des *arguments* (valeurs de leurs *paramètres d'appel*) via leur *liste de variables*. La liste des variables n'est rien d'autre que l'énumération séquentielle des variables de la fonction et utilise la seule structure agrégative de données permise pour la représentation des programmes : la liste. À la fonction Δ mathématiquement ainsi définie :

$\Delta\ : a,b,c \longrightarrow b^2 - 4ac$

correspond la fonction `delta` ainsi transcrite en Lisp :

$\boxed{\texttt{delta}}$

```
(defun delta (a b c)
  (- (* b b) (* 4 a c)) )
```

a, b et c sont les *variables*. Les valeurs de a, b et c sont les *arguments*. Au sein de la forme (delta 5 (1- 5) (1+ 2)), les expressions 5, (1- 5) et (1+ 2) sont les *paramètres d'appel*. Quoique le poids des laxismes propres au jargon de Lisp pousse au mélange de ces trois notions distinctes, le présent ouvrage respecte la présente terminologie.

Les deux écritures, mathématique ou lispienne, contiennent autant d'informations l'une que l'autre mais chacune possède ses propres fioritures syntaxiques ou tics langagiers.

Outre de pouvoir être définies, les fonctions doivent aussi pouvoir être invoquées. Un appel à la fonction Δ revêt la forme :

$\Delta(5, 4, 3)$

Du côté de Lisp, l'appel équivalent à `delta` s'écrit quant à lui

```
(delta 5 4 3)
```

Là encore, les deux écritures sont similaires et contiennent autant d'informations. La structure adoptée par Lisp est dite polonaise préfixée de Cambridge. La notation "polonaise" fut inventée par le non moins polonais logicien Jan Łukasiewicz (1878-1956) afin de se débarrasser des parenthèses qui encombrent, suivant une logique floue, les expressions mathématiques et ainsi atteindre une syntaxe pure. La variante préfixée consiste à citer les fonctions en tête de leurs paramètres. La sous-variante dite de Cambridge (Massachusetts : le siège du MIT où fut conçu Lisp) consiste à enclore de parenthèses les appels fonctionnels et ainsi permet de couler une écriture préfixée en une S-expression convenable.

L'appariement des variables aux arguments est dit *terme à terme, positionnel* puisque les valeurs des paramètres d'appels sont calculées puis liées aux variables de même rang. L'appel doit bien sûr comporter autant de paramètres que ne compte de variables la fonction appelée.

2.2 La liaison n-aire

Le besoin s'est vite fait sentir de *fonctions* n-*aires*, c'est-à-dire de fonctions prenant un nombre variable d'arguments. Les fonctions fondamentales `list`, `append`, `plus` ... sont de cette nature. Cette extension qui n'a pas de strict équivalent mathématique, représente en revanche une importante facilité de programmation que souvent les langages classiques ignorent. Lisp répond à ce besoin grâce à trois conventions permettant — de définir des fonctions n-aires, — d'invoquer des fonctions n-aires — et de représenter des valeurs n-aires. Ces conventions s'appuient fortement sur la structure parenthétique de Lisp.

On exprime qu'une fonction prend une suite d'arguments en restreignant la liste de variables de cette fonction à une unique variable comme dans

```
(defun revlist args                                    revlist
  (reverse args) )

(revlist 1 '2 (1- 4)) ⇒ (3 2 1)
```

Ce mécanisme syntaxique se généralise aisément aux fonctions prenant un nombre fixe d'arguments suivis d'autres, optionnels ou en nombre quelconque :

```
(defun pref (prefix . args)                            pref
  (mapcar #'(lambda (term) (list prefix term))
          args ) )

(pref 'bar 'bar 'hack) ⇒ ((BAR BAR) (BAR HACK))
```

Cette généralisation syntaxique repose sur le fait que la surprenante expression (. *symbole*) est identifiable et (souvent) identifiée (par `read`) au seul *symbole*.

Grâce à la syntaxe parenthésée, un appel à une fonction n-aire est simple à imaginer : il suffit d'enserrer tous les paramètres d'appel entre les mêmes parenthèses qui enclosent ceux d'une fonction d'arité fixe. Il faut noter au passage que la notation polonaise ne pouvait se passer de parenthèses que parce que tous les opérateurs étaient d'arité fixe : vouloir manipuler des opérateurs d'arité variable impose de pouvoir syntaxiquement circonscrire l'ensemble de leurs paramètres d'appel et donc de réintroduire un mécanisme de regroupement syntaxique. Le mécanisme de liste, omniprésent en Lisp, étant déjà adopté pour l'appel fonctionnel d'arité fixe, se voit donc naturellement étendu à l'arité variable.

La troisième convention impliquée dans le mécanisme des fonctions n-aires est que la suite d'arguments est regroupée en une liste. D'autres choix sont possibles tels que vecteur ou séquence [Steele 90, page 388]. Le problème essentiel induit par cette représentation est qu'elle confère à l'objet "*argument multiple*" plus de propriétés que souhaitable. Un argument multiple ne devrait être qu'une structure de données regroupant un certain nombre de termes, chaque terme représentant un argument, et n'autorisant que le seul accès à ces arguments. Une liste, au contraire, est une structure muable, modifiable par `rplacd` ce qui a pour effet d'altérer le nombre d'arguments reçus : le sens mathématique attribuable à cette opération laisse rêveur !

La liaison n-aire permet de condenser dans une unique variable une suite d'arguments manipulable, par là même, comme un tout. Le mécanisme de valeurs multiples [Steele 90, page 179] est complémentaire de celui d'arguments multiples et sera parfois utilisé dans cet ouvrage.

2.3 La liaison déstructurante

Le mécanisme précédent a pu encore être étendu en vision paire pointée grâce à la liaison déstructurante [Queinnec 84, Chapitre 7.7 page 179] où, à l'acquisition d'un nombre déterminé d'arguments, se superpose un déconditionnement d'iceux. Dans l'exemple :

```
(defun fooA (bar (baz . hux) . wrek)                                    fooA
  ... )
```

`fooA` est une fonction requérant au moins deux arguments. Le premier est quelconque, le second doit être une liste non vide décomposée en ses `car` et `cdr`, les suivants, s'il y en a, peuvent être quelconques. Cette écriture semble n'être qu'une contraction extrêmement concise et tout à fait innocente de :

```
(defun fooB bar                                                         fooB
   (let ((bar (if (consp bar) (car bar)
                  (error ...)))
         (baz (if (and (consp (cdr bar))
                       (consp (cadr bar)) )
                  (caadr bar)
                  (error ...) ))
         (hux (cdadr bar))
         (wrek (cddr bar)) )
        ...) )
```

Le danger provient de deux côtés. En premier lieu, la fonction `fooA`, si elle use de liaison déstructurante, ne peut contrôler d'éventuelles erreurs d'appel puisque c'est l'évaluateur qui remarquera et traitera ces dysfonctionnements. À l'appel `(fooA` α β γ `...)`, seront évalués tour à tour les paramètres α, β, γ etc. L'évaluateur tentera alors de lier leur valeur aux variables de `fooA` enfin, en cas de succès, le corps de `fooA` sera évalué dans l'environnement ad hoc. Cette succession d'étapes montre bien l'existence d'une phase de liaison qui est du ressort de l'évaluateur et qui n'appartient plus au traitement de la forme d'appel sans toutefois encore appartenir à la fonction appelée. S'il y a erreur dans le processus de liaison, qu'incriminer ? L'appel à `fooA` effectué avec des arguments incorrects ? Ou, si l'on suppose que les arguments sont corrects, que ce n'était point `fooA` qu'il fallait invoquer ? Ce ne peut donc être que l'appelant qui peut traiter cette exception. Dans la seconde programmation plus explicite, c'est la fonction `fooB`, elle-même, qui détectera ces erreurs et invoquera `error` (ou tout autre traitement qu'elle jugera approprié).

En second lieu, la déstructuration induit des différences dans le traitement de la liste des variables. En effet, dans `(bar (baz . hux) . wrek)` le nombre de termes indique le nombre d'arguments légal pour la fonction (ici au moins deux), ce qui peut être aisément vérifié, dans toute invocation de `fooA`, à la main ou par compilation. Les longueurs des sous-termes sont, quant à elles, quelconques et ne rendent compte que de propriétés dynamiques : la valeur du deuxième argument de `fooA` doit être une liste non vide, ce qui, du fait du typage dynamique de Lisp, ne peut en général être prouvé.

Ce raisonnement ne tient pas pour les macros qui peuvent utiliser cette liaison particulière compte tenu de leur non-appartenance au monde de l'évaluation : les macro-formes sont généralement traitées lors de la compilation. Tous leurs paramètres d'appel sont connus statiquement et toute erreur est traitée par le macro-expanseur.

Un dernier écueil lié à la déstructuration est que si l'on utilise celle-ci pour décomposer une donnée, tout changement de représentation de cette donnée impliquera une modification de cette fonction qui porte gravée en elle la structure de la dite donnée. De plus utiliser des listes pour représenter des enregistrements est coûteux en terme d'espace et d'accès et ne se peut concevoir que comme substitut temporaire d'une programmation correcte.

La liste des variables (ou, en liaison déstructurante, l'arbre binaire des variables) recèle une double information. C'est tout d'abord un ensemble de noms ainsi qu'une description de la structure des arguments attendus. Cette description sert elle-même un double dessein : vérifier la forme que doit revêtir un appel à la fonction (nombre et nature des paramètres) puis extraire les arguments grâce à la combinaison de `car` ou `cdr` ad-hoc. L'intérêt de cette description est une fois de plus double : sa concision et sa simplicité permettent de ne plus penser en termes de sélecteurs mais de l'appréhender en fonction de sa seule apparence.

Les mécanismes précédents concernant les variables n-aires et la déstructuration ont été encore augmentés en MacLisp puis COMMON LISP [Steele 90]. On y trouve des *lambda-lists* (sic) ou *listes de variables* truffables de directives diverses pour introduire des variables optionnelles (`&optional`), auxiliaires (`&aux`), des mots-clés (`&key`) etc. La liste des variables d'une fonction devient une structure complexe correspondant à un traitement non négligeable tant à la définition de la fonction que lors de ses applications[1].

Les fonctions *génériques* furent introduites en premier par Common Loops [Bobrow *et al.* 86] puis affinées par COMMON LISPObject System [Bobrow *et al.* 88]. La liste des variables possède en plus de ce qui précède des indications de type d'arguments. Le corps de la fonction générique n'est plus unique mais est formé d'une collection de *méthodes* dont le choix sera déterminé par le type des arguments. Cet apport de la programmation par objets conduit à encore enrichir la liste des variables d'indication de types, d'ordre dans la discrimination ou même d'exceptions liées à des instances particulières (`eql`).

2.4 Exemples de filtrage

Le filtrage (*Pattern Matching* ou *Template Filling*) est une technique déjà ancienne mais incomparablement puissante que cet ouvrage exposera progressivement. Techniquement le filtrage est une fonction prédicative comparant une expression quelconque et un filtre qui code à la fois la structure attendue, les vérifications à effectuer et la description des parties à extraire : le filtrat ! Nous présenterons tout d'abord un moteur à filtrer qui interprète son filtre. C'est un premier pas vers une programmation déclarative (le filtre est une description a priori non attachée à un algorithme quelconque) et non déterministe (puisque l'on verra des filtrages conduisant à plusieurs solutions).

Donnons de suite quelques exemples de filtrage qui devraient favoriser l'intuition de ce que sont filtres et filtrage.

`(nom ?-)`
: filtre les listes de deux termes dont le premier doit être le symbole `nom`. "`?-`" est, quant à lui, l'attrape-tout et signifie "n'importe quoi".

`(?qui est une femme)`
: filtre les listes de quatre termes (dont les trois derniers doivent être les symboles `est`, `une` et `femme`) pour en extraire le premier sous le nom `qui`. "`?qui`" est un filtre unaire, il ne filtre qu'une unique expression : "`qui`" est une variable de filtrage.

`(?thm implique ?thm)`
: filtre les listes de trois termes dont le second doit être le symbole `implique` et dont les troisième et premier sont égaux. On assiste là au double emploi d'une variable de filtrage : liaison dans un premier temps puis référence pour comparaison dans un second temps.

`(??- ?p aime ?v ??-)`

[1] Sous cet angle, l'exemple le plus complexe de l'ouvrage figure page 104 avec la macro-fonction `de-filtre6`.

`??-` est l'attrape-tout plural qui accepte toute séquence d'expressions. Le filtre tout entier n'accepte que les listes comportant quelque part en elles le symbole `aime` précédé et suivi d'au moins une expression, ces deux expressions étant extraites sous les noms `p` et `v`. Confronté à `(On dit que Paul aime Virginie la belle)`, il en extraira `p=Paul` et `v=Virginie`. Confronté à `(Pierrot aime Colombine tandis que Colombine aime Arlequin)` le filtrage donnera deux solutions : `p=Pierrot` et `v=Colombine` ou `p=Colombine` et `v=Arlequin`. On voit là se profiler simplement le non-déterminisme.

`((??- ?element ??-) (??- ?element ??-))`
: teste si un même terme apparaît en `car` et `cadr` d'une liste de deux termes.

`(??seq ??seq ??seq)`
: vérifie qu'une liste est formée de trois segments égaux, comme le sont, par exemple : `()`, `(lisp lisp lisp)` ou encore `(bi (za) ((rre)) bi (za) ((rre)) bi (za) ((rre)))`.

Le filtrage vu comme un moyen de décortiquer des données est le complément naturel du mécanisme connu sous le nom de `backquote` [Bawden & Burke & Hoffman 81]. Présent en COMMON LISPet Scheme, `backquote` permet une synthèse agréable de programmes. Un programme engendrant une forme particulière a une structure qui en est le strict décalque. On ignore ce qui est engendré pour ne s'intéresser qu'aux valeurs qui seront obtenues et qui elles, doivent être conformes à ce qui a été exprimé. L'effort est là encore minimisé entre ce que l'on désire et ce que l'on écrit : on diminue donc la distance téléologique [Wertz 84] et par là même le risque d'erreur. Voyez plutôt :

```
(defun gazonk (l)                                          gazonk
  (match-case l
   ;;Si l est ()
   ( ()    '() )
   ;;Si l a un "centre"
   ( (??x ?y ?y ??z)
         `(,y ,@(gazonk `(,@x ,@z)) ,y) )
   ;;Autrement l n'est pas acceptable
   ( ?-    (error "Pas un palindrome ~A." l)) ) )
```

Cette fonction éviscère une liste palindromique : `(gazonk '(amour delice orgue orgue delice amour))` a pour valeur `(orgue delice amour amour delice orgue)`. La construction `match-case` compare successivement son argument `l` avec chacun des filtres `()`, `(??x ?y ?y ??z)` puis `?-` et, en cas de succès, évalue l'expression associée, son résultat est alors retourné comme valeur. La macro `match-case` sera expliquée en page 57.

Lorsque l'on écrit `` `(,y ,@(gazonk `(,@x ,@z)) ,y) ``, nul ne pense au code équivalent qui est engendré : la seule connaissance de la forme du résultat importe. D'ailleurs, plusieurs programmes équivalents sont imaginables qui ont tous comme valeur ce qui est spécifié par l'expression `backquot`ée comme, par exemple :

```
(cons y
      (append (gazonk (append x z '()))
              (list y) ) )
```

ou encore :

```
(append (list y) (gazonk (append x z)) (list y))
```

De la même manière, l'écriture sous forme de filtres, diminue l'effort mental du programmeur qui, s'il exprime son souhait sous la forme `(??x ?y ?y ??x)`, ne souhaite pas connaître le code équivalent engendré. Et là encore, plusieurs programmations sont envisageables.

Résumé

Un filtre est donc une expression décrivant à la fois une forme, des vérifications à effectuer et des désignations de sous-expressions à extraire. La force du filtre est sa concision pour exprimer une forme puisque c'est cette forme même que revêt le filtre. Ces épousailles entre filtre et forme simplifient l'interprétation mentale qu'opère le programmeur à la vue du filtre et lui permettent d'appréhender sans effort la forme que doivent respecter les expressions reconnues par ce filtre.

Le degré zéro du filtrage 3

De toutes les représentations de la béatitude imaginées par les peuples et les poètes, la plus grande et la plus profonde m'a toujours semblé être celle que l'on éprouve en écoutant l'harmonie des sphères. Mon rêve le plus intime, le plus précieux fut d'entendre, pour la durée d'un battement de cœur, résonner dans leur harmonie secrète toute la structure de l'Univers et toutes les manifestations de la vie.

Hermann Hesse, *Gertrude*

Le filtre dirige l'examen de l'expression. Il contient toutes les informations dont la satisfaction sera l'objet du filtrage. Le plus simple des filtrages est donc de vérifier que l'expression est rigoureusement conforme au filtre, c'est-à-dire égale à ce même filtre !

Le codage du filtre demeure toutefois. S'agissant d'une technique employée dans des programmes, les filtres doivent s'écrire avec les mêmes conventions de représentation : listes et symboles. Nous prendrons des règles simples :

Si le filtre est un atome
l'expression doit être cet atome.
Si le filtre est une liste alors
l'expression doit être une liste
(le filtre sera alors considéré comme une liste de filtres qui se distribuera sur l'expression, elle même considérée comme une liste d'expressions).

La programmation est immédiatement décalquée de ces règles :

filtrer0

```
(defun filtrer0 (e f)
  (cond ((atom f) (equal e f))
        ((consp e)      ; Vérification de la structure de E
         (and (filtrer0 (car e) (car f))
              (filtrer0 (cdr e) (cdr f)) ) ) ) ) )
```

La fonction `filtrer0` où `e` et `f` sont, respectivement, la donnée et le filtre, est semblable au prédicat fondamental `equal` et ne vérifie que l'égalité d'un filtre et d'une expression. Lorsqu'un filtre ϕ est confronté à une expression ϵ, c'est-à-dire lorsqu'ils sont l'un et l'autre arguments d'une fonction de filtrage comme `filtrer0`, on dit que ϕ *filtre* ϵ, que ϕ *accepte* ϵ ou encore que ϵ *satisfait* ϕ si (`filtrer0` ϵ ϕ) vaut Vrai. On dit qu'il y a *échec* lorsque la valeur ramenée par (`filtrer0` ϵ ϕ) est Faux.

```
(filtrer0 '(foo (bar Quux)) '(foo (bar Quux)))
⇒ T     ; succès
```

```
(filtrer0 'nil 'nil)
⇒ T     ;succès
(filtrer0 '(blood and guts) '(bloody mary))
⇒ NIL   ;échec
```

Une simple verrue va augmenter les possibilités de **filtrer0** et permettre une recherche de quasi-égalité.

3.1 L'attrape-tout

L'ensemble des filtres va s'enrichir du filtre *attrape-tout* qui, confronté à une expression quelconque, l'acceptera dans tous les cas. Cet attrape-tout sera noté "`?-`" et permettra de spécifier, dans un filtre, des îlots de variance. Par exemple

`?-`
: filtre tout objet Lisp,

`(?- ?-)`
: filtre toute liste de deux termes quelconques,

`(?- foo ?-)`
: filtre toute liste de trois termes dont le second est le symbole `foo`.

On programmera **filtrer1** par simple ajout d'une verrue en tête, testant tout d'abord si le filtre courant et l'attrape-tout ne font qu'un.

filtrer1

```
(defun filtrer1 (e f)
  (cond ((equal f '?-) t)         ;L'attrape-tout !
        ((atom f) (equal f e))    ;et la suite
        ((consp e)                ;sans changement
         (and (filtrer1 (car e) (car f))
              (filtrer1 (cdr e) (cdr f)) ) ) ) )
```

L'attrape-tout permet de tester simplement la présence de mots-clés à l'intérieur de listes.

```
(filtrer1 '(tire bou schtroumpf) '(tire bou ?-))
⇒ T   ;Succès [Peyo 82]
(filtrer1 '(schtroumpf bouchon) '(?- bouchon))
⇒ T
(filtrer1 '(?-) '(schtroumpf)) ;Notez l'ordre des arguments !
⇒ NIL
```

3.2 Représentation des filtres

Il est temps de s'inquiéter de la nature des filtres ou plus exactement de leur représentation : comment les écrire, à quels objets Lisp correspondent-ils ? Pour l'heure, nous ne disposons que de "`?-`" mais puisque notre arsenal va bientôt s'enrichir, nous nous devons d'adopter une représentation extensible. Les filtres sont avant tout des entités programmatiques, ils n'existent qu'au sein de programmes qui les mettent en œuvre. Ils doivent donc s'écrire comme s'écrivent les programmes : de lettres, de blancs et de parenthèses. Les filtres seront donc représentés par des listes et des symboles.

Un schéma extensible et naturel en Lisp est qu'un filtre soit codé comme une liste dont le `car` est un symbole distingué nommant le filtre et dont le `cdr` contient les éventuels paramètres associés. Pour marquer encore plus leur particularité, les noms de filtres seront tous préfixés du signe "`*`". L'attrape-tout `?-` sera ainsi codé comme `(*sexp)`.

> Si `?-` est `(*sexp)` alors nous perdons une grande qualité de Lisp qui est l'art de pouvoir parler de lui-même ! En effet, tout codage qui empiète sur les listes ou les symboles pose le problème que, sans contexte pour lever le doute, certains objets possèdent au moins deux sémantiques possibles : celle naturelle des listes ou symboles, ainsi que celle artificiellement plaquée sur ces mêmes objets. Ce même problème se pose entre listes et programmes. Il est alors utile de disposer d'un moyen permettant, sans équivoque, de lever

l'ambiguïté. Entre données et programmes, `quote` fait figure de discriminateur. La virgule agit de même dans le contexte de la synthèse de liste qu'opère l'accent grave (*backquote*).
Un autre problème lié est le suivant. Comment donc écrire un filtre spécialisé dans la reconnaissance de la seule liste `(*sexp)` ? En d'autres termes, comment concevoir un filtre ϕ tel que `(filtrer1 '?- '`ϕ`)` vaille `T` et que pour toute autre expression Lisp η différente de `?-`, `(filtrer1 '`η` '`ϕ`)` retourne `NIL` ?
Filtrer des filtres[1] considérés comme des données nécessite donc un nouveau filtre spécial capable de forcer le "moteur à filtrer" à ne pas interpréter le filtre mais à le considérer comme une simple donnée. Ce nouveau filtre possède donc un argument : l'objet à considérer comme une expression. Il y a une analogie frappante avec `quote` : pour s'en convaincre, il suffit de remplacer "filtre" par "forme" et "filtrer" par "évaluer" dans les phrases précédentes. Usant de la syntaxe proposée plus haut, nous proposons donc le filtre `(*quote` *expression*`)` qui ne se satisfait que de la seule *expression*; *expression* est l'argument de ce filtre judicieusement nommé `*quote`.
Filtrer des expressions pouvant ressembler à des filtres ne posera, dès lors, plus aucun problème. Par exemple `(*quote ?-)` ne peut filtrer que la seule expression `?-`, c'est-à-dire `(*sexp)`, tandis que `(*quote (*sexp *quote))` ne peut filtrer que la liste composée des deux termes `*sexp` et `*quote` (qui d'ailleurs n'est pas un filtre légal).

La notation `(*sexp)` peut sembler lourde pour signifier seulement `?-`. Le point d'interrogation peut être redéfini et devenir un macro-caractère autorisant de nombreux raccourcis syntaxiques.

```
(set-macro-character #\? #'(lambda (stream void-ch)
   "Version de ? avec *sexp et *quote"
   (let ((e (read stream)))
        (cond ((equal e '|-|) '(*sexp))  ;?- est l'attrape-tout
              ((and (consp e)            ;?'e est l'échappement
                    (equal (car e) 'quote) )
               '(*quote ,(cadr e)) )
              (t (error "?~A : filtre inconnu" e)) ) ) ))
```

On peut préférer, pour ne pas interférer avec Scheme qui utilise le point d'interrogation comme suffixe des prédicats, une notation "`#?`*quelquechose*" moins légère mais plus portable. Enfin, puisque tous les filtres ne seront pas nécessairement pourvus d'abréviations, on peut aussi choisir de les écrire en toutes lettres et ce sera la solution retenue au chapitre final. Les "?*notations*" ne seront utilisées que pour faciliter l'écriture et la compréhension des exemples par préférence aux épuisantes "`*`*notations*" .
On peut aussi introduire une version abrégée du filtre `*quote` : `?'`*expression* sera l'abréviation de `(*quote` *expression*`)`. Ainsi écrira-t'on plus commodément `(*quote (*sexp))` en `?'?-`.

Munis de ces deux filtres, le filtrage devient

```
(defun filtrer2 (e f)                                    filtrer2
   (cond ((equal f '(*sexp)) t)          ;L'attrape-tout
         ((and (consp f)                 ;L'échappement
               (equal (car f) '*quote) )
          (equal e (cadr f)) )
         ((atom f) (equal e f))          ;et la suite
         ((consp e)                      ;sans changement
          (and (filtrer2 (car e) (car f))
               (filtrer2 (cdr e) (cdr f)) ) ) ) ) )
```

3.3 Autocritique et moteur à filtrer

Les prédicats `filtrer1` et `filtrer2` comportaient deux verrues disgracieuses. Une nouvelle version, qui suppose que d'autres filtres pourront exister, adoptera le principe d'une programmation dirigée par les données (ou *Data Driven Programming*). Ce sont les filtres eux-mêmes qui informeront le moteur à filtrer de leur méthode associée, à charge pour le dit moteur de s'assurer de la reconnaissance des filtres et de leur invocation spécifique. Ces méthodes seront codées sur la liste des propriétés du symbole `filtrer3`.

[1] De cet exercice, qui peut paraître paradoxal à première vue, le lecteur verra un grand nombre d'emplois à partir du chapitre 5.

filtrer3

```
(defun filtrer3 (e f)
    (if (atom f) (equal e f)
        (funcall
         (or ; Chercher le filtre particulier
             (if (symbolp (car f))
               (get 'filtrer3 (car f)) )
               ; Sinon chercher le traitement par défaut
               (get 'filtrer3 '*list)
               ; Ne devrait jamais être évalué :
               (error "FILTRER3 ne connait pas *LIST !!!") )
         e f ) ) )
```

Tout filtre non atomique dont le `car` n'est pas un nom de filtre spécifique est assimilé par défaut à une liste de filtres et est donc traité par le filtre `*list`. On admettra en effet que `(a b c)` est l'écriture concise du filtre régulier, explicite mais plus long à écrire, `(*list a b c)`. Il n'y a donc plus qu'un seul mécanisme pour ce simple moteur : lancer un filtre spécifique (dont le nom figure en `car` de l'expression) ou lancer, par défaut, le filtre de nom `*list`.

Les filtres partageant tous un même mode de lancement, seront définis par `de-filtre3`.

de-filtre3

```
(defmacro de-filtre3 (nom . def)
   `(progn (setf (get 'filtrer3 ',nom)
                 #'(lambda (e f) ,@def) )
           ',nom ) )
```

La macro `de-filtre3` est une abréviation commode qui met en situation le corps du filtre, codé en fait comme une fonction, et cache cette définition dans les propriétés de `filtrer3`. Redéfinissons donc les filtres naguère vus :

*sexp

```
(de-filtre3 *sexp t)
```

*quote

```
(de-filtre3 *quote (equal (cadr f) e))
```

sans oublier celui dont l'existence est nécessaire à `filtrer3` :

*list

```
(de-filtre3 *list
   (if (and (consp f) (consp e))
       ; S'il y a encore des filtres
       (and (filtrer3 (car e) (car f))
            ; (get 'filtrer3 '*list) est SELF !
            (funcall (get 'filtrer3 '*list)
                     (cdr e) (cdr f) ) )
       (equal e f) ) )
```

`*list` et `*quote` ne sont pas des filtres sans rapport. En effet, rendre explicite à tous niveaux l'appel à `*list` permet de s'affranchir de `*quote`. Par exemple, `(*list *quote (*list *sexp))`, où tous les `*list` sont explicites, ne peut filtrer que la seule expression `(*quote (*sexp))` c'est-à-dire `?'?-`. Le filtre `(*list *quote (*list *sexp))` est donc le strict équivalent de `(*quote (*quote (*sexp)))` ou sous sa forme plus abrégée `?'?'?-`.

3.4 Extensions

Une des grandes qualités de la programmation dirigée par les données est que de nouveaux filtres simples peuvent aisément être ajoutés sans modification de l'existant. Par exemple, `*even`, qui ne reconnaît qu'un nombre entier pair, sera défini comme :

*even

```
(de-filtre3 *even
   (and (numberp e)
        (evenp e) ) )
```

ou encore `*cons` qui prend deux filtres comme arguments et ne reconnaît qu'une paire pointée dont le `car` est filtrable par son premier argument et le `cdr` par son second.

```
(de-filtre3 *cons                                                    *cons
   (and (consp e)
        (filtrer3 (car e) (cadr f))
        (filtrer3 (cdr e) (caddr f)) ) )
```

Muni de ces filtres, toute liste débutant par un entier pair sera reconnue (quelque soit sa longueur) par `(*cons (*even) ?-)`.

3.5 Filtres booléens

Bien que le moteur à filtrer soit tout à fait rudimentaire, nous allons maintenant développer quelques filtres somme toute assez puissants. Le premier sera le filtre `*or` qui offre des possibilités de choix : `(*or` $filtre_1$ $filtre_2$ ... $filtre_n$`)` filtrera toute expression filtrable par l'un au moins des $filtre_1$, $filtre_2$, ..., $filtre_n$. C'est trivialement simple pour peu que l'on écrive :

```
(de-filtre3 *or                                                      *or
   (some #'(lambda (f) (filtrer3 e f))
         (cdr f) ) )
```

où `some` est un itérateur classique de la famille des `map` qui recherche si *un* terme au moins d'une liste vérifie un certain prédicat.

La sœur de `some` est `every` qui vérifie si *tous* les termes d'une liste satisfont un même prédicat. L'itérateur `every` permet de définir le filtre `*and`, frère de `*or`.

```
(de-filtre3 *and                                                     *and
   (every #'(lambda (f) (filtrer3 e f))
          (cdr f) ) )
```

`(*and` $filtre_1$ $filtre_2$ ... $filtre_n$`)` ne filtre qu'une expression filtrable par tous les $filtre_1$, $filtre_2$, ..., $filtre_n$. `(*and)` accepte donc tout objet puisqu'il n'impose aucune contrainte : `(*and)` est donc équivalent à `?-`. En revanche `(*or)` n'accepte rien et force l'échec. On reconnaîtra la parenté de ces noms de filtres avec les connecteurs logiques `and` et `or`.

On achèvera la série des filtres booléens avec, bien sûr, la négation :

```
(de-filtre3 *not                                                     *not
    (if (consp (cdr f))
        (not (filtrer3 e (cadr f)))
        (error "*NOT sans argument") ) )
```

L'arsenal de filtres ainsi constitué permet déjà quelques belles réalisations comme

`(*list ?- (*or cube sphere) (*or rouge parme garance))`
: qui filtre quelques propositions comme `(le cube rouge)` ou encore `(joe sphere garance)`.

`(*and ?- (*not (*cons ?- ?-)))`
: ne filtre, quant à lui, que les seuls atomes puisque ce sont les seuls objets qui soient tout sauf une paire pointée.

3.6 Le davier expressionnel

Avec `?-`, il était possible de décrire simplement les structures ou formes que le moteur à filtrer vérifiait. Nous allons maintenant nous intéresser à la seconde facette du filtrage : l'extraction d'informations avec, comme corollaire, la vérification de sous-structures identiques. Pour cela, nous introduirons la notion de *variables de filtrage* qui, au cours de celui-ci, nomment des parties de l'expression.

`(*term` x`)` sera la variable de filtrage de nom x, `(*term` x`)` sera plus lisiblement abrégé en "`?`x".

L'ensemble des associations que réalisera le filtrage entre les noms des variables et leur valeur formera un *environnement*. Afin qu'il puisse être exploité après et à l'extérieur du filtrage, cet environnement doit figurer quelque part dans le résultat. Coder un environnement comme une liste associant variables à valeurs est simple mais pose le problème de la distinction entre l'échec et l'environnement vide. Nous représenterons le premier par `nil` et le second par une unique association sans retentissement possible. Quelque chose comme `((#:G43 #:G43))`.

Le filtreur (ici `filtrer4`) devient donc un semi-prédicat, c'est-à-dire une fonction retournant Vrai ou Faux mais où Vrai, c'est-à-dire l'environnement de filtrage, est nettement plus informatif que le symbole particulièrement dépouillé qu'est `t`.

Une variable de filtrage confrontée à une expression acquerra pour valeur (vis-à-vis du filtrage et non, bien entendu, de l'évaluation) cette même expression. Une fois valuée, une variable de filtrage ne pourra accepter que des expressions égales (par `equal`) à sa valeur. Ainsi donc

`(paul aime ?e)`

accepte toute liste de trois termes (dont les deux premiers sont les symboles `paul` et `aime`) et retourne, en cas de succès, un environnement où la variable de filtrage `e` sera liée au troisième terme de l'expression. Par exemple

```
(filtrer4 '(paul aime boire) '(paul aime ?e))
⇒ ((E . BOIRE))
```

`(?e ?e)`

reconnaît toute liste de deux termes égaux. En cas de succès, la valeur retournée sera un environnement où `e` sera liée au `car` de l'expression originale. Par exemple

```
(filtrer4 '((virginie) (virginie)) '(?e ?e))
⇒ ((E VIRGINIE)) ;C'est aussi ((E . (VIRGINIE))) !
                 ;méfiez vous de l'imprimeur !
```

Le filtrage compare donc un filtre à une expression dans un certain environnement. Cet environnement, progressivement enrichi ou `nil` en cas d'échec, sera retourné comme valeur. Les semi-prédicats de filtrage, correspondant à chacun des filtres élémentaires comme `*sexp` ou `*and`, doivent donc comporter une variable supplémentaire `env`, contenant l'environnement, et ne plus retourner bêtement `t` ou `nil` mais bien `env` ou `nil`. La transformation est simple et quasiment automatique : là où une expression *exp* imposait sa valeur au filtrage, on la remplacera par `(if` *exp* `env nil)` ou plus rapidement par `(if` *exp* `env)` puisque le codage de l'échec n'a pas varié.

Le filtrage s'articule autour d'un couple de fonctions mutuellement récursives : `filtrer4-element` et `filtrer4-liste`. La première confronte une expression et un filtre, la seconde une suite d'expressions et une suite de filtres. De fait, `filtrer4-liste` est la méthode associée à `*list`.

```
(defun filtrer4-element (e f env)                                 filtrer4-element
   (if env
      (if (atom f) (if (equal e f) env)
         ;Chercher le filtre particulier
         (if (and (symbolp (car f))
                  (get 'filtrer4-element (car f)) )
            ;et l'appliquer
            (funcall (get 'filtrer4-element (car f))
                     e f env )
            ;sinon chercher le traitement par défaut
            (filtrer4-liste e f env) ) ) ) )

(defun filtrer4-liste (e f env)                                   filtrer4-liste
  (if env
     (if (and (consp f) (consp e))
        ;S'il y a encore des filtres
        (filtrer4-liste (cdr e) (cdr f)
           (filtrer4-element (car e) (car f) env) )
```

```
            (if (equal e f) env) ) ) )
```

Notez, dans **filtrer4-liste**, l'appel récursif imbriqué sur l'environnement qui impose de tester qu'il n'est pas **nil** en tête des définitions de **filtrer4-liste** et de **filtrer4-element**.

On introduira aussi **filtrer4** comme fonction d'interface simple :

filtrer4

```
(defun filtrer4 (e f &optional (env init.env))
   (filtrer4-element e f env) )
(defvar init.env (let ((g (gensym))) `((,g . ,g))))
```

Le définisseur de filtres sera mis à jour en **de-filtre4** qui prendra soin d'ajouter **env** à la liste des variables des méthodes spécifiques. La variable **env**, tout comme précédemment **e** ou **f** devient une sorte de mot-clé utilisable à l'intérieur d'un filtre.

de-filtre4

```
(defmacro de-filtre4 (nom . def)
   `(progn (setf (get 'filtrer4-element ',nom)
                 #'(lambda (e f env) ,@def) )
           ',nom ) )
```

Les filtres précédents s'écriront alors :

*sexp

```
(de-filtre4 *sexp env)
```

*quote

```
(de-filtre4 *quote (if (equal (cadr f) e) env))
```

*list

```
(de-filtre4 *list
    (filtrer4-liste e (cdr f) env) )
```

La raison ultime de **filtrer4** est l'introduction du filtre ***term** que voici :

*term

```
(de-filtre4 *term
  (let ((v (assoc (cadr f) env)))
    (if v
      ;On a déjà rencontré la variable
      (if (equal (cdr v) e) env)
      ;enrichir l'environnement
      (cons (cons (cadr f) e) env) ) ) )
```

On reconnaît dans les deux branches de l'alternative **if**, la liaison de la variable et sa référence.

Notre arsenal de filtres prolifère tant que c'en est merveille ! L'extraction de sous-parties est une fonctionnalité majeure comme vont le prouver les quelques exemples qui suivent.

Tester si une liste a même **car** et **cdr** est l'œuvre de :

```
(*cons ?exp ?exp)
```

Si en outre, et **car** et **cdr** doivent avoir trois termes de long, le filtre suivant fera l'affaire :

```
(*and (*cons ?exp ?exp)
      (*list ?- ?- ?- ?-) )
```

Mettons le à l'épreuve :

```
(filtrer4 '((bar abbas) bar abbas)
          '(*and (*cons ?e ?e) (*list ?- ?- ?-)) )
⇒ NIL ;échec
(filtrer4 '((Perrier c-est foo) Perrier c-est foo)[2]
          '(*and (*cons ?e ?e) (*list ?- ?- ?-)) )
⇒ ((E PERRIER C_EST FOO)) ;Succès !
```

Reconnaître les alexandrins palindromiques est aussi trivial avec :

```
(*list ?un ?deux ?trois ?quatre ?cinq ?six
       ?six ?cinq ?quatre ?trois ?deux ?un )
```

[2] J'ai souvent lu que les auteurs étaient abreuvés des marques qu'ilscitaient. Je commence donc, mais modestement !

3.7 Exploiter les résultats

Le problème maintenant est de pouvoir exploiter aisément le résultat d'un filtrage. Scruter une liste d'association est une activité pénible et opaque. Il est plus naturel que les valeurs de filtrage deviennent les valeurs des variables de même nom. La forme spéciale **progv** permet que soit converti un environnement de filtrage réussi en un environnement dynamique d'évaluation. Rappelons que **progv** prend en premier argument une liste de symboles et en second argument une liste de valeurs, **progv** lie alors terme à terme et dans l'environnement dynamique, les symboles aux valeurs puis évalue son corps c'est-à-dire tous ses paramètres à partir du troisième, dans ce nouvel environnement.

```
(defmacro match-let (filtre exp . body)                          match-let
   (let ((env (gensym)))
     `(let ((,env (filtrer4 ,exp ,filtre)))
         (if ,env
             (progv (mapcar #'car ,env)
                    (mapcar #'cdr ,env)
                 ,@body ) ) ) ) )
```

Cette macro est un peu longue, apprécions-la mieux à l'œuvre :

```
(match-let '(*list ?d ?d ?d et ?d)
           '(Chester Chester Chester et Chester)
   `(son nom est ,d) )
⇒ (SON NOM EST CHESTER) ; [Morris & Goscinny 73]
```

Le filtrage retourne la liste `((D . CHESTER))` qui est démembrée par **progv** pour lier la variable dynamique `D` à sa valeur `CHESTER`. Toute référence à `D` dans le corps de **match-let** aura pour valeur le symbole `CHESTER`.

Un exemple plus intéressant est la définition du définisseur de filtres lui-même en terme de filtres : **de-filtre4**

```
(defmacro de-filtre4 appel                                       de-filtre4
   (match-let (*cons (*and ?nom (*not (*cons ?- ?-)))
                     (*and ?def (*cons ?- ?-)) )
              appel
         `(progn (setf (get 'filtrer4-element ',nom)
                       #'(lambda (e f env) ,@def) )
                 ',nom ) ) )
```

Le définisseur **de-filtrer4** vérifie que la forme d'appel est correcte, c'est-à-dire comporte un nom atomique et une définition comprenant au moins une forme.

3.8 Variantes

A — Nous avons adopté une représentation uniforme pour les filtres qui nous a conduit à coder les variables de filtrage par des listes (***term** *nom-de-la-variable*). Dans d'autres littératures, celles-ci sont souvent aussi codées comme des symboles dont le premier caractère est "`?`". Le symbole `?foo` est alors la représentation de la variable de filtrage `foo`. La reconnaissance qu'un tel symbole représente une variable de filtrage ne peut être efficace que si l'on sait accéder rapidement au premier caractère qui compose le nom d'un symbole. Les noms des symboles étant immuables, ils sont souvent archivés dans des zones particulières dont l'accès augmente le nombre de défauts de page dans le contexte d'une mémoire virtuelle. En outre on brise l'uniformité du codage ici utilisé et l'on introduit donc des cas particuliers dans le traitement des filtres.

B — Certains filtreurs différencient l'affectation d'une variable de filtrage (notée `>foo`) de son utilisation postérieure (notée `<foo`) ce qui revient à éclater en deux filtres différents les deux branches de l'alternative présente dans ***term**. Pour notre part, nous préférons, pour l'instant, ne pas multiplier les notations et

n'utiliser que le seul filtre `*term` qui, de plus, n'impose pas d'ordre pour le filtrage : à l'inverse, `>foo` doit toujours précéder `<foo`. Cette différence entre liaison et référence sera cependant exploitée au chapitre 5.

C — Les méthodes de filtrage sont archivées sur la liste de propriétés de `filtrer4-element`. C'est pédagogique mais inefficace puisque le traitement de tout filtre impose le balayage partiel de cette liste de propriétés et conduit à un coût d'autant plus exorbitant qu'il existe de propriétés différentes. On peut dans un premier temps renverser `(get 'filtrer4-element (car e))` en `(get (car e) 'filtrer4-element)` qui, statistiquement, conduit à des recherches plus courtes car les P-listes des noms de filtres sont sûrement moins encombrées que celle de `filtrer4-element`. Une autre solution est de recourir à des tables de hachage. La comparaison entre ces deux solutions dépend du nombre de filtres présents. Les tables de hachage ont un coût minimal plus important que celui d'une recherche linéaire parmi une ou deux propriétés. Dans tous les cas, il est bon d'encapsuler la définition, la recherche et l'invocation des méthodes de filtrage.

À essayer : *Encapsuler les méthodes de filtrage*

D — La macro `match-let` assimile les variables de filtrage à des variables dynamiques. Cette identification peut toutefois engendrer quelques problèmes de références, comme dans :

```
(dynamic-let ((x 1) (y 2))
     ;;on ajouterait (declare (special x y)) en COMMON LISP
     (match-let '(*or ?x ?y)
                'xyzzy
           (list x y) ) )
```

Dans le corps de `match-let`, `x` et `y` ne peuvent être l'une et l'autre liées par `progv` puisque le filtrage n'en lie qu'une seule suivant l'ordre dans lequel il considère les filtres `?x` et `?y`. Du fait de la double référence présente dans le corps de `match-let`, il y a capture inopinée d'une autre variable dynamique extérieure de nom `x` ou `y`.

E — Ce précédent problème se serait posé de façon encore plus accrue si l'on avait codé l'environnement `env`, dans le cours même du filtrage, par une des propriétés naturelles des symboles accédés : par exemple, `(symbol-value` *symbole*`)` ou `(get` *symbole* `'*term)`. On aurait pu écrire, dès `filtrer3` :

```
(de-filtre3 *term                                                    *term
   (if (boundp (cadr f))
       ;On a déjà rencontré la variable
       (filtrer3 e (symbol-value (cadr f)))
       ;enrichir l'environnement (gare au SET !!!)
       (progn (set (cadr f) e)
              t ) ) )
```

```
(de-filtre3 *term                                                    *term
   (let ((v (get '*term (cadr f))))
        (if v
            ;On a déjà rencontré la variable
            (filtrer3 e v)
            ;enrichir l'environnement
            (setf (get '*term (cadr f)) e) ) ) )
```

Cette programmation pose des problèmes de filtrages au sein de filtrages (i.e. des filtrages récursivement emboîtés) ainsi que de remise à *indéfini* des propriétés utilisées. Par exemple :

```
(progn (setf (get '*term 'x) 'inus)
       (filtrer4 '(term inal) '(?- ?x)) )
⇒ NIL ;?!
```

alors que dans un autre contexte, l'on obtiendrait probablement `((X . INAL))`. Cette méthode qui rend le filtrage contextuel sans que l'on puisse maîtriser ce contexte n'a donc pas été retenue.

Résumé

Ce chapitre a permis d'exhiber une première architecture de filtrage fondée sur un moteur à filtrer. Cet exemple de programmation, dirigée par les données, illustre bien les capacités modulaires de cette approche :

extensibilité : le moteur supporte sans changement de nouveaux filtres,

modularité : les filtres ainsi ajoutés sont indépendants les uns des autres. Le moteur ne fixe que le protocole de lancement des filtres particuliers.

Plusieurs réalisations par liste de propriétés ou table de hachage sont, de plus, possibles.

Ces divers essais ont aussi révélé l'intérêt de l'encapsulation des principales structures de données du moteur à filtrer ainsi qu'exposé une première approche des problèmes d'efficacité.

Exercice 3.1 : Écrire le filtre `*atom` qui ne reconnaît qu'un atome.

Exercice 3.2 : Généraliser le filtre `*cons` en `*mcons`. On désire que `(*mcons` filtre_1 filtre_2 ... filtre_n`)` soit équivalent à `(*cons` filtre_1 `(*cons` filtre_2 ...`(*cons` filtre_{n-1} filtre_n`)` ...`))`. Bien entendu, `(*mcons` *filtre*`)` ne doit être autre que *filtre*.

Exercice 3.3 : À l'instar de `*cons` qui filtre des paires pointées, écrire `*vector` qui filtre des vecteurs. `(*vector` filtre_1 filtre_2 ... filtre_n`)` n'accepte que des vecteurs de n composantes, chaque composante devant être acceptée par le filtre de rang correspondant. Ainsi `(*vector ?x (*quote 2) ?x)` filtre-t'il avec succès le vecteur `#3(1 2 1)` mais par contre refuse `#(a b c)`.

Exercice 3.4 : Écrire le filtre `*struct` reconnaissant des stuctures plus générales que des paires pointées. Par exemple `(*struct hux :foo ?x :bar ?x)` reconnaît toute instance de `hux` (ou de ses sous-types) dont les champs `foo` et `bar` sont les mêmes. Les autres champs, s'il y en a, peuvent être quelconques.

Exercice 3.5 : Adapter les filtres `*or` et `*and` pour `filtrer4`.

Exercice 3.6 : Écrire une nouvelle version de `filtrer4` avec table de hachage. En profiter pour encapsuler les méthodes.

Les segments 4

Au moment où je pensais cela, une mesure de la Sonate me frappa, mesure que je connaissais bien pourtant, mais parfois l'attention éclaire différemment des choses connues pourtant depuis longtemps et où nous remarquons ce que nous n'y avions jamais vu.

Marcel Proust, *À la Recherche du Temps Perdu*

Les filtres précédemment introduits ne pouvaient reconnaître que des objets Lisp, c'est-à-dire des listes ou des atomes, bref de simples expressions. On désire maintenant développer des filtres capables de reconnaître des suites quelconques de zéro, un ou plusieurs objets quelconques. Une suite d'objets ou *segment* n'est pas un objet Lisp et n'existe pas en tant que tel : ce n'est qu'une suite de termes contigus d'une liste.

On s'attachera dans un premier temps à définir le filtre de segment anonyme "`??-`" puis le segment nommé "`??x`" qui généralise la notion de variables de filtrage.

4.1 Les segments anonymes

La fonction `member` est bien connue pour rechercher un segment terminal de liste repéré par un symbole. Un filtre spécial `(*any)`, ou en abrégé "`??-`", permettra de réaliser un effet similaire. L'expression :

```
(match-let '(??- :nom ?nom ??-)
           expression
     nom )
```

signifie, à peu près :

```
(cadr (member ':nom expression))
```

Mais une traduction plus fidèle serait :

```
(let ((l (memq ':nom l)))
     (if (consp l)
         (if (consp (cdr l))
             (cadr l) ) ) )
```

On notera les différences de taille entre ces deux spécifications. Plus intéressante encore devient la comparaison, si l'on fait légèrement évoluer le filtre. Supposons que l'objet recherché soit préfixé de deux symboles et non plus d'un seul, comme dans le filtre `(??- paul aime ?qui ??-)`. Le code associé pourrait être :

```
(lambda (l)
   (let ((l (memq 'paul l)))
     (if (consp l)
       (if (consp (cdr l))
         (if (equal (cadr l) 'aime)
           (if (consp (cddr l))
```

```
          (cons (cons 'qui (caddr l)) t) ;l'environnement
          nil )  ;Échec du genre (??- frederic aime)
        nil )    ;Échec du genre (??- paul rêve a ??-)
      nil )      ;Échec du genre (??- martine)
    nil ) ) )    ;Échec du genre (??- pierre et anne)
```

Cette simple modification a plus que doublé la taille du code équivalent ! Plus complexe encore serait de vouloir extraire l'objet qui précède le mot "`aime`" comme dans le filtre `(??- ?qui aime ?- ??-)`. Une solution, quoique peu efficace, serait :

```
(labels ((look (l)
           (if (consp l)
               (if (consp (cdr l))
                   (if (eq (cadr l) 'aime)
                       (if (consp (cddr l))
                           (car l)
                           (look (cdr l)) ) ) ) ) ))
        (look l) )
```

On retiendra de ce qui précède qu'ô combien plus simples et plus aisément modifiables sont les filtres, comparés aux programmes !

`??-` est le segment anonyme de liste et l'abréviation de `(*any)`. Désignant une suite quelconque d'objets, il ne peut être employé qu'à l'intérieur d'une liste de filtres. Confronté à une suite d'expressions, il accepte un nombre quelconque de ces expressions. C'est ce que rappelle son nom : `??-`, où les deux points d'interrogation marquent l'éventuelle multiplicité des termes filtrables[1].

Donnons quelques autres exemples d'emploi de `??-`

`(??-)`
: reconnaît toutes les listes.

`(??- ??-)`
: est analogue à `(??-)`.

`(??- foo ??-)`
: filtre toutes les listes contenant le symbole `foo`.

`(??- ?x ??- ?x ??-)`
: reconnaît toutes les listes contenant deux fois un même terme. Ce filtre est largement plus descriptif que le, pourtant élégant, prédicat suivant :

```
;;;teste si L :: (??- ?x ??- ?x ??-)
(de am1mt (l)                                              am1mt
    (if (consp l)
        (or (member (car l) (cdr l) :test #'equal)
            (am1mt (cdr l)) ) ) )
```

4.2 Lancement des filtres de segments

Puisque `*any` ne s'occupe que de l'intérieur des listes, sa programmation ne nécessite qu'une légère modification de la fonction qui filtre l'intérieur des listes. La fonction `filtrer4-liste` sera doté de la même structure que `filtrer4-element`, c'est-à-dire :

reconnaître un filtre et le lancer,
ou se rabattre sur le traitement par défaut.

[1] La littérature nomme souvent ce filtre "`!-`" où le point d'exclamation préfixe les filtres-segments. On trouve cependant une autre littérature, tout aussi répandue, où les sens de `?` et `!` sont rigoureusement inverses ! Nous avons donc convenu de préfixer les filtres ne pouvant filtrer qu'un unique objet par un seul point d'interrogation et ceux pouvant prendre plusieurs objets (un segment) par deux points d'interrogation.

On introduit donc la notion de *filtres de segment* que reconnaîtra et lancera `filtrer44-liste` par opposition aux *filtres d'éléments* que reconnaît et lance `filtrer4-element`. Voici donc la nouvelle version de **`filtrer4-liste`**, nommée `filtrer44-liste` :

```
;;;remplace filtrer4-liste
(defun filtrer44-liste (e f env)                                  filtrer44-liste
   (if env
     (if (consp f)
       ;Le premier filtre de F est-il un filtre spécial ?
       (if (and (consp (car f))
                (symbolp (caar f))
                (get 'filtrer44-liste (caar f)) )
         ;le lancer !
         (funcall (get 'filtrer44-liste (caar f))
                  e f env )
         (if (consp e)
           ;sinon faire comme avant.
           (filtrer44-liste (cdr e) (cdr f)
             (filtrer44-element (car e) (car f) env) ) ) )
       (if (equal e f) env) ) ) )
```

Les filtres de segments qui sont lancés par **`filtrer44-liste`** sont des λ-termes de forme `(lambda (e f env) ... )`. Le filtre de segment ainsi lancé peut connaître sa propre forme d'appel qui se situe en `(car f)`. Le contexte droit du filtre, c'est-à-dire les filtres qui le suivent, est en `(cdr f)`.

Le premier filtre de segment sera `*any`.

Pour définir `*any`, on aura bien sûr étendu le définisseur de filtres **`de-filtre4`** qui trouvera, en second paramètre, le type (`element` ou `segment`) du filtre à définir. `*any` sera donc :

```
(de-filtre44 *any segment                                         *any
    (if (consp (cdr f)) ;*any n'est pas terminal
        (or (filtrer44-liste e (cdr f) env)
            (if (consp e)
                (filtrer44-liste (cdr e) f env) ) )
        ;*any est terminal : il prend tout le reste
        (if (null (cdr f)) env
            ;Attention aux ultimes CDR !
            (if (equal e (cdr f)) env) ) ) )
```

La définition de `*any` mérite quelques explications car on peut distinguer deux cas selon que `*any` est employé en cours ou en queue de filtre. Si `*any` est terminal, comme dans `(??-)`, alors il accepte toute expression. Le second cas est un peu plus complexe. Si `*any` n'est pas terminal alors (`??-` *autres-filtres*) confronté à l'expression (*car . cdr*) ne sera couronné de succès que si (*autres-filtres*) filtre (*car . cdr*) tout entière, auquel cas `??-` n'a rien pris du tout ou si (`??-` *autres-filtres*) filtre *cdr*, auquel cas `??-` a pris un segment initial de l'expression et au moins *car*.

La programmation du filtre `*any` est récursive car, se laissant lui-même en tête du filtre, il force son réappel ultérieur. En termes plus formels :

$$\begin{array}{lcl} \texttt{??-}\ \textit{autres-filtres} & ::= & \textit{autres-filtres} \\ & | & \texttt{?-}\ \texttt{??-}\ \textit{autres-filtres} \end{array}$$

Ces deux équations caractérisent la programmation de `*any`.

> Pour ceux qui aiment manipuler des listes pointées, ils remarqueront que `(??- . foo)` ne filtre que des listes d'ultime `cdr` égal à `foo`. Il y a cependant une légère distorsion de notation car `(??-)` est aussi `(??- . nil)` qui ne vérifie pas, dans la programmation ci-dessus, que l'expression est une vraie liste i.e. que l'ultime `cdr` est bien `nil`.
>
> *Quand donc échappera-t'on à l'odieuse tyrannie des paires pointées !*
>
> Pour pallier ce dysfonctionnement, il suffit d'écrire un filtre spécial (par exemple **`*proper-list`**, cf. 4.6.8) qui n'accepte que des véritables listes.

À essayer : `?-` $\not\equiv$ `(??-)` $\equiv$ `(??- . nil)` *?*

4.3 Encore d'autres filtres

L'intérêt de cette légère variante que constitue `filtrer44-liste` ne se limite pas au seul `*any`, elle permet d'introduire de nombreux, nouveaux et intéressants filtres de segments mais, en contrepartie, soulève un grave problème.

A — Voici un premier filtre technique nommé `*progn` :

`*progn`

```
(de-filtre44 *progn segment
    (filtrer44-liste e (append (cdar f) (cdr f)) env) )
```

`(*progn` *filtre*$_1$ *filtre*$_2$ ... *filtre*$_n$`)` permet de grouper un ensemble de filtres (éléments ou segments) en un unique filtre de segment. Comme `*any`, `*progn` ne peut être employé qu'au sein d'une liste.

`(foo (*progn bar bar) ian)`
est équivalent à `(foo bar bar ian)`

`(*list ?x (*progn (*progn ?x) ?x))`
est analogue à `(?x ?x ?x)`

B — Le filtre technique `*progn` est surtout utile à `*times` que voici

`*times`

```
(de-filtre44 *times segment
    (or (filtrer44-liste e (cdr f) env)
        (filtrer44-liste e (append (cdar f) f) env) ) )
```

Le filtre *n*-aire `(*times` *filtre*$_1$ *filtre*$_2$ ... *filtre*$_n$`)` est l'équivalent du filtre unaire `(*times (*progn` *filtre*$_1$ *filtre*$_2$ ... *filtre*$_n$`))`. Les arguments de `*times` subissent donc un `*progn` implicite. Le rôle de `*times` est de reconnaître un nombre quelconque de fois ses arguments. Par exemple

`(*list (*times ?x))`
filtre toute liste composée de termes égaux comme `(foo foo)` , `((bar) (bar) (bar))` ou encore `()`.

```
(*list (*times (*cons ?- ?-))
       (*and ?atome (*not (*cons ?- ?-)))
       ??- )
```
lie `atome` au premier terme atomique d'une liste.

`(*any)`
n'est autre que `(*times (*sexp))` !

C — On peut aussi étendre le filtre `*or` pour être un filtre de segment que, pour ne pas confondre avec le filtre d'élément, on nommera `*oorr`.

`*oorr`

```
(de-filtre44 *oorr segment
   (if (consp (cdar f))
     (some #'(lambda (f1)
               (filtrer44-liste e (cons f1 (cdr f)) env) )
           (cdar f) )
     (filtrer44-liste e (cdr f) env) ) )
```

Quelques exemples apparaissent plus bas.

Notre arsenal devient maintenant impressionnant ! Il est quasiment complet et permet les filtres suivants qui tous savent reconnaître diverses propositions du Tendre :

`(??- ?qui (*or aime adore idolâtre) ?quoi ??-)`
reconnaît entre autres `(Déjanire aime Nessus)` et `(Le puissant Hercule idolâtre Déjanire sans raison)`.

`(??- ?qui (**oorr aime (*progn est fou de)) ?quoi ??-)`
reconnaît `(Victor est fou de Victoria)`.

```
(??- ?qui aime (*times (*oorr (*progn un peu)
                              (*or beaucoup passionnément)
                              (*progn à la folie) )
                       (*oorr et (*progn)) )
               ?quoi ??- )
```
accepte `(elle aime à la folie et passionnément la sieste)`.

D — La puissance de ces filtres peut aussi être appréciée lorsque l'on remarque que `*times` et `*oorr` ne sont que des arrangements de filtres utilisant le seul `*or` !

`*times`

```
(de-filtre44 *times segment
   (filtrer44-liste e
     `((*oorr (*progn (*progn . ,(cdar f))
                      . ,f )     ; Récursif !!!
              (*progn . ,(cdr f)) ))
     env ) )
```

La définition de `*times` repose sur l'identité :

(`*times` *filtre*$_1$ *filtre*$_2$ ... *filtre*$_n$) ≡
(`*oorr` (`*progn` *filtre*$_1$ *filtre*$_2$ ... *filtre*$_n$
(`*times` *filtre*$_1$ *filtre*$_2$ ... *filtre*$_n$))
(`*progn`))

et si l'on fait explicitement figurer le contexte droit du filtre `*times`, on retrouve la définition sus-mentionnée.

Le filtre `*oorr` met en œuvre les identités suivantes

((`*oorr`) *filtre*$_1$ *filtre*$_2$ ... *filtre*$_n$) ≡ (`*list` *filtre*$_1$ *filtre*$_2$ ... *filtre*$_n$)
((`*oorr` *filtre*$_1$...*filtre*$_i$) *filtre*$_{i+1}$...*filtre*$_n$)
≡ (`*or` (`*list` *filtre*$_1$ *filtre*$_{i+1}$...*filtre*$_n$)
(`*list` (`*oorr` *filtre*$_2$ ... *filtre*$_n$) *filtre*$_{i+1}$...*filtre*$_n$)

ce qui se traduit donc immédiatement en :

`*oorr`

```
(de-filtre44 *oorr segment
    (filtrer44-element e
                  (if (consp (cdar f))
                      `(*or (*list ,(cadar f)
                                   . ,(cdr f) )
                            (*list (*oorr . ,(cddar f))
                                   . ,(cdr f) ) )
                      ;(*OORR) se réécrit en :
                      `(*list . ,(cdr f)) )
                  env ) )
```

Ces dernières écritures qui consistent à réécrire un filtre en une combinaison d'autres, introduisent un mécanisme comparable à une sorte de macro-expansion.

À essayer : *macro-filtre*

4.4 Les problèmes

La fonction `filtrer44-liste` ne sait pas manipuler des segments de liste non anonymes. À l'instar des variables de filtres qui peuvent accepter un objet, pourraient exister des variables de filtres qui accepteraient des segments de liste. Ainsi

```
(??- paul ??verbe virginie ??-)
```

serait un filtre qui extrairait, sous le nom `verbe`, ce que `paul` peut bien faire à `virginie`, c'est-à-dire ... les expressions situées entre ces deux symboles. Confronté à `(on dit que paul est follement amoureux de virginie la belle)` `verbe` aurait pour valeur, vis-à-vis du filtrage, le segment de liste "... `est follement amoureux de` ...". L'architecture actuelle n'autorise pas cette extension car s'est subrepticement introduit le non-déterminisme !

Depuis l'apparition de `*or` et de `*any`, les filtrages peuvent conduire à de multiples solutions. Par exemple, le filtre `(??- ?x ??-)` confronté à `(virginie aime valmont)` (*sic*) possède trois solutions : `x` = `virginie`, `x` = `aime` et `x` = `valmont`. Même `filtrer3` permettait déjà d'écrire :

```
(filtrer3 '(foo) '(*or ?- (foo) (?-)))
```

ce qui donnait trois façons de filtrer victorieusement une si piètre expression, alors que nous nous serions bien contentés d'une seule solution !

Le moteur `filtrer44` ne recherche que la première solution et ne procure aucun mécanisme permettant de relancer les calculs pour obtenir les suivantes. `filtrer44` peut même ne pas trouver la première solution : par exemple, `((??- ?x ??-) (??- ?x ??-))` n'accepte pas `((f o o) (b o r))` ! La programmation de `*any` n'est pas en cause (on verra par la suite qu'elle est correcte), ce n'est que l'architecture du moteur entier qui est mauvaise. La solution passe par l'explicitation des problèmes à résoudre et leur manipulation explicite. Il faut donc abandonner la "pile" pour une codification des continuations (d'échec ou de succès).

La nouvelle version du moteur à filtrer procurera les segments nommés, interprètera correctement les filtres tels que `((??- ?x ??-) (??- ?x ??-))` qui nécessitent un certain nombre de retours arrière avant de trouver un terme commun au `car` et au `cadr` d'une expression. Ce problème ne peut être résolu par `filtrer44`, néanmoins toutes les idées de nouveaux filtres, ici introduites, subsisteront par la suite.

Exercice 4.7 : Définir le filtre `*maybe` qui accepte que son argument soit présent ou pas. Par exemple,

```
(*list (*times (*or Pierre Paul Jean Vincent)
               (*maybe et) )
       (*maybe et les autres) )
```

accepte des listes de prénoms éventuellement séparés par la copulative `et` et non moins éventuellement suffixés par `(et les autres)`.

Exercice 4.8 : `(??-)` est-il analogue à `(*or () (*cons ?- ?-))` ou à `(?-)` ?

Exercice 4.9 : Analyser pourquoi échoue le filtrage suivant :

```
(filtrer44 '((f o o) (b o r)) '((??- ?x ??-) (??- ?x ??-)))
```

4.5 Le moteur à double continuation

Nous voici mûrs pour aborder la description d'un moteur à filtrer procurant des segments nommés et doté des derniers raffinements de la technique. Nous nous livrerons en surplus à quelques variations en quête d'efficacité et de fonctionnalité.

L'analyse de `filtrer4` montre que les filtrages élémentaires (`*sexp`, `*term` ou `*even`) sont la plupart du temps triviaux mais que, par contre, leur enchaînement obéit à des enchevêtrements complexes de `progn`, `or`, `if` ou `and`. Les voies à suivre en cas d'échec ou de succès sont implicites dans le code Lisp et ne peuvent être explicitement manipulées. En outre les filtrages sont réduits à leur première solution dite "*gauche courte*" [Gengler 88] et il n'est pas possible de les obtenir toutes. Il est cependant vrai que la tradition Lisp pousse fortement à n'utiliser que cette première solution (cf. chapitre 7)

Les segments nommés ne sont pas les seuls à introduire de multiples solutions. Déjà les filtres `*or` et `*any` étaient responsables de cette immixtion du non-déterminisme.

`(*or ?- ?-)`
: filtre tout et n'importe quoi mais de deux manières différentes : l'une grâce à `?-` (celui de la première branche de l'alternative), l'autre grâce à `?-` (celui de la seconde branche).

`(??- ?- ??-)`
: reconnaît toute liste non vide et offre n solutions lorsqu'est filtrée une liste de n termes.

`((??- ?x ??-) (??- ?x ??-))`
: Confronté à `((p i) (p h i))` possède deux solutions `x` = `p` et `x` = `i`. Confronté, par contre, à `((p i e r r e) (c l a i r e))` il produit cinq solutions `x` = `i`, `x` = `e`, `x` = `r`, `x` = `r` et `x` = `e`. On voit aussi ici la multiplicité de certaines solutions (`e` et `r`).

  La solution gauche-courte est `x` = `i`. Elle est obtenue lorsque le `??-` le plus à gauche est le plus court possible, c'est-à-dire vaut `(p)`. Inversement la solution `x` = `e` (la seconde fois que l'on rencontre `e`) est dite gauche-longue car alors le `??-` le plus à gauche est le plus long possible, c'est-à-dire vaut `(p i e r r)`.

Le jeu de fonctions `filtrer5` et consorts obéit à quelques grandes lignes.

- Tous les filtres sont encapsulés : créateur et sélecteurs sont engendrés par des définitions de structures.
- Les continuations sont explicites.
- Enfin le confort d'emploi du nouveau définisseur de filtres `de-filtre5` est accru.

Les fonctions `filtrer5` et associées réifient[2] les continuations, c'est-à-dire les suites de calculs qui restent à effectuer tant en cas d'échec (pour retrouver une autre ébauche de solution) qu'en cas de succès (pour poursuivre son avantage). À tout calcul présent dans le moteur à filtrer, seront associées deux continuations : l'une d'échec et l'autre de succès. Dès qu'un calcul élémentaire sera achevé et suivant son résultat, une des deux continuations sera choisie puis lancée.

Les deux situations importantes sont :

- *tenter tel filtrage OU tel autre.* Le second filtrage sera codé comme la continuation d'échec du premier. Si celui-ci échoue, alors le second sera tenté. L'alternative (OU) est codée par la continuation d'échec.
- *Effectuer tel filtrage PUIS enchaîner sur tel autre.* Le second filtrage sera codé comme la continuation de succès du premier. Lorsque celui-ci sera achevé, on débutera tout aussitôt le second. Les séquences (PROGN, AND) sont codées par des continuations de succès.

---

[2] de *res facire*, ("rendre chose") qui consiste à donner le statut d'objet Lisp à une entité généralement implicite et d'une classe autre que première.

Comme lors du passage de `filtrer3` à `filtrer4` où `T` s'était transformé en `env`, les succès seront marqués, dans `filtrer5`, par un appel à `reussir5` tandis que les échecs recourront à `reprendre5`. Là où l'on retournait **`env`** pour valeur, l'on écrira maintenant **`(reussir5 env succ alt)`** qui lancera la continuation de succès **`succ`**. Là où l'on retournait le signe de l'échec : **`nil`**, l'on écrira **`(reprendre5 alt)`** qui lancera la continuation d'échec **`alt`**.

Donnons de suite la définition du couple fondamental **`filtrer5-element`** et **`filtrer5-segment`** :

`filtrer5-element`

```
(defun filtrer5-element (e f env succ alt)
    (if (filtre5type-p f 'element5)
        (lancer5 (obtenir5filtre (car f)) e f env succ alt)
        ; Traitement par défaut :
        (if (atom f) (if (equal e f)
                         (reussir5 env succ alt)
                         (reprendre5 alt) )
            (filtrer5-segment e f env
                 (et-puis5 (e env alt)
                          (if (null e)
                              (reussir5 env succ alt)
                              (reprendre5 alt) ))
                 alt ) ) ) )
```

`filtrer5-segment`

```
(defun filtrer5-segment (e f env succ alt)
   (if (consp f)
       (if (filtre5type-p (car f) 'segment5)
           (lancer5 (obtenir5filtre (caar f))
                e f env
                (et-puis5 (e env alt)
                   (filtrer5-segment e (cdr f)
                                     env succ alt ) )
                alt )
           ; Traitement par défaut :
           (if (consp e)
             (filtrer5-element (car e) (car f) env
                 (et-puis5 (env alt)
                    (filtrer5-segment (cdr e) (cdr f)
                                      env succ alt ) )
                  alt )
             (reprendre5 alt) ) )
       ; On a achevé la liste de filtres
       (continuer5 e env succ alt) ) )
```

Le lecteur attentif y retrouvera les grandes lignes de `filtrer4` sauf que le schéma de traitement des filtres d'élements, procédant du paradigme de la programmation dirigée par les données, y a été étendu aux filtres de segment et que tout succès est marqué par un appel à `reussir5` (ou `continuer5`) tandis que tout échec se signale par un appel à `reprendre5`.

Les sections qui suivent passeront en revue tous les détails des codages utilisés et permettront d'appréhender dans leur plénitude tous les éléments présents dans les définitions du couple fondamental : **`filtrer5-element`** et **`filtrer5-segment`**. Tous les codages seront effectués par des objets. Le concept d'objet s'introduit naturellement puisque les filtres s'ordonnent en une hiérarchie de classes (cf. figure 4.1). De communs comportements existent qui s'accommoderont parfaitement d'être des méthodes. Nous utiliserons une sympathique couche d'objets dénommée "Picolo" et due à Vincent Delacour et Nitsan Séniak [Delacour & Séniak 89]. Nous ne détaillerons que trois seules macro-fonctions que nous emploierons, la syntaxe étant très proche de cell de CLOS [Steele 90].

`defgeneric`

(defgeneric *nom* (*variables...*)
    [:argument-precedence-order *variable*])

Une fonction générique est une fonction qui analyse la classe de son premier argument et cherche la méthode appropriée à son traitement. Les fonctions génériques ont été introduites par Common Loops

[Bobrow *et al.* 86] et réalisent l'équivalent fonctionnel de l'envoi de messages. Il est possible d'imposer que ce soit une autre variable que la première qui soit analysée par la fonction générique. On nomme cette variable : *variable de discrimination*, elle peut être citée par la directive `:argument-precedence-order`. Lorsqu'une fonction générique est créée, elle ne comporte aucune méthode. Tout appel à cette fonction générique conduira à une erreur qui sera dûment signalée.

(defmethod *nom* ((*variable classe*) ...)
définition) `defmethod`

Il est possible d'ajouter une méthode particulière sur une fonction générique grâce à **defmethod**. La méthode a *définition* pour corps et sera activée lorsque la variable de discrimination aura *classe* pour nature.

L'emploi de `(call-next-method)` dans le corps d'une méthode correspond à l'envoi du message courant au même récepteur considéré comme appartenant à la super-classe.

(defclass *nom* [:super *super-classe*]
[:slots (*champs...*)]
[:methods (*méthodes...*)]) `defclass`

Une nouvelle classe est définie par `defclass`. À part son nom, tous ses autres attributs sont facultatifs et introduits par mots-clés. Ses champs propres, outre ceux dont elle hérite de la super-classe, sont ceux figurant après le mot-clé `:slots`. On notera que la définition des méthodes pouvant opérer sur les instances de la classe doivent obligatoirement figurer après le mot-clé `:methods`. Cela simplifiera grandement la compréhension des programmes qui suivent.

Comme pour les structures, tous les accesseurs en lecture et en écriture (par `setf`) aux champs des instances de la classe sont automatiquement créés. Leurs noms sont formés par la concaténation du nom de la classe, d'un tiret et du nom du champ. Par exemple, `(defclass foo :slots (a b))` définit implicitement le prédicat `foo-p`, le constructeur `make-foo`, les accesseurs `foo-a` et `foo-b`.

Les capacités attendues du système d'objet sont faibles au regard de ce qu'ils savent procurer, néanmoins ces légers désirs se peuvent programmer aisément dans n'importe quel langage d'objets normal. Aucun exemple n'est ici donné car les sections qui suivent en sont bourrées.

4.5.1 Les continuations

Les continuations d'échec (resp. de succès) sont bâties par les macros `ou-sinon5` (resp. `et-puis5`) et lancées par `reprendre5` (resp. `reussir5`). Diverses solutions existent pour le codage de ces continuations. Les Lisps modernes permettent que soient utilisées des fermetures, ce qui confère un statut de fonction à ces continuations conformément à la philosophie qui les fit inventer [Strachey & Wadsworth 68]. Toutefois l'encapsulation complète pratiquée ici permet que ce soit aussi des objets.

Plus précisément, nous présenterons des définitions de continuations dans lesquelles subsistent des éléments de mise au point. Il est assez important que les techniques de test soient conçues en même temps que l'écriture des programmes qu'ils mettent à l'épreuve. En programmation par continuations, l'erreur la plus courante consiste à se tromper de continuation. Et plus on en manipule, plus les risques de confusion augmentent. Dans `filtrer5`, existent trois types de continuation

nature	créée par	lancée par
échec	`ou-sinon5`	`reprendre5`
succès après filtrage d'un élément	`et-puis5`	`reussir5`
succès après filtrage d'un début de liste	`et-puis5`	`continuer5`

Des filtres comme `*not` jouent même de façon quasi-perverse avec les continuations, aussi, bien des erreurs sont à craindre. Les fermetures encodant les continuations seront donc elle-mêmes encapsulées dans des objets dont les "invoqueurs" spécialisés (`reprendre5`, `reussir5` et `continuer5`) testeront le type avant toute action risquant d'être infortunée. Les continuations auront donc un champ contenant une fermeture :

```
(defclass continuation5 :slots (fn))
```

Cette réification des continuations permet une mise au point plus parlante. Un appel erroné comme `(reprendre5 (make-et-puis5 ...))` provoquera une erreur "méthode indéfinie" et l'on saura alors quels était le type réel de la continuation reçue et celui attendu.

Quelles informations doivent contenir ces continuations ?

A — Une continuation d'échec ne sert qu'à mémoriser une voie non explorée; elle ne nécessite rien de l'état que l'on désire abandonner. En revanche, elle mémorise tout ce qui est nécessaire pour reprendre l'exploration à partir d'un précédent point de choix. Elle sera donc codée comme une fermeture sans variable.

```
(defclass alt5 :super (continuation5)                    alt5
  :methods (reprendre5) )

(defmacro ou-sinon5 (filtrage)                           ou-sinon5
   `(make-alt5 :fn #'(lambda () ,filtrage)) )

(defgeneric reprendre5 (alt))                            reprendre5

(defmethod reprendre5 ((alt alt5))
   (funcall (alt5-fn alt)) )
```

B — À l'opposé, les continuations de succès, si elles mémorisent les calculs qui restent à effectuer, ont besoin d'informations mises à jour pour poursuivre leur travail. Il s'agit notamment de l'environnement et des points de choix ouverts. Après le filtrage d'un objet par un filtre d'élément, la continuation de succès sera donc codée comme une fonction à deux arguments `env` et `alt`.

```
(defclass succ5element :super (continuation5)            succ5element
  :methods (reussir5) )

(defgeneric reussir5 (env succ alt)                      reussir5
  :argument-precedence-order (succ) )

(defmethod reussir5 (env (succ succ5element) alt)
   (funcall (succ5element-fn succ) env alt) )
```

L'architecture de `filtrer5-segment`, lorsque la liste de filtres débute par un filtre de segment, requiert que soit délégué à ce filtre de segment le soin d'accepter un segment initial de l'expression puis de relancer le filtrage en indiquant l'expression qui reste à filtrer (c'est-à-dire le segment final non filtré de l'expression), l'environnement et les points de choix résultants.

Le filtrage de l'expression `(dans le mitan du lit)` par `(??- mitan ??fin)` se déroule donc ainsi :

1. comme le filtre est une liste, `filtrer5-element` délègue à `filtrer5-segment` son traitement,
2. `filtrer5-segment` délègue alors à `*any` le soin de filtrer un segment initial de `(dans le mitan du lit)`,
3. `*any` choisit de ne rien prendre et donc de relancer le filtrage de l'expression originale `(dans le mitan du lit)` avec `(mitan ??fin)`. Cette expérience s'effectue avec `ou-sinon5` qui indique qu'en cas d'échec `*any` tentera autre chose et notamment prendra au moins le terme `dans` de la liste `(dans le mitan du lit)`.

C — La continuation de succès d'un filtre de segment implique donc de savoir quel segment final de l'expression le reste du filtre s'évertuera à traiter. Une telle continuation sera donc codée comme une fonction à trois variables `e`, `env` et `alt`.

```
(defclass succ5segment  :super (continuation5)           succ5segment
  :methods (continuer5) )
```

continuer5

```
(defgeneric continuer5 (e env succ alt)
  :argument-precedence-order (succ) )

(defmethod continuer5 (e env (succ succ5segment) alt)
   (funcall (succ5segment-fn succ) e env alt) )
```

D — La macro `et-puis5` bâtira ces deux types de continuations de succès : il suffira d'indiquer les variables désirées **`(env alt)`** ou **`(e env alt)`**. La discrimination se fera sur la longueur de la liste des variables.

et-puis5

```
(defmacro et-puis5 (variables filtrage)
   (let ((fn `#'(lambda ,variables ,filtrage)))
     (case (length variables)
       (2 `(make-succ5element :fn ,fn))
       (3 `(make-succ5segment :fn ,fn))
       (t (error
           "ET-PUIS ne peut creer une telle chose : ~A"
           `(et-puis5 ,variables ,filtrage) )) ) ) )
```

Donnons de suite quelques exemples d'emploi de ces divers types de continuations.

Dans `filtrer5-segment` se trouvait

```
... (et-puis5 (env alt)
           (filtrer5-segment (cdr e) (cdr f)
                             env succ alt ) ) ...
```

qui signifie qu'il y aura à effectuer, avec les nouvelles valeurs de `env` et `alt`, le filtrage du reste de l'expression `e` par le reste des filtres `f`. Cette forme sera expansée en

```
... (make-succ5element
          :fn #'(lambda (env alt)
                  (filtrer5-segment (cdr e) (cdr f)
                                    env succ alt ) )) ...
```

où la saisie des nouvelles valeurs de `alt` et `env` est clairement indiquée. Toutes les autres variables (`e`, `f` et `succ`) auront pour valeur celles qu'elles ont lors de la définition de cette fonction anonyme. Le second type de continuation de succès est illustré, toujours dans `filtrer5-segment`, par

```
... (et-puis5 (e env alt)
            (filtrer5-segment e (cdr f) env succ alt) ) ...
```

Cette dernière continuation sera relancée par `continuer5` et correspond à

```
... (make-succ5segment
          :fn #'(lambda (e env alt)
                 (filtrer5-segment (cdr e) (cdr f)
                                   env succ alt ) ) )...
```

D'une manière générale, les continuations bâties par `et-puis5` **`(env alt)`** sont relancées par `reussir5` dans le corps des filtres d'éléments, celles que construit `et-puis5` **`(e env alt)`** sont exploitées par `continuer5` dans le corps des filtres de segments.

Les continuations sont conceptuellement complexes à pratiquer. Il existe toutefois un moyen assez simple de les lire. L'expression (issue de la définition de **`filtre5-segment`**) :

```
...(filtrer5-element (car e) (car f) env
      (et-puis5 (env1 alt1)
         (filtrer5-segment (cdr e) (cdr f) env1 succ alt1) )
      alt ) ...
```

peut se subvocaliser ainsi : *filtrer la première des expressions par le premier des filtres puis donner l'environnement résultant* **`env1`** *et les points de choix ouverts* **`alt1`**, *au filtrage du reste de l'expression par le reste des filtres.* **`env1`** et **`alt1`** représentent le *portail* par lequel transitent les informations entre le filtrage des **`car`** et celui des **`cdr`**.

4.5.2 Le résultat

Le moteur à filtrer devant procurer la possibilité d'obtenir toutes les solutions d'un filtrage, il est nécessaire que les résultats soient composés et de l'environnement de réussite et de la continuation d'échec qui permet de relancer le moteur et ainsi d'obtenir les autres solutions s'il y en a. Comme d'habitude, un filtrage erroné ramènera `NIL`.

Le résultat Vrai sera donc une structure[3] à deux champs `env` et `alt` exploitée par `resultat5-env` et `resultat5-alt`. Le résultat est bâti par la continuation initiale de succès qui apparaît dans la fonction d'interface `filtrer5`.

```
(defstruct resultat5  env alt)                                    resultat5
```

Pour mettre en œuvre commodément le filtrage, `filtrer5` sera ainsi définie :

;;;L'interface : (FILTRER5 expression filtre [env]) ou;;;(FILTRER5 précédent-résultat-de-filtrer5)

```
(defun filtrer5 (arg1 &optional                                   filtrer5
                      (arg2 'ignore si-arg2)
                      (env (initial5-env)) )
   (if si-arg2
     (filtrer5-element arg1 arg2
       env
       (et-puis5 (env alt)
          (make-resultat5 :env env :alt alt) )
       (ou-sinon5 nil) )
     (reprendre5 (resultat5-alt arg1)) ) )
```

Selon son nombre d'arguments, `filtrer5` lancera un filtrage, si on lui fournit une expression et un filtre, ou relancera un filtrage, si on ne lui fournit qu'un précédent résultat de filtrage. Ainsi :

```
(filtrer5 '(q u e j a i m e a f a i r e a p p r e n d r e
            u n n o m b r e u t i l e a u x s a g e s )
           '(??- ?char ??- ?char ??- ?char ??-) )
⇒ #S(resultat5 env ((char u)) alt #<closure17443Q>)
(filtrer5 *) ;* a pour valeur le précédent résultat
⇒ #S(resultat5 env ((char e)) alt #<closure17765Q>)
```

4.5.3 L'environnement

L'environnement associe aux variables de filtrage les valeurs qu'elles acquièrent au fil du processus. Deux types de variables existent : les variables d'élément ou les variables de segment. Afin d'éviter l'important coût de mémorisation des variables de segment, celles-ci seront représentées par un triplet *(nom-de-la-variable, taille, liste)* dont la signification sera que la variable sera liée aux *taille* premiers termes de *liste*. Le coût de mémorisation d'une telle variable est donc constant et en particulier indépendant de la taille du segment.

On nommera *liaison* cette association entre un nom de variable de filtrage et sa valeur. Les liaisons sont donc de deux types, accordés à la nature des variables associées. Comme il est possible de convertir une valeur de filtrage d'un type en l'autre comme l'illustrent, par exemple, les filtres `(?x ??x)` ou `(*cons (??x) ?x)` , les deux types de liaisons répondront aux mêmes méthodes. On utilisera pour ce faire des fonctions génériques. Voici donc les définitions des liaisons

```
(defclass liaison5                                                liaison5
  :slots (nom) )

(defclass liaison5element :super (liaison5)                       liaison5element
  :slots (valeur)
  :methods (liaison5-valeur liaison5-taille
            liaison5-reference ) )

(defclass liaison5segment :super (liaison5)                       liaison5segment
  :slots (taille reference)
```

[3] C'est une structure et non une classe car cela suffisait ici.

```
    :methods (liaison5-valeur liaison5-taille
              liaison5-reference ) )
```

Les fonctions génériques communes à ces deux types de liaisons permettent :

A — d'extraire la valeur associée à une liaison (et donc de la construire si la liaison était de type segment) :

liaison5-valeur

```
(defgeneric liaison5-valeur (lien))

(defmethod liaison5-valeur ((lien liaison5element))
   (liaison5element-valeur lien) )

(defmethod liaison5-valeur ((lien liaison5segment))
   (subseq (liaison5segment-reference lien)
           0 (liaison5segment-taille lien) ) )
```

B — d'extraire la taille associée à une liaison (et donc de la calculer si la liaison était de type élément) :

liaison5-taille

```
(defgeneric liaison5-taille (lien))

(defmethod liaison5-taille ((lien liaison5segment))
   (liaison5segment-taille lien) )

(defmethod liaison5-taille ((lien liaison5element))
   (longueur (liaison5element-valeur lien)) )
```

longueur

```
;;; Une longueur qui ne tient pas compte des ultimes cdr.
(defun longueur (l)
   (if (consp l) (1+ (longueur (cdr l))) 0) )
```

C — d'extraire un pointeur sur la liste dont le début est la valeur associée à une liaison (surtout pour simuler une liaison de type segment à partir d'une liaison de type élément) :

liaison5-reference

```
(defgeneric liaison5-reference (lien))
```

Les environnements sont eux-mêmes encapsulés par `initial5-env` qui les crée, `ajouter5element` et `ajouter5segment` qui les enrichissent de liaisons de type élément ou segment.

initial5-env

```
(defun initial5-env () nil)
```

ajouter5element

```
(defun ajouter5element (env nom valeur)
   (cons (make-liaison5element :nom nom
                               :valeur valeur ) env) )
```

ajouter5segment

```
(defun ajouter5segment (env nom taille reference)
   (cons (make-liaison5segment :nom nom
                               :taille taille
                               :reference reference )
         env ) )
```

Les liaisons sont extraites de l'environnement grâce à `env5-liaison`.

env5-liaison

```
(defun env5-liaison (nom env)
    (find-if env :key #'liaison5-nom
                 :test #'eq ) )
```

4.5.4 Le définisseur de filtres

Les définitions des filtres des précédentes versions étaient elles-mêmes encapsulées dans une forme `de-filtre` appropriée. Cette enveloppe avait permis qu'au fil des variantes leur aspect extérieur ne change pas trop. Nous poursuivrons cette approche en augmentant encore les qualités de ce définisseur.

- L'invocation des filtres s'effectue par `lancer5`. Le définisseur a donc à charge (et à cœur) de conditionner, comme il le faut, le corps des filtres.
- Les arguments des filtres seront nommés.

  Pour les filtres d'éléments qui possèdent des arguments, il est préférable de les nommer plutôt que de les utiliser sous les noms occultes de `(cadr f)` , `(caddr f)` , `(cdr f)` , etc.

Un filtre d'élément sera défini comme suit

```
(de-filtre5 (nom variables...) element
      définition )
```

Lorsqu'un tel filtre sera lancé, les paramètres qui suivent son nom seront liés aux *variables*. Par exemple, le filtre d'élément `*quote` est ainsi défini :

```
(de-filtre5 (*quote expression) element
    (if (equal expression e)
        (reussir5 env succ alt)
        (reprendre5 alt) ) )
```

*quote

Lorsque le filtre `(*quote (foo bar))` est invoqué, le corps de `*quote` sera évalué dans un environnement où la variable nommée `expression` vaudra `(foo bar)`. La liste des *variables* obéit à la même syntaxe que les variables des fonctions de Scheme, l'ultime `cdr` d'une liste pointée, s'il n'est pas `nil`, récupère donc tous les paramètres correspondants. Ainsi en est-il pour le filtre `*list` :

```
(de-filtre5 (*list . filtres) element
    (filtrer5-segment e filtres env
                    ... ) )
```

*list

Lorsque le filtre `(*list foo (*quote bar))` est invoqué, le corps de `*list` sera évalué dans l'environnement où la variable nommée `filtres` vaudra `(foo (*quote bar))`.

Les filtres de segments sont plus complexes car ils peuvent prendre des arguments, contenus dans `(cdar f)`, mais aussi se saisir de leur contexte droit c'est-à-dire des filtres qui les suivent : `(cdr f)`. Un filtre de segment sera défini comme suit :

```
(de-filtre5 ((nom variables...) autres-filtres) segment
      définition )
```

Les *variables* et *autres-filtres* possèderont la même syntaxe qu'une liste de variables d'une fonction Scheme. Donnons un exemple,

```
(de-filtre5 ((*terms nom) . reste) segment
   (let ((v (env5-liaison nom env)))
       ... ) )
```

Lorsque le filtre `((*terms foo) (*any) (*terms foo))` sera invoqué, le corps de `*terms` sera évalué avec `nom` valant `foo` et `reste` valant `((*any) (*terms foo))`.

Le corps des filtres peut utiliser toutes les variables mentionnées dans sa forme d'emploi et, si c'est un filtre-segment, son contexte droit. Le corps des filtres peut aussi utiliser les quatre variables prédéfinies qui définissent l'état du moteur à filtrer : `e`, `env`, `succ` et `alt`.

Il n'est par contre pas possible d'utiliser la variable `f` dont la valeur était le filtre ou la liste de filtres courante dans les précédentes versions. Le filtre `*times` qui l'utilisait devra donc être reprogrammé. Les raisons de cette interdiction sont d'une part son inutilité puisque arguments et contexte droit du filtre sont directement nommables et accessibles, d'autre part de rendre plus difficile la programmation récursive par les filtres. Elle prépare les techniques de compilation qui seront vues au prochain chapitre et qui ne supportent guère la synthèse dynamique de filtres.

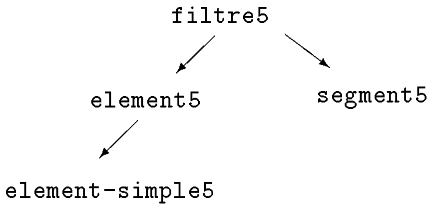


Figure 4.1: Hiérarchie des filtres

Il sera du rôle de `lancer5` de lier les variables aux arguments, ce qui nécessite une certaine gymnastique avec les `&rest` et `apply`. Ce conditionnement est entièrement réalisé, dès la définition du filtre, par `de-filtre5`.

Enfin à côté des filtres de segment et des filtres d'élément, a été ajouté un raffinement de ces derniers : les filtres d'*élément simple*. Les filtres d'élément simple répondent au schéma :

si *condition*
    alors *succès*
    sinon *échec* **finsi**

Il suffira donc d'écrire la seule portion de code intéressante et d'indiquer le type `element-simple` pour que la définition correcte soit engendrée. En d'autres termes

```
(de-filtre5 (nom variables...) element-simple
  condition )
```

est équivalent à

```
(de-filtre5 (nom variables...) element
   (if condition
       (filtrer5-reussir env succ alt)
       (filtrer5-reprendre alt) ) )
```

Loin de n'être qu'une commode abréviation, l'on verra que tous les filtres simples peuvent prêter main-forte à un *oracle* améliorant l'efficacité du tout.

Les filtres de type `element-simple` sont un raffinement des filtres `element`. Ceci nous incite à adopter encore une fois une architecture d'objet avec, ce coup-ci, un héritage entre filtres présenté en figure 4.1.

Les méthodes seront encore des fonctions génériques pourvues de méthodes par défaut définies sur `filtre5`. Tous les filtres auront la même structure : un nom et un comportement.

```
(defclass filtre5                                    filtre5
  :slots (nom comportement) )

(defclass element5        :super (filtre5)           element5
  :methods (lancer5) )

(defclass element-simple5 :super (element5))         element-simple5

(defclass segment5        :super (filtre5)           segment5
  :methods (lancer5) )
```

Un prédicat particulier `filtre5type-p` détermine si une expression correspond à un filtre d'un certain type.

```
(defun filtre5type-p (exp type)                      filtrer5type-p
   (and (consp exp)
        (symbolp (car exp))
        (obtenir5filtre (car exp))
        (typep (obtenir5filtre (car exp)) type)) )
```

Le lanceur est immédiat pour peu que l'on sache convertir un nom de filtre en l'objet qui code ce filtre. Cette conversion reposera sur une table de hachage globale nommée `*filtres*`.

*filtres

```
(defvar *filtres*)
(setq *filtres* (make-hash-table :size 30))

;;; convertir un nom en un filtre,
;;; répond NIL si ce n'est pas un filtre.
```

obtenir5filtre

```
(defun obtenir5filtre (nom)
   (gethash nom *filtres*) )
```

lancer5

```
(defgeneric lancer5 (filtre e f env succ alt))

(defmethod lancer5 ((filtre element5) e f env succ alt)
   (apply (filtre5-comportement filtre)
          e env succ alt (cdr f) ) )

(defmethod lancer5 ((filtre segment5) e f env succ alt)
   (apply (filtre5-comportement filtre)
          e env succ alt (cdar f) (cdr f) ) )
```

Et voici donc, au terme de tous ces préparatifs, le fameux et tant attendu définisseur `de-filtre5`. Attention, il est complexe !

de-filtre5

```
(defmacro de-filtre5 (appel type . def)
   (let* ((args (gensym))
          (nom (if (consp (car appel))
                   (caar appel) (car appel) ))
          ;(de-filtre5 (filtre . arg1) type. body)
          (arg1 (analyse5-vars
                   (if (consp (car appel))
                       (cdar appel) (cdr appel) ) ))
          ;(de-filtre5 ((filtre . arg1) . arg2) type . body)
          (arg2 (if (consp (car appel))
                    (analyse5-vars (cdr appel)) )) )
      `(setf (gethash ',nom *filtres*)
          (let ((filtre
                 ,(creer5filtre type arg1 arg2 args def) ))
             (setf (filtre5-nom filtre) ',nom)
             filtre ) ) ) )

;;; change (a b . c) en (a b &rest c).
```

analyse5-vars

```
(defun analyse5-vars (vars)
  (if (consp vars)
      (cons (car vars) (analyse5-vars (cdr vars)))
      (if (null vars) vars (list '&rest vars)) ) )
```

creer5filtre

```
(defun creer5filtre (type arg1 arg2 args def)
   (case type
     (element-simple
      `(make-element-simple5 :comportement
            #'(lambda (e env succ alt . ,arg1)
                 (if (progn . ,def)
                     (reussir5 env succ alt)
                     (reprendre5 alt) ) ) ) )
     (element
      `(make-element5 :comportement
         #'(lambda (e env succ alt . ,arg1) . ,def) ) )
     (segment
```

```
      (setq arg2 (if (null arg2)
                     (analyse5-vars (gensym)) arg2) )
      `(make-segment5 :comportement
         #'(lambda (e env succ alt ,args . ,arg2)
             (apply #'(lambda ,arg1 . ,def) ,args ) ) ) ) )
     (t (error "DE-FILTRE type inconnu ~A" type)) ) )
```

Le lecteur précautionneux étudiera avec soin le conditionnement des corps de filtres en comportements. Cette transformation use d'`apply` et de `&rest` de façon non triviale.

Cette longue préparation s'achève. Les mécanismes mis en place vont permettre de définir les filtres de manière naturelle et pourront être totalement oubliés, encapsulés qu'ils sont ! Les primitives qui seront les plus employées sont :

`filtrer5-element`	`filtrer5-segment`		*les filtreurs*
`et-puis5`	`reussir5`	`continuer5`	*les continuateurs de succès*
`ou-sinon5`	`reprendre5`		*les relanceurs après échec*

4.6 Les filtres

Il ne reste plus qu'à passer en revue les filtres. Ils ont tellement été rabâchés dans les chapitres précédents et il y a eu tellement de préparatifs que leur énoncé va être simple et bref, car leur définition est quasiment triviale. Les grandes conventions de dénomination des variables sont les suivantes :

`e`	l'expression à filtrer
`env`	l'environnement courant
`succ`	la continuation de succès
`alt`	la continuation d'échec

Dans les fonctions où elles figurent, elles seront toujours énoncées dans cet ordre, ce qui simplifie l'effort d'écriture des programmes mais a imposé le recours à l'option `:argument-precedence-order` dans `reussir5`. Les variables `e`, `env`, `succ` et `alt` doivent être considérées comme de véritables mots-clés dont la portée est le corps des définitions de filtres.

4.6.1 Les filtres de toujours

À l'appel, arrivent en premier les filtres de toujours :

`*sexp`

```
(de-filtre5 (*sexp) element-simple
    t )
```

`*quote`

```
(de-filtre5 (*quote expression) element-simple
    (equal expression e) )
```

`*list`

```
(de-filtre5 (*list . filtres) element
    (filtrer5-segment e filtres env
                      (et-puis5 (e env alt)
                             (if (null e)
                                 (reussir5 env succ alt)
                                 (reprendre5 alt) ) )
                      alt ) )
```

`*cons`

```
(de-filtre5 (*cons filtre1 filtre2) element
   (if (consp e)
     (filtrer5-element (car e) filtre1 env
```

```
        (et-puis5 (env alt)
            (filtrer5-element (cdr e) filtre2 env succ alt) )
        alt )
     (reprendre5 alt) ) )
```

Le filtre suivant transcende le filtre `*even`. Il prend en argument un prédicat et ne reconnaît qu'une expression qui satisfait ce prédicat.

*check

```
(de-filtre5 (*check predicat) element-simple
   (funcall predicat e) )
```

À la place donc de `(*even)`, l'on pourra écrire `(and (*check numberp) (*check even))`. Le filtre `*even` était, naturellement, un filtre de catégorie `element-simple`, rien ne garantit en revanche que le prédicat argument de `*check` soit une vraie fonction mathématique qui, à égalité d'arguments, calcule un même résultat. `(*check (lambda (x) (< (random 10) 5)))` n'est certainement pas un filtre tout à fait normal ! Ainsi qualifier `*check` de filtre d'élément simple est peut-être présomptueux.

4.6.2 Les filtres booléens

Les filtres booléens restent fidèles à eux-mêmes.

*or

```
(de-filtre5 (*or . filtres) element
   (labels ((filtrer5*or (e fs env succ alt)
              (if (consp fs)
                (filtrer5-element e (car fs) env succ
                  (if (consp (cdr fs))
                    (ou-sinon5 (filtrer5*or e (cdr fs)
                               env succ alt ))
                    ;;Le cas particulier de (*OR filtre)
                    alt ) )
                ;;(*OR) est l'échec
                (reprendre5 alt) ) ))
     (filtrer5*or e filtres env succ alt) ) )
```

*and

```
(de-filtre5 (*and . filtres) element
   (labels ((filtrer5*and (e fs env succ alt)
              (if (consp fs)
                (filtrer5-element e (car fs) env
                  (if (consp (cdr fs))
                    (et-puis5 (env alt)
                       (filtrer5*and e (cdr fs)
                                    env succ alt ) )
                       ;;Le cas de (*AND filtre)
                       succ )
                    alt )
                  ;;(*AND) est ?-
                  (reussir5 env succ alt) ) ))
     (filtrer5*and e filtres env succ alt) ) )
```

*not

```
(de-filtre5 (*not filtre) element
    (filtrer5-element e filtre env
         (et-puis5 (env1 alt1) (reprendre5 alt))
         (ou-sinon5 (reussir5 env succ alt)) ) )
```

On remarquera cependant, par rapport aux versions précédentes, quelques améliorations portant sur leur caractère "récursif terminal". La fonction locale `filtrer5*and` (resp. `filtrer5*or`) ne construit une continuation grâce à `et-puis5` (resp. `ou-sinon5`) que si elle n'est pas triviale. En effet,

```
succ ≡ (et-puis5 (env alt) (filtrer5*and e () env succ alt))
alt  ≡ (ou-sinon5 (filtrer5*or e () env succ alt))
```

La négation perd, bien sûr, l'environnement de succès puisque c'est celui dont on ne veut pas. On méditera aussi sur les limites naturelles de `*not` telles qu'elles apparaissent dans l'exercice suivant.

Exercice 4.10 : Comparer `(?x (*not ?x))` et `((*not ?x) ?x)` ?

4.6.3 Les variables de filtre

La raison d'être de `filtrer5` est d'introduire les variables de segment. Nous présenterons à la fois les filtres d'éléments et les filtres de segments nommés car ils entretiennent certains rapports. Voici le code bien connu de `*term` :

`*term`

```
(de-filtre5 (*term nom) element
   (let ((v (env5-liaison nom env)))
        (if v
            ;; On a déjà rencontré la variable
            (if (equal (liaison5-valeur v) e)
                (reussir5 env succ alt)
                (reprendre5 alt) )
            ;; Enrichir l'environnement
            (reussir5 (ajouter5element env nom e)
                      succ
                      alt ) ) ) )
```

En réminiscence des variables de type élément, les variables de type segment seront nommées : `(*terms` *nom*) et abrégées en `??`*nom*.

`*terms`

```
(de-filtre5 ((*terms nom) . reste) segment
   (let ((v (env5-liaison nom env)))
     (if v (verifier5 (liaison5-taille v)
                      e (liaison5-reference v)
                      reste env succ alt )
       (labels ((filtrer5*terms (n nom e reste env succ alt)
                  (if n
                    (continuer5 (nthcdr n e)
                      (ajouter5segment env nom n e)
                      succ
                      (let ((n (rectifier5 n e reste env)))
                        (if n (ou-sinon5
                                (filtrer5*terms n nom
                                  e reste env succ alt ))
                              alt ) ) )
                    (reprendre5 alt) ) ))
         (filtrer5*terms (deviner5 e reste env) nom
                         e reste env succ alt ) ) ) ) )
```

Dans le cas où la variable de filtrage est valuée, la fonction `verifier5` vérifie que l'expression débute par la valeur de la variable de filtrage et relance, en cas de succès, le processus. `verifier5` est tout simplement

`verifier5`

```
;;; Vérifier que ES1 débute par les N premiers termes de ES2
;;; (assume (≥ (longueur es2) n))
(defun verifier5 (n es1 es2 reste env succ alt)
   (if (> n 0)
     (if (and (consp es1) (equal (car es1) (car es2)))
       (verifier5 (1- n) (cdr es1) (cdr es2)
                  reste env succ alt )
       (reprendre5 alt) )
     (continuer5 es1 env succ alt) ) )
```

La technique employée, dans le cas non trivial où le filtre n'est pas valué, est la suivante : `deviner5` fait l'hypothèse que le segment cherché a une certaine taille. On vérifie alors que les filtres qui suivent filtrent bien le reste de l'expression. Si l'hypothèse est erronée, on la rectifie grâce à `rectifier5`, et l'on recommence. Pour finir par deviner juste, il suffit d'énumérer tous les entiers depuis zéro jusqu'à la taille entière de l'expression à filtrer ! Lorsque toutes les hypothèses ont été successivement formulées, l'échec est alors déclaré et propagé.

```
;;; deviner la taille du segment
(defun deviner5 (e f env)                                    deviner5
    0 )

;;; rectifier le choix de la taille du segment
;;; ou NIL si pas d'autre choix possible
(defun rectifier5 (n e f env)                                rectifier5
    (if (< n (longueur e)) (1+ n)) )
```

Soit le filtre `(??x ??x)` et l'expression `(ma mere jezabel)`. Le premier essai que tente `??x` est de ne rien prendre, de lier `x` à `()` et de relancer le reste du filtre `(??x)` confronté au reste de l'expression `(ma mere jezabel)`. L'échec de cette tentative interpelle `*terms` au niveau de son vécu, en le pressant d'effectuer un nouveau choix. `*terms` décide alors de lier `x` à `(ma)` et de soumettre au reste des filtres `(??x)` le segment terminal de liste dont il n'a pas voulu : `(mere jezabel)`. Le processus continuera jusqu'à l'échec du filtrage par `??x` de `()` qui forcera `*terms` à invoquer sa continuation d'échec et de reconnaître par là-même sa totale incapacité.

> Il est évident que si `deviner5` voyait que `(??x ??x)` ne peut filtrer que des listes de nombre pair de termes, s'avisait que `(ma mere jezabel)` a une longueur de trois, et, en connaissance de cause, retournait `nil` de suite pour signe de sa clairvoyance divinatoire, alors `*term` jugerait bien sûr inutile de faire toutes les précédentes et malheureuses tentatives. Disposer d'un bon oracle serait un bonheur !

À essayer : *Améliorer* `deviner5`

Il ne reste plus qu'à étudier les deux filtres qui firent capoter la précédente version du moteur à filtrer : `*any` et `*times`. L'attrape-tout plural `??-` a une programmation similaire à celle de `*terms` sauf que `??-` n'effectue aucune liaison. Comme `??-` use du même couple `deviner5` – `rectifier5`, toute amélioration de l'oracle sera bénéfique pour `*any`.

```
(de-filtre5 ((*any) . reste)  segment                        *any
   (labels
     ((filtrer5*any (n e reste env succ alt)
         (if n
           (continuer5 (nthcdr n e) env succ
             (let ((n (rectifier5 n e reste env)))
               (if n (ou-sinon5
                       (filtrer5*any n
                          e reste env succ alt ))
                     alt ) ) )
           (reprendre5 alt) ) ))
     (filtrer5*any (deviner5 e reste env)
                   e reste env succ alt ) ) )
```

Le répéteur de filtres `*times` s'inspire, quant à lui, de l'équation

`(*times` *filtre*`)` ≡ `(*progn)`
≡ `(*progn` *filtre* `(*times` *filtre*`))`

et en constitue une paraphrase concise.

```
(de-filtre5 (*times . filtres) segment                       *times
   (labels ((filtrer5*times (filtres e env succ alt)
              (continuer5 e env succ
                (ou-sinon5 (filtrer5-segment
                             e filtres env
                             (et-puis5 (e env alt)
                                 (filtrer5*times filtres
```

```
                                            e env succ alt ) )
                              alt )) ) ) ))
      (filtrer5*times filtres e env succ alt) ) )
```

On peut voir sur cet exemple l'intérêt des continuations de succès de type `succ5segment`. Le filtre `*times` relance le reste du filtrage en lui ordonnant que la partie de l'expression non retenue lui soit de nouveau soumise : il peut ainsi répéter son argument autant qu'il lui plaît.

Exercice 4.11 : Imaginer un filtre fini acceptant l'expression `(liste toute bete)` un nombre infini de fois.

4.6.4 Les macro-filtres

On l'a vu auparavant à l'occasion de `*maybe` (cf. page 30), il est agréable de définir certains filtres par des schémas de filtrage. À l'instar des macros de Lisp, c'est un artifice syntaxique séduisant, limitant la complexité et procurant de non négligeables capacités d'extension. Nous allons donc introduire le concept de *macro-filtres* par modification du couple de base `filtrer5-element` et `filtrer5-segment` pour les reconnaître et les traiter dans la foulée. Cette manière est tout à fait réminiscente des usages naturels des évaluateurs Lisp. L'amélioration est très légère et tient en trois lignes.

`filtrer5-element`

```
(defun filtrer5-element (e f env succ alt)
   (if (filtre5type-p f 'macro-element5)          ;nouveau
       (filtrer5-element                          ;nouveau
         e (expanser5 (obtenir5filtre (car f)) f) ;nouveau
         env succ alt )                           ;nouveau
       (if (filtre5type-p f 'element5)
         (lancer5 (obtenir5filtre (car f))
                  e f env succ alt )
         ; Traitement par défaut :
         (if (atom f) (if (equal e f)
                          (reussir5 env succ alt)
                          (reprendre5 alt) )
             (filtrer5-segment e f env
               (et-puis5 (e env alt)
                        (if (null e)
                            (reussir5 env succ alt)
                            (reprendre5 alt) ))
               alt ) ) ) ) )
```

`filtrer5-segment`

```
(defun filtrer5-segment (e f env succ alt)
   (if (consp f)
     (if (filtre5type-p (car f) 'macro-segment5)   ;nouveau
       (filtrer5-segment                           ;nouveau
         e (expanser5 (obtenir5filtre (caar f)) f) ;nouveau
         env succ alt )                            ;nouveau
       (if (filtre5type-p (car f) 'segment5)
         (lancer5 (obtenir5filtre (caar f))
              e f env
              (et-puis5 (e env alt)
                  (filtrer5-segment e (cdr f)
                                   env succ alt ) )
              alt )
         ; Traitement par défaut :
         (if (consp e)
             (filtrer5-element (car e) (car f) env
                 (et-puis5 (env alt)
                   (filtrer5-segment (cdr e) (cdr f)
                     env succ alt ) )
```

```
            alt )
          (reprendre5 alt) ) ) )
      ;On a achevé la liste de filtres
      (continuer5 e env succ alt) ) )
```

Comme précédemment la reconnaissance et l'expansion des macro-filtres sont encapsulées.

```
(defclass macro5          :super (filtre5))
```

macro5

```
(defclass macro-element5  :super (macro5)
  :methods (expanser5) )
```

macro-element5

```
(defclass macro-segment5  :super (macro5)
  :methods (expanser5) )
```

macro-segment5

Le reconnaisseur existe déjà, c'est `filtre5type-p` ! Les expanseurs sont des sortes de lanceurs (cf. `lancer5`) mais limités à la seule donnée du filtre. L'expansion ne dépend de rien d'autre et ne peut aucunement être paramétrée par `e`, `env`, `succ` ou `alt`.

```
(defgeneric expanser5 (filtre f))

(defmethod expanser5 ((filtre macro-element5) f)
   (apply (filtre5-comportement filtre) (cdr f)) )

(defmethod expanser5 ((filtre macro-segment5) f)
   (apply (filtre5-comportement filtre) (cdar f) (cdr f)) )
```

expanser5

Le définisseur générique de filtres se complique donc un peu plus pour reconnaître et traiter ces macro-filtres. Heureusement la syntaxe de `de-filtre5` absorbe cela sans problème aucun. Plus exactement l'extension ne concernera que `creer5filtre` qui devient

```
(defun creer5filtre (type arg1 arg2 args def)
   (case type
     (macro-element                               ;nouveau
      `(make-macro-element5 :comportement         ;nouveau
         #'(lambda ,arg1 . ,def) ) )              ;nouveau
     (macro-segment                               ;nouveau
      (setq arg2                                  ;nouveau
            (if (null arg2)                       ;nouveau
                (analyse5-vars (gensym)) arg2 ) ) ;nouveau
      `(make-macro-segment5 :comportement         ;nouveau
         #'(lambda (,args . ,arg2)                ;nouveau
             (apply #'(lambda ,arg1 . ,def)       ;nouveau
                    ,args ) ) ) )                 ;nouveau
     (element-simple
      `(make-element-simple5 :comportement
         #'(lambda (e env succ alt . ,arg1)
             (if (progn . ,def)
               (reussir5 env succ alt)
               (reprendre5 alt) ) ) ) ) )
     (element
      `(make-element5 :comportement
         #'(lambda (e env succ alt . ,arg1)
                . ,def ) ) ) )
     (segment
      (setq arg2
           (if (null arg2)
             (analyse5-vars (gensym)) arg2 ) )
      `(make-segment5 :comportement
         #'(lambda (e env succ alt ,args . ,arg2)
             (apply #'(lambda ,arg1 . ,def) ,args ) ) ) )
```

creer5filtre

```
    (t (error "DE-FILTRE type inconnu ~A" type)) ) )
```

Les comportements des macro-filtres sont conformes à ce qu'attendent les expanseurs et usent d'`apply` et de `&rest`. On peut aussi voir que les macro-filtres sont codés comme des objets ayant un comportement, ils s'intègrent donc facilement dans la hiérarchie des filtres comme le montre la figure 4.2.

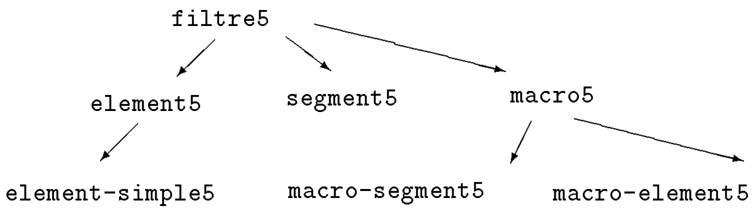


Figure 4.2: La hiérarchie complète des filtres

4.6.5 Un arroi de macro-filtres

Il ne reste plus qu'à définir quelques macro-filtres. Tout comme les filtres élémentaires, deux catégories de macro-filtres existent suivant qu'ils sont de type élément ou segment. En guise d'exemples, voici les définitions de deux macro-filtres de type élément suivies de deux autres de type segment.

A — La définition de `*mcons` (cf. exercice 3.8) sous forme de filtre d'élément était longue et pénible. Sa spécification sous forme de réécriture en `*cons` la rend plus immédiatement compréhensible.

*mcons

```
(de-filtre5 (*mcons . filtres) macro-element
   (if (consp filtres)
       (if (consp (cdr filtres))
           `(*cons ,(car filtres) (*mcons . ,(cdr filtres)))
           (car filtres) )
       (error "*MCONS sans argument") ) )
```

Ainsi donc le filtre `(*mcons` *filtre*$_1$ *filtre*$_2$ ... *filtre*$_n$`)` se transcrit tout simplement en `(*cons` *filtre*$_1$ `(*cons` *filtre*$_2$ ...`(*cons` *filtre*$_{n-1}$ *filtre*$_n$`)` ...`))`.

B — Le macro-filtre `*listwhere` reconnaît toute liste où se trouvent ses arguments.

*listwhere

```
(de-filtre5 (*listwhere . filtres) macro-element
   `(*and . ,(mapcar #'(lambda (filtre) `(??- ,filtre ??-))
                     filtres )) )
```

Par exemple :

`(*listwhere anne)`
accepte l'expression `(annie ann anna anne anny)`.

`(*listwhere paul damien)`
reconnaît toute liste dont `paul` et `damien` sont des termes et notamment `(lin clet damien clement paul cyprien)` ou `(paul pierre damien)` mais pas `(damien (paul))`.

`((*listwhere ?x) (*listwhere ?x))`
reconnaît toute liste de deux termes ayant au moins un terme en commun.

De fait `(*listwhere` *quelque-chose*`)` est équivalent au filtre `(??-` *quelque-chose* `??-)` dont il n'est qu'une commode abréviation.

C — Les deux macro-filtres suivants ont déjà été vus, ce sont deux macro-filtres pour segment dont la définition est immédiate.

*progn

```
(de-filtre5 ((*progn . filtres) . reste) macro-segment
    (append filtres reste) )
```

*maybe

```
(de-filtre5 (*maybe . filtres) macro-segment
   `(*oorr (*progn . ,filtres) (*progn)) )
```

On notera que le macro-filtre `*progn` se saisit de son contexte droit (`reste`) et le fait figurer dans son expansion.

`((*progn amour ?delice) ??orgue)`
s'expanse en `(amour ?delice ??orgue)`.

Au contraire, `*maybe` est un macro-filtre de segment qui s'expanse individuellement sans requérir connaissance de son contexte droit.

`(*maybe not)`
s'expanse en `(*oorr (*progn not) (*progn))` ce qui, replacé dans le contexte du filtre `(*list maybe (*maybe not))` conduit au filtre équivalent `(*list maybe (*oorr (*progn not) (*progn)))` qui n'accepte que les listes `(maybe)` et `(maybe not)`.

Pour simple qu'elle puisse apparaître, l'introduction des macro-filtres est une adjonction fondamentale. Par un choix réduit mais judicieux de filtres élémentaires, on peut espérer que quantité d'autres se réduiront à n'être que des macro-filtres.

À essayer : *Isoler un jeu réduit de filtres*

D — Un tel filtre élémentaire pourrait être `*tail`. `*tail` est un filtre de segment qui doit apparaître en dernier d'une liste de segments, son argument (qui doit nécessairement être un filtre d'élément) spécifie comment doit s'achever la liste en cours de filtrage. `*tail` permet de revenir de `filtrer5-segment` à `filtrer5-element` à l'inverse de `*list` qui opérait le transfert contraire.

*tail

```
(de-filtre5 ((*tail filtre) . reste) segment
   (if (consp reste)
       (error "*TAIL n'est pas le dernier filtre: " reste)
       (filtrer5-element e filtre env
             (et-puis5 (env alt)
                    (continuer5 nil env succ alt) )
             alt ) ) )
```

`(a (*tail b))`
équivaut à `(a . b)` , c'est-à-dire à `(*cons a b)`

`(hux foo (*tail ?bar))`
équivaut à `(*cons hux (*cons foo ?bar))`

`(??alpha (*tail (foo ??alpha)))`
est équivalent à `(??alpha foo ??alpha)`

S'appuyant sur `*tail`, on peut redéfinir le filtre de segment `*oorr` à l'aide de `*tail` et `*or` en :

*oorr

```
(de-filtre5 ((*oorr . filtres) . reste) macro-segment
   `((*tail (*or ,@(mapcar #'(lambda (f)
                                `(*list ,f . ,reste) )
                           filtres )))) )
```

Nous arrêterons là l'exposé des macro-filtres en remarquant toutefois qu'expanser encore et encore les macro-filtres est un facteur de lenteur qui sera corrigé par la suite.

À essayer : *N'expanser qu'une fois*

4.6.6 Valeurs multiples en résultat

Manipuler les environnements tels qu'ils sont ici codés relève du masochisme pur car il n'est pas possible d'esquiver les fonctions l'encapsulant. Si l'on considère toutefois que la plupart du temps, les filtrages ne sont effectués que pour leur première solution, on peut utiliser la technique des valeurs multiples et modifier la continuation de succès initiale (celle qui bâtit le résultat) en

```
... (et-puis5 (env alt) (values T env alt) ) ...
```

La première valeur représente le résultat booléen du filtrage. Les deux autres sont respectivement l'environnement de succès et la continuation de relance. Il sera du ressort de l'appelant de récupérer ces valeurs ou seulement l'environnement s'il ne désire pas relancer le filtrage. Il sera alors possible d'écrire en toute légalité

```
(if (filtrer5 '((f o o) (b o r))
              '((??- ?x ??-) (??- ?x ??-)) )
    'ok 'ko )
⇒ OK
```

Les renseignements supplémentaires fournis par le filtrage ont été ignorés de l'appelant grâce au mécanisme des valeurs multiples. Si celui-ci désire cependant les récupérer, il emploiera par exemple `multiple-value-bind` et écrira :

```
(multiple-value-bind
          (bool env alt)
          (filtrer5 '((f o g) (b o r))
                    '((??- ?x ??-) (??- ?x ??-)) )
    (filtrer5 alt) )
⇒ NIL ;Il n'y avait qu'une seule solution !
```

4.6.7 Oracle et efficacité

La programmation des filtres de segment `*terms` et `*any` usait de deux sous-fonctions `deviner5` et `rectifier5`. Leur rôle était de délivrer, tour à tour, toutes les longueurs possibles du segment en cours d'analyse. L'idée qui suit (due à Emmanuel Saint-James, cf. aussi [Emanuelson 82]) améliore `deviner5` et lui confère un statut d'oracle avec un bon taux de réussites divinatoires.

La fonction `deviner5` détermine les longueurs les plus plausibles et ne se résoud à énumérer toutes les longueurs que dans les cas les plus défavorables. `rectifier5` est la seconde face de l'oracle qui, après un premier choix malheureux, tente de se racheter en proposant une nouvelle longueur. Pour se prononcer, l'oracle inspecte les filtres qui suivent la séquence en cours, cherche quelques indications lui permettant de cerner la longueur que peut avoir la séquence, enfin analyse l'expression à filtrer et propose une telle longueur.

Supposons avoir à filtrer l'expression : `(une fois la borne franchie il n 'est plus de limite)` [Christophe 04] au travers de `(??sequence borne ??- ?dernier)`. Le premier problème du filtrage est de déterminer la longueur de `??sequence`. Un coup d'œil rapide vous convaincra que, si oracle vous étiez, une bonne réponse serait `3`. Efforçons-nous de faire en sorte que ce soit sa réponse.

Le filtre `??sequence` est suivi de la constante `borne`. Rechercher cette constante dans l'expression à filtrer n'est le fait que d'un seul instant, ce qui conduit à

```
(une fois la borne ...)
(??sequence   borne ...)
```

et la réponse de `deviner5` tombe : `3` !

Le même problème se pose quant à l'analyse de `(??- ?dernier)` associé à `(franchie il n 'est plus de limite)`. L'oracle, à nouveau consulté, inspecte ce qui suit la séquence `??-`, détermine qu'il y aura un terme de longueur unité : `?dernier`, suivi de la fin du filtre. La longueur est donc la taille de l'expression moins le dernier terme à lier, soit 6 :

```
(franchie il n 'est plus de limite)
(           ??-               ?dernier)
```

La réponse fournie par `deviner5` est donc :

```
(- (length '(franchie il n 'est plus de limite))
   (length '(?dernier)) )
```

Si pour une raison ou une autre, ce filtrage doit être recommencé alors `rectifier5` tentera de retrouver une nouvelle longueur plausible pour `??-`. Il n'y a là pas d'espoir et l'oracle retournera `nil`, signifiant par là qu'aucune autre longueur n'est possible. Le filtrage reviendra donc à `??sequence`, l'oracle derechef consulté, recherchera `borne` dans le segment encore inexploré `(franchie il n 'est plus de limite)` et retournera `nil` signifiant un nouvel échec. Notez que c'est un échec pour le filtrage et non pour l'oracle qui n'a délivré que deux réponses :

- oui, il est possible de filtrer `(une fois la borne franchie il n 'est plus de limite)` par `(??sequence borne ??- ?dernier)` si l'on confère une longueur de 3 à `??sequence`,
- non, il n'y pas d'autre solution.

Il existe cependant des cas où l'oracle ne produit rien comme, par exemple, dans le cas suivant : `(??passe ?moi ??rhubarbe)`. Là, toutes les longueurs sont possibles pour `??passe`. Par contre, celle-ci fixée, tous les filtrages postérieurs sont déterminés.

L'efficacité de l'oracle est proportionnel à sa compréhension des filtres. Son but est de se saisir d'un filtre débutant par une séquence dont il doit découvrir la longueur et de chercher à sa droite un point d'ancrage simplement vérifiable. Ce point d'ancrage peut être une constante (symbole ou liste), un filtre d'élément déjà valué ou un objet simple comme un nombre ou un symbole (filtres `(*check atom)` ou `(*check numberp)`) ou encore un filtre de la catégorie `element-simple` qui sont simplement vérifiables.

La programmation de cet oracle est assez simple.

```
;;;devine la taille minimale du segment initial.
;;;retourne une taille, 0 si non devinable ou NIL si filtrage impossible
(defun deviner5 (e reste env)                                    deviner5
   (if (consp fs)
       (if (ancrage5-p (car fs) env)
           (ancrer5 e (car fs) env 0)
           ;;on n'a rien pu deviner !
           0 )
       ;;le contexte est vide, il faut tout accepter !
       (longueur e) ) )

;Teste si F est un filtre simple à reconnaître sachant ENV.
(defun ancrage5-p (f env)                                        ancrage5-p
   (or (atom f)
       (filtre5element-simple-p f)
       (case (car f)
        (*term (env5-liaison (cadr f) env))  ;?x est valué
        (*terms (env5-liaison (cadr f) env)) ;??x est valué
        (t nil) ) ) )

;;;ancrer5 retourne un rang ou NIL si échec
(defun ancrer5 (e f env n)                                       ancrer5
  (if (consp f)
      (if (filtre5element-simple-p f)
          (chercher5element-simple f e n env)
          (case (car f)
            (*term (let ((v (env5-liaison (cadr f) env)))
                     (if v (chercher5-element
                             (liaison5-valeur v) e n ) ) ) ))
```

```
            (*terms (let ((v (env5-liaison (cadr f) env)))
                      (if v (chercher5-segment
                               (liaison5-reference v)
                               (liaison5-taille v)
                               e n ) ) ))
            (t nil) ) )
      (chercher5-element f e n) ) )
```

rectifier5

```
(defun rectifier5 (n e f env)
  (if (< n (longueur e))
      (let ((p (deviner5 (nthcdr (1+ n) e) f env)))
           (if p (+ n p 1)) ) ) )
```

chercher5-element

```
;;;rechercher C dans E et retourner son rang N.
(defun chercher5-element (c e n)
  (if (consp e)
      (if (equal c (car e))
          n
          (chercher5-element c (cdr e) (1+ n)) ) ) )
```

chercher5element-simple

```
;;;rechercher un element simple dans E, filtrable par F
;;;et retourner son rang N.

(defun chercher5element-simple (f e n env)
  (if (consp e)
    (if (lancer5 (obtenir5filtre (car f))
             (car e) f env
             (et-puis5 (env alt) t) (ou-sinon5 nil) )
      n
      (chercher5element-simple f (cdr e) (1+ n) env) ) ) )
```

chercher5-segment

```
;;;rechercher le segment formé des L premiers termes de S
;;;dans E, N est une variable tampon.
(defun chercher5-segment (s l e n)
  (if (zerop l) n
      (if (consp e)
          (if (and (equal (car e) (car s))
                   (comparer5 (cdr s) (1- l) (cdr e)) )
              n
              (chercher5-segment s l (cdr e) (1+ n)) ) ) ) )
```

L'introduction de l'oracle correspond fondamentalement à un désir accru d'efficacité. On n'hésite donc plus à analyser plus avant le filtre afin d'en extirper des méthodes de filtrage plus rapides. L'ennui est que cela coûte de nombreux et répétés balayages de filtres qu'il serait plus sage de n'analyser qu'une seule fois. Toutefois la technique de l'oracle s'apparentant au *look-ahead* des analyseurs syntaxiques bat en brèche la séquentialité gauche-droite habituelle.

À essayer : *Rompre la séquentialité gauche-droite ?*

D'autres schémas sont encore concevables pour améliorer encore l'oracle. Des analyses de longueur permettraient qu'un oracle encore plus élaboré analyse le filtre `(??seq ?- ??seq)` , détermine que seules les listes ayant un nombre impair de termes ont une quelconque chance d'être acceptées. Confrontée alors à l'expression `(le prince d 'aquitaine a la tour abolie)`, l'oracle ne testerait que la longueur (8) et retournerait de suite `nil` sans même chercher si fin et début coïncident.

4.6.8 Ordonnancement et efficacité

Tous les moteurs précédemment vus n'étaient que des interprètes de filtres opérant dans le sens usuel de Lisp, c'est-à-dire de gauche à droite et par profondeur d'abord. La raison de ce choix est que les filtres sont représentés par des listes et que l'accès privilégié à cette structure s'effectue de gauche à droite et par profondeur d'abord, tout comme le font les interprètes.

Ce choix va à l'encontre du traitement de certains filtres comme `((??- AND TRUE -> ??thm) => ??thm)` dont le bon traitement consiste à remarquer que `??thm` est formé de l'expression originale privée de ses deux premiers termes et que cette valeur peut être réinjectée dans le filtrage du premier terme fixant ainsi tout de suite la taille que peut avoir `??-`. Il n'y a alors aucun retour en arrière : la méthode est déterministe. Au contraire, la méthode gauche-droite consiste à essayer différentes tailles pour `??-` (même si l'astuce consistant à chercher le symbole `AND` est payante) puis d'en déduire la valeur de `??thm` comme segment terminal du premier terme. Elle cherche alors à vérifier si la fin du filtre tient ses promesses quitte, s'il n'y a pas succès, à recommencer le filtrage du premier terme autant de fois qu'il y de symboles `AND` présents. Le lecteur s'en persuadera sur l'expression suivante, s'il a le courage de mener les calculs jusqu'à leur terme :

```
(( ((p AND q) -> p) AND ((p AND q) -> q) AND
   ((p AND q) -> (p AND q)) AND TRUE ->
   ((p AND q) -> (p OR q)) ) => ((p AND q) -> (p OR q)) )
```

Si l'on abandonne l'ordre gauche-droite, par profondeur d'abord et si l'on voit le filtrage comme un système inordonné d'équations, alors il peut être intéressant, pour des raisons d'efficacité, de transformer le filtre en un équivalent plus rapide. Marc Gengler [Gengler 88] a travaillé dans ce domaine et proposé une caractérisation des filtres et des schémas de composition de filtres après évaluation de leur complexité intrinsèque de résolution.

Divers niveaux de difficulté sont envisageables et nous nous proposons de décrire les filtres *simples*, *mous* et *durs*. Tout filtre d'élément élémentaire (comme `(*check atom)` ou `?-`) est simple : il ne prend qu'un seul terme et ne pose aucun problème. Un filtre comportant un filtre de segment terminal est simple car sa solution est immédiatement connue. Par exemple, le filtre `(Gall amant ??suite)` est simple puisque `??suite` est déterminée simplement.

Le filtre `(??debut magnanime)` est mou. Sa résolution est un peu moins simple que celle du précédent car il faut vérifier que l'expression à filtrer se termine par `magnanime`, `??debut` consiste alors en le segment initial de cette expression. Un filtre de segment suivi d'un ancrage constitue donc un filtre mou. Il en est de même de tout filtre qui ne contient qu'un seul filtre-segment comme `(Galamment ??milieu Nymes)`.

Les filtres durs sont tous les autres. Les filtres qui comportent au moins deux filtres-segments sont durs comme par exemple `(??mi et ??moi sont dans un bateau)` ou `(??palin ??drome)`. Tous deux conduisent probablement à de multiples solutions. Même si le premier semble légèrement plus mou que le second, tous deux relèvent néanmoins de la catégorie des "durs" !

Au terme de cette classification il importe, pour être efficace, de traiter les filtres simples d'abord, puis les mous et enfin seulement les durs. Cet ordre permet que les durs deviennent mous ou simples si les filtres-segments qu'ils contiennent ont pris des valeurs grâce aux filtrages antérieurs. Par exemple, voici un filtre simple composé des filtres mous et durs cités précédemment :

```
(Petit echantillon de phrases (1 ??palin)
 (2 (??palin ??drome) ??mi)
 (3 (??mi et ??moi sont dans un bateau))            Filtre (1)
 ??moi )
```

Traités dans l'ordre gauche-droite, tous les segments sont déterminés sans anicroche et simplement.

Pour réaliser cette innovation, nous allons introduire les simples filtres `*xcons` et `*proper-list`. `*xcons` est semblable à `*cons` sauf qu'il traite ses arguments en ordre inverse c'est-à-dire en commençant par le filtrage des `cdr` pour s'achever par celui des `car`.

```
(de-filtre5 (*xcons filtre1 filtre2) element                        *xcons
   (if (consp e)
     (filtrer5-element (cdr e) filtre1 env
        (et-puis5 (env alt)
```

```
          (filtrer5-element (car e) filtre2 env succ alt) )
        alt )
      (reprendre5 alt) ) )
```

De son côté, (`*proper-list` *filtre*) filtre ce que filtre son argument mais vérifie en outre que l'expression ainsi filtrée est une vraie liste i.e.a un ultime `cdr` égal à `()`.

`*proper-list`

```
(de-filtre5 (*proper-list filtre) element
   (if (null (last e))
       (filtrer5-element e filtre env succ alt)
       (filtrer5-reprendre alt) ) )
```

Donnons quelques exemples :

`(*xcons a b)`
: filtre la paire pointée `(b . a)`.

`(*xcons ?x do)`
: filtre les listes préfixées par `do` comme `(do re mi fa)`.

`(*xcons (??palin ??drome) (??palin))`
: On le reconnaît comme un filtre simple puisque `??palin` est déterminé en premier puis injecté dans `(??palin ??drome)`. Il accepte donc sans retour arrière l'expression `((ou vas tu) ou vas tu te dis je)`.

L'introduction des filtres de segment pose le problème de comparer `?x` et `(??x)`. Le premier prend toute expression tandis que le second, analogue à `(*list ??x)` ne filtre qu'une liste dont l'ultime `cdr` est `nil`. Si donc on assure cette condition (ce que réalise justement `*proper-list`), on a bien l'équivalence :

```
(*proper-list ?x)}  ≡ (??x)
```

Le filtre (1) précédent, simple quoique composé de sous-filtres durs, s'écrit alors :

```
(*list (*progn Petit echantillon de phrases)
       (*cons 1 (*proper-list ?palin))
       (*cons 2
              (*cons (*list ??palin
                            (*tail (*proper-list ?drome)) )
                     (*proper-list ?mi) ) )
       (*xcons (*proper-list ?moi)
               (*list ??mi et ??moi sont dans un bateau) ) )
```

Cette transformation a usé de `*cons` et de `*xcons` mais aussi de l'équivalence entre `(*proper-list ?x)` et `(??x)`.

Qu'en est-il de l'intérêt de ces transformations ? De fait, tout dépend des emplois du filtrage. Parmi les usages lispiens de ce style, peu de filtrages durs sont rencontrés. Les filtres conçus par les Lispiens peu ou prou chevronnés, s'accordent plutôt d'une syntaxe gauche-droite : les séquences sont soit terminales, soit bordées de mots-clés facilement reconnaissables : dans ces deux cas les filtres associés sont au plus mous. En outre, la comparaison de séquences ne se rencontre presque jamais, sauf dans des exemples démonstratifs comme `(??foo ??foo)`. Dans tous ces cas, il ne semble pas rentable de faire une analyse approfondie de la dureté du filtre.

4.6.9 Le moteur à pile

Le moteur à filtrer auquel nous avons abouti manipule deux continuations explicites. La plupart des calculs menés en Lisp disposent plutôt d'une continuation implicite et gèrent au moyen d'échappements (`catch`, `throw`, `block` et `return-from`) les continuations exceptionnelles. L'intérêt est avant tout l'efficacité puisqu'un codage implicite est toujours moins coûteux que l'allocation d'une fermeture. Cette catégorisation normale/exceptionnelle ne semble pourtant pas adaptée à notre moteur car il fait autant usage des continuations d'échec que de succès. Il est cependant possible de remarquer que les continuations d'échecs ne nécessitent aucune information en provenance du point où a été détecté l'échec et c'est donc elles qui peuvent migrer dans l'implicite (c'est-à-dire, plus vulgairement, la pile d'évaluation).

Le problème essentiel de cette variante est que si l'on ne se restreint pas à la première solution, il faut bâtir explicitement la continuation d'échec (`alt`) lorsque le filtrage doit retourner sa réponse. Cela peut néammoins se faire mais ne semble pas procurer d'avantages décisifs de vitesse. La programmation de `filtrer5` est récursive terminale et de ce fait assez véloce. Toutefois la prochaine variante, compilée, codera l'échec par `nil` et les continuations d'échec par des alternatives (`if` ou `or`) bien placées : le moteur sera donc à pile mais compilé !

Résumé

Ce chapitre est né du désir d'introduire des filtres de segments non anonymes. Ce faisant toute l'architecture précédemment obtenue a dû être refondue et adopter un style de programmation fonctionnel et par continuations. Le résultat est proche d'une sémantique dénotationnelle décrivant précisément les actions attendues du moteur à filtrer.

Toutes les données ont été dûment encapsulées y compris les continuations elles-mêmes et le code final obtenu a profité des techniques de programmation par objet, fonctions génériques et héritage simple.

De nombreux filtres ont été introduits dont les définitions n'excèdent pas une dizaine de lignes, ce qui prouve la qualité des préparatifs ayant préludé au moteur à filtrer.

Enfin quelques variantes ont été esquissées visant à améliorer l'efficacité de l'interprète de filtres.

Exercice 4.12 : Les macros `et-puis5` et `ou-sinon5` peuvent entraver la lecture d'un habitué de style à continuation. Réécrire le moteur à filtrer en supprimant tous les objets, toutes les fonctions génériques et toutes les macros superflues.

Exercice 4.13 : Écrire le filtre `*circle` qui reconnaît des listes circulaires dont les termes sont filtrables par les arguments de `*circle`. Par exemple, `(*circle ?x)` reconnaît la valeur de :

```
(let ((liste (list 'foo 'foo)))
    (nconc liste liste) )
```

`(*circle ?x ?x ?x)` l'accepte tout autant, par contre `(*circle ?x ?y ?x)` refuse la valeur de :

```
(let ((liste (list 1 2)))
    (nconc liste liste) )
```

Exercice 4.14 : Les fonctions `ancrage5-p` et `ancrer5` de l'oracle ne peuvent être étendues à de nouveaux filtres, puisque la connaissance de `*term`, `*terms`, `*quote` est encodée dans leurs définitions. Réécrire ces fonctions et profiter du fait que les filtres sont des objets.

Exercice 4.15 : Développer l'oracle pour que `deviner5` appliqué à `(on peut faire mieux)` et `(?- faire ??-)` retourne 2 et non 0. On ne regarde donc plus si le seul premier filtre du contexte droit est synchronisable mais si l'un des suivants l'est.

La compilation de filtres

5

Les signes ne sont pas des preuves, puisque n'importe qui peut en produire des faux ou d'ambigus. De là à se rabattre, paradoxalement, sur la toute-puissance du langage: puisque rien n'assure le langage, je tiendrai le langage pour la seule et dernière assurance: je ne croirai plus à l'interprétation.
Roland Barthes, *Fragments d'un discours amoureux*

Tous les filtreurs précédents, même s'ils étaient de qualité croissante et même si `filtrer5` en représentait la quintessence, appartiennent tous au genre "interprète". Confrontés à une donnée, ils décodent le filtre sempiternellement et ne peuvent partager ou réemployer les fruits de son interprétation : le travail doit être repris encore et encore à mesure que le filtre est sollicité. De plus, quel est l'intérêt d'écrire `(filtrer e '(a b c))` plutôt que `(equal e '(a b c))`, si ce n'est pour perdre du temps ! Comme le filtre condense toute l'information utile, nous nous proposons de le traduire en un code Lisp directement utilisable, où toute référence au filtre original aura disparue et, bien évidemment, d'exécution efficace. La suite de ce chapitre montrera comment ces buts sont atteints et le prix associé c'est-à-dire les restrictions de filtrage induites, ainsi que les nouveaux avantages obtenus.

Le compilateur qui sera présenté est complet et doté de nombreux raffinements entre autres perfectionnements. Nous ne détaillerons pas les méandres intellectuels ayant présidé à son invention qui tel Athéna surgira adulte.

La compilation des filtres est assez ancienne puisqu'on peut en trouver un exemple en InterLisp. Ce compilateur dû à Larry Masinter [Masinter 78] est de facture classique mais est moins développé que le nôtre.

Une intéressante voie pour la compilation des filtres est proposée par les techniques de l'évaluation partielle [Emanuelson & Haraldson 80], [Emanuelson 82] ou [Danvy 90]. Étant donné un interprète `filtrer5` et un filtre ϕ, on sait *spécialiser* `(lambda (e) (filtrer5 e '`ϕ`))` en une fonction efficace équivalente : c'est donc une compilation de ϕ. De très intéressants résultats ont été obtenus mais beaucoup reste à explorer encore dans cette approche. Nous ne mettrons donc ici en œuvre que des techniques de compilation classiques.

5.1 Mise en œuvre du filtrage

Remis dans une perspective historique, les filtreurs ont successivement introduit, pour leurs besoins propres, l'environnement et les continuations. Ils ont par là même augmenté le pouvoir de l'utilisateur par l'ajout des variables et la possibilité d'obtenir les différentes solutions d'un filtrage. À part quelques mentions éparses, comme la représentation des variables et la macro `match-let` (cf. page 22), peu a été fait pour l'utilisateur qui doit relancer des continuations ou décoder des environnements pour y extraire ce qu'il souhaite au moyen de fonctions rébarbatives encapsulant ces mêmes structures de données. Tâchons donc de remédier à ces défauts.

La mise en œuvre d'un filtrage s'accompagne de la mise en place de plusieurs intentions :

- un filtre,
- une expression à filtrer,
- une continuation de succès : c'est-à-dire un morceau de code exploitant les résultats du filtrage,
- une continuation d'échec : que faire quand rien ne va plus ?
- enfin, la volonté de ne filtrer qu'une seule fois ou bien d'obtenir plusieurs solutions.

La fonction `filtrer5` précédente avait comme signature :

$$\text{Expression} \times \text{Filtre} \rightarrow (\text{Environnement} \times \text{Continuation}) \cup \{\texttt{nil}\}$$

La fonction `filtrer5` prend une expression et un filtre en arguments, elle retourne `nil` ou un couple environnement-continuation. `filtrer5` conduit à une complexification importante de l'interface de sortie avec l'utilisateur puisque le code de succès doit extirper de l'environnement les valeurs des variables du filtre afin de calculer un résultat : c'est ce qu'énonce la signature :

$$\text{Environnement} \times \text{Code-Succès} \rightarrow \text{Résultat}$$

La macro `match-let`, quant à elle, condensait agréablement les deux précédentes signatures en :

$$\text{Expression} \times \text{Filtre} \times \text{Code-Succès} \rightarrow \text{Résultat} \cup \{\texttt{nil}\}$$

Mais la forme `match-let` fait perdre le caractère fonctionnel du filtrage où le filtre est vu comme une fonction unaire se saisissant d'une expression. Il faut donc "currifier"[1] `match-let` et ainsi obtenir `match-lambda`. Cette transformation rappelle le lien unissant `let` et `lambda`, la signature de `match-lambda` sera donc

$$\text{Filtre} \times \text{Code-Succès} \rightarrow (\text{Expression} \rightarrow \text{Résultat} \cup \{\texttt{nil}\})$$

On voit bien alors le caractère compilatoire de `match-lambda` qui dissocie l'analyse du filtre de ses utilisations postérieures.

5.2 Les macros d'interface

Il y en a essentiellement trois :

`(match-lambda` *filtre formes...*`)`

Une telle forme s'expansera en une fonction `(lambda (e) ... )` qui se saisira d'une expression, la filtrera et, en cas de succès, évaluera les *formes* dans l'environnement de filtrage. La valeur finale sera celle de `(progn` *formes...*`)` ou `NIL` en cas d'échec. On remarquera qu'ici les variables de filtrage apparaissant dans *formes* sont lexicales et de fait insensibles au contexte dynamique d'emploi du filtre. Prenons un exemple :

```
(match-lambda (*cons ?a ?d) (list d))
```

sera expansé en quelque chose ressemblant à :

```
(lambda (g1)
   (flet ((g2 (a d) (list d)))
         (and (consp g1)
              (let ((g3 (car g1)))
                   (let ((g4 (cdr g1)))
                        (g2 g3 g4) ) ) ) ) )
```

[1] Du mathématicien Haskell B. Curry qui usa de ce mécanisme.

Les variables présentes dans le corps de `match-lambda` sont mises en situation lexicale. La continuation de succès initiale est `g2` qui, lorsqu'elle est appelée, lie les variables de filtrage `a` et `d` à leur valeur de filtrage.

> Un bon compilateur (cette quête des temps modernes) verrait que `g2`, `g3` et `g4` ne sont utilisées qu'une seule et unique fois dans un contexte sans effet de bord aucun, ces variables peuvent donc être expansées là où elles sont utilisées. On obtient ainsi
>
> ```
> (lambda (g1)
>   (and (consp g1) (list (cdr g1))) )
> ```
>
> ce qui semble vraiment minimal ! Insistons que le précédent fragment n'est qu'*une* des expansions possibles.

L'échec total est codé comme `NIL`, c'est donc à l'appelant de `match-lambda` de prendre les mesures qui alors s'imposent. Comme le codage par `NIL` est une banalité en Lisp mais que, surtout, l'échec total ne retourne rien comme information (autre que le concept même qu'il y a eu échec total), on n'a pas à se soucier de continuations d'échec : il n'y en a pas ! Il peut toutefois être utile de traiter plusieurs solutions à un filtrage : procéder à un filtrage multiple. C'est l'objet de :

(`match-all-lambda` *filtre formes...*)

Une telle forme s'expansera en une fonction `(lambda (e) ... )` qui se saisira d'une expression, la filtrera et, en cas de succès, évaluera les *formes* dans l'environnement de filtrage, ignorera la valeur obtenue et poursuivra le filtrage dans les mêmes conditions. Quand toutes les solutions auront été énoncées, la valeur finale ramenée sera `NIL`. Il sera simple avec `match-all-lambda` et grâce à des effets de bord ou des échappements, d'extraire d'un filtrage ses deux premières solutions ou toutes celles de rang impair. Considérez, par exemple,

```
(let ((compteur 0)   ;extrait les deux premières solutions
      (solutions nil) )
  (block fin
     (match-all-lambda filtre
         (incf compteur)
         (push (progn code...) solutions)
         (when (>= compteur 2) (return-from fin)) ) )
  solutions )
```

Le corps de `match-all-lambda` sera invoqué au plus deux fois et s'échappera vers l'étiquette `fin` juste après le second appel. La valeur du tout est la liste des (au plus) deux premières solutions de filtrage. Il n'y a cependant là plus aucun moyen de relancer le filtrage.

D'autres combinaisons que `match-lambda` et `match-all-lambda` peuvent être utiles : `match-case-lambda` énonce une série de filtres et de formes associées à évaluer si le filtrage réussit.

(`match-case-lambda`
($filtre_1$ $formes_1$)
($filtre_2$ $formes_2$)
...   ...
($filtre_n$ $formes_n$))

Une telle forme possède un corps qui épouse la forme et le sens des clauses de `cond`. Derrière le mot-clé `match-case-lambda` se trouvent des clauses dont le `car` est un filtre et le `cdr` une suite de formes à évaluer si le filtrage réussit. Comme `cond`, les clauses sont essayées séquentiellement les unes après les autres jusqu'à en trouver une qui réussit. La valeur de la forme totale est alors celle des *formes* associées. Si toutes les clauses échouent, `match-case-lambda` retourne `NIL`.

La macro `match-case-lambda` est particulièrement intéressante pour des analyses de cas, on en verra de nombreux exemples dans les générateurs de code spécialisés, cf. page 76.

Nous fournirons aussi `match-let` et `match-case` qui s'expriment facilement comme :

match-let

```
(defmacro match-let (filtre expression &body faire)
  `(funcall (match-lambda ,filtre . ,faire)
            ,expression ) )
```

match-case

```
(defmacro match-case (expression regles)
  `(funcall (match-case-lambda . ,regles)
            ,expression ) )
```

Nous en fournirons d'autres versions plus efficaces plus loin.

Les schémas de filtrage sont si utiles que l'on peut penser à définir des fonctions ou macros entièrement fondées sur eux et intégrés à leur définition même. Voici donc les macros **`def-match-function`** et **`def-match-macro`** naïvement exprimées pour l'instant.

def-match-macro

```
(defmacro def-match-macro ((nom . filtres) . faire)
   (let ((g (gensym)))
     `(defmacro ,nom (&rest ,g)
        (funcall (match-lambda ,filtres
                   (return-from ,nom (progn . ,faire)) )
                 ,g )
        (error "~A : Syntaxe incorrecte :~%~
                *** Forme attendue      : ~A~%~
                *** Parametres actuels  : ~A."
               ',nom ',filtres ,g )) ) )
```

def-match-function

```
(def-match-macro
   (def-match-function (*setq nom (*check symbolp))
             (*ssetq doc-string (*maybe (*check stringp)))
             ??rules )
   (let ((g (gensym)))
     `(defun ,nom (&rest ,g) ,@doc-string
        (match-case ,g
          ,@rules
          ((*sexp)
           (error
             "Function ~A did not recognize its arguments."
             ',nom ) ) ) ) ) )
```

Les fonctions que `def-match-function` définit discriminent la liste de leurs arguments, cherchent la règle qui s'applique puis l'évaluent pour finir. Donnons un exemple :

lastelt

```
(def-match-function lastelt
  ( ((?x))     x)
  ( ((?x ??y)) (lastelt y)) )
```

La fonction **`lastelt`** retourne le dernier terme de la liste que l'on lui soumet en unique argument, c'est en quelque sorte **`(car (last`** *liste***`))`**. Cette formulation, si elle est pratique pour les fonction qui prennent un nombre quelconque d'arguments, l'est moins en cas d'arité fixe car encombrée de parenthèses superflues. Une autre syntaxe est donc possible (non procurée par la définition actuelle de **`def-match-function`**) qui use d'un marqueur syntaxique délimitant les filtres du corps des règles. Le précédent exemple se réécrira comme :

```
(def-match-function lastelt
  ( (?x)      => x)
  ( (?x ??y)  => (lastelt y)) )
```

Enfin une fonction ternaire comme `mappairs` (une sorte de `mapcar` binaire) s'écrira :

```
(def-match-function mappairs
  ( ?fn () ?-                 => ())
```

```
  ( ?fn ?- ()               => ())
  ( ?fn (?x ???y) (?xx ???yy) =>
        `(,(funcall fn x xx) . ,(mappairs fn y yy)) ) )
```

Ces deux derniers exemples sont tirés de [Peyton Jones 86] et exhibent bien le charme de cette programmation déclarative opérant par filtrage. De plus un œil suspicieux pourra s'en convaincre au chapitre 6, le code engendré est loin d'être inefficace !

Exercice 5.16 : Définir `match-case-lambda` à partir de `match-lambda`.

Exercice 5.17 : Définir `match-lambda` à partir de `match-all-lambda`.

5.3 Architecture

Fort de notre longue expérience du filtrage, nous fonderons l'architecture de notre compilateur suivant nombre des préceptes parmi ceux qui se sont auparavant dégagés. Le mécanisme de macro-filtres nous conduit à définir une sorte de jeu minimal de filtres dans lesquels tous les autres seraient exprimés. Cette obtention d'un jeu "RISC"[2] scinde le compilateur en deux parties :

- une première passe, dite de *normalisation*, traduit le filtre en un filtre équivalent mais conforme à ce jeu rudimentaire,
- une seconde partie engendre le code final à partir de ce filtre traduit.

Cette division se compare tout à fait aux techniques usuelles de compilation où un programme est traduit en une expression d'un certain langage intermédiaire avant que d'être expansé dans le langage cible. L'intérêt d'être en Lisp et d'adopter la philosophie lispienne, est que le langage intermédiaire est un sous-ensemble du langage de filtres et peut ainsi être mis au point et testé par un moteur comparable à **`filtrer5`**.

La première passe n'est pas à proprement parler une phase de normalisation du filtre initial vers le jeu réduit car plusieurs filtres équivalents peuvent être dérivés. On ne se privera d'ailleurs pas, au passage, d'effectuer quelques améliorations sur le filtre initial afin de le rendre plus efficace. La seconde passe est celle qui connaît le langage cible : c'est la seule à changer si l'on veut engendrer du Scheme ou du ML, plutôt que du Lisp. Le code résultat sera bien sûr lui-même compilé ou interprété par l'évaluateur Lisp sous-jacent. Le schéma global est celui de la figure 5.1.

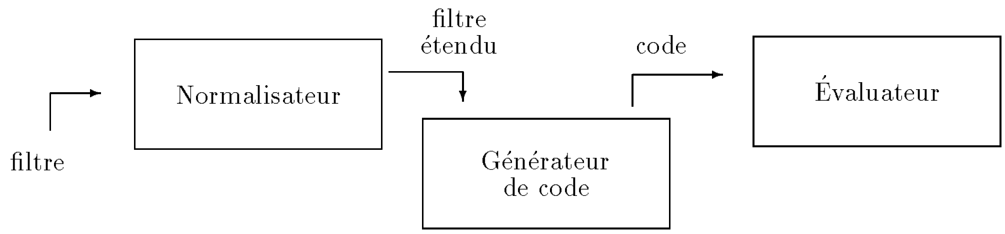


Figure 5.1: Architecture globale du compilateur de filtres

Conformément à la thèse d'Eugen Neidl [Neidl 83] qui considère qu'interprète et compilateur sont un même algorithme associé à des bibliothèques différentes : des bibliothèques d'exécution dans le cas de l'interprète et de génération de code dans le cas du compilateur, les traducteurs des deux passes respecteront l'architecture habituelle des moteurs à filtrer ce qui en facilitera l'exposition.

Le reste de ce chapitre définira tout d'abord le langage intermédiaire et sa compilation puis la normalisation des filtres vers le langage intermédiaire. Diverses extensions et améliorations seront exposées avant que ne soit dévoilé le florilège final.

[2]RISC pour *Reduced Instruction Set Computer* qualifie une philosophie d'architecture d'ordinateurs où l'accent est mis, entre autre, sur le faible nombre d'instructions présentes dans le code d'ordre.

5.4 Le langage intermédiaire

Le langage intermédiaire doit répondre à plusieurs critères.

- Il doit être suffisamment réduit pour limiter le travail du générateur de code qui est proportionnel au nombre de filtres différents qu'il doit connaître.
- Il doit être suffisamment large pour couvrir tout le langage de filtres dont dispose l'utilisateur.
- Chaque élément langagier doit être simple, c'est-à-dire n'effectuer qu'une seule chose mais la faire bien.
- Les éléments doivent être composables sans restriction bref "orthogonaux".
- Tous les filtres auront un nombre fixe d'arguments (une arité fixe) ne requérant plus aucune analyse syntaxique.

À l'instar de Lisp qui peut se définir en son propre langage, le langage intermédiaire sera un sous-ensemble du langage des filtres. On tâchera donc d'isoler les filtres primitifs et d'exprimer les autres en fonction de ces seuls-là. Nous introduirons toutefois quelques nouveaux filtres qui rempliront des tâches très précises et de très bas niveaux.

Aucun macro-filtre ne subsistera dans le langage intermédiaire : ils seront tous expansés auparavant et ainsi auront disparu. La restriction associée est que l'expansion doit être bornée et doit donc s'achever en un filtre dépourvu de macro-filtre. Ceci exclut, par exemple, la définition suivante pour `*any`

```
(de-filtre5 (*any) macro-segment
   (*oorr (*progn) (*progn (*sexp) (*any))) )
```

Cette définition récursive et calquée sur l'algorithme utilisé pour interpréter `*any` est malheureusement non finiment expansible. En termes plus mathématiques :

`??-`	$\equiv$	<*rien*>
	$\equiv$	`?- ??-`
	$\equiv$	`?- ?- ??-`
	$\equiv$	`?- ?- ?- ??-`
	...	

Et l'on voit bien qu'il subsiste toujours un `??-` dans l'expansion ce qui la rend infinie. Le même problème se rencontre en Lisp avec la macro `while` qui si elle est définie comme

```
(defmacro while (&whole call condition &rest body)
   `(if ,condition (progn (progn . ,body) ,call)))}
```

est non finiment expansible puisque la forme d'appel `call` se retrouve à l'identique dans son expansion. Comme le mécanisme de macro-filtres n'a été introduit que comme une facilité syntaxique permettant de condenser, sous une courte écriture, une composition quelconque de filtres et devant notre réticence à imaginer une sémantique pour les filtres infinis (quoique !?), nous interdirons donc ces derniers.

Voici donc, sans plus attendre, un tableau récapitulatif des filtres du langage intermédiaire. N'ayez crainte, ils seront tous bientôt explicités !

`**sexp`	`**quote`	`**check`
`**and2`	`**or2`	`**not`
`**setq`	`**eval`	`**cons`
`**ssetq-append`	`**end-ssetq`	`**eval-append`
`**labelled-times`	`**end-times`	`**success`
`**ssetq-terminal`	`**struct`	

Plusieurs intentions ont permis d'aboutir à ce jeu rudimentaire.

Supprimer les filtres de segments au profit des filtres d'éléments. Cette transformation est syntaxique puisque par exemple `(??x foo ??x)` se traduit en

```
(*ssetq-append x
    (*labelled-times g63
          (*cons (*sexp) (*end-times g63))
          (*end-ssetq x) )
    (*cons (*quote foo)
           (*eval-append x (*quote ())) ) )
```

dont une paraphrase est "`x` est un segment sans contrainte (comme l'est `??-`) suivi du symbole `foo` puis de la répétition de la valeur du segment `x` et enfin de `nil`".

Décomposer les effets : Les filtres liant des variables comme `?foo` ou `??bar` sont analysés et traduits différemment suivant qu'ils sont values ou pas.

`?foo` deviendra `(*setq foo (*sexp))`
ou bien `(*eval foo)`

Les effets mis en œuvre par ces filtres sont ainsi plus rudimentaires et de traduction plus aisée.

Éviter tout filtre composé qui peut donc s'exprimer par composition d'autres. Ainsi en est-il de `??-` qui n'est autre que `(*times ?-)`.

Respecter l'ordre gauche-droite : Cela peut sembler une régression mais ce sera le rôle de la première passe que d'ordonner la traduction. Les filtres rudimentaires représentent une sorte d'assembleur des actions de filtrage et comme tels s'accomodent mieux d'une sémantique séquentielle simple. Cette rusticité de l'assembleur n'impose pas que la première phase de traduction soit stupidement faite, au contraire elle est un gage d'efficacité puisque le code engendré sera très exactement celui spécifié et nul couche pseudo-intelligente ne viendra s'interposer entre le filtre et sa réalisation programmatique.

La sémantique des filtres rudimentaires sera exprimée tout d'abord informellement puis sous forme d'un nouveau moteur à filtrer très similaire à `filtrer5`. Ce nouveau moteur propre aux seul filtres rudimentaires respecte les canons de la sémantique dénotationnelle.

5.5 Sémantique des filtres

Précisons la sémantique de ces divers filtres regroupés en catégories homogènes. Cette description peut servir de manuel de référence. Afin de mieux marquer les filtres rudimentaires, leurs noms seront préfixés de deux étoiles.

`(**sexp)`

Ce filtre accepte tout et n'importe quoi, et sans vérification en plus !

`(**quote` *expression*`)`

Ce filtre n'accepte que des expressions égales (par `equal`) à *expression*.

`(**check` *prédicat*`)`

Ce filtre n'accepte que les expressions qui satisfont le *prédicat*. `**check` semble a priori plus rudimentaire que `**sexp` ou `**quote` qui ne sont autres que

```
(**quote expression) ≡
     (**check (lambda (e) (equal e 'expression))

(**sexp) ≡ (**check (lambda (e) t))
```

Mais devant l'immense difficulté que représente l'analyse du corps des prédicats, on conservera `**sexp` et `**quote` dont les intentions et les sémantiques sont claires. On verra notamment, lors des améliorations que l'on peut apporter aux filtres, que `(**or2 (**sexp) (**sexp))` n'est autre que `(**sexp)`. Cette propriété est indépendante du contexte lexical du filtre alors que leurs équivalents en terme de `**check` sont sensibles aux valeurs lexicales locales de `t` et `equal`.

`(**and2` *filtre*$_1$ *filtre*$_2$`)`

Ce filtre réalise la conjonction de ses deux filtres arguments. Il filtre toute expression que ses arguments filtrent l'un comme l'autre. Le filtre `**and2` possède exactement deux arguments.

`(**or2` *filtre*$_1$ *filtre*$_2$`)`

Ce filtre réalise une disjonction, il filtre toute expression qu'un au moins de ses arguments filtre. Tout comme son compère `**and2`, `**or2` possède exactement deux arguments.

`(**not` *filtre*`)`

Ce filtre réalise une négation, il filtre toute expression non filtrable par son argument. Attention, l'environnement résultant de *filtre* est perdu et ne peut être utilisé à l'extérieur de `**not`. On peut par exemple écrire `(*not (*cons ?x ?x))` sous la forme :

```
(**not (**cons (**setq x (**sexp))
               (**eval x) ))
```

pour refuser les paires pointées qui ont même `car` et `cdr` mais on ne peut utiliser la variable de filtrage `x` à l'extérieur de `**not`.

`(**setq` *nom filtre*`)`

Ce filtre réalise une liaison de la variable de filtrage *nom* avec toute expression que filtre *filtre*.

`(**eval` *nom*`)`

Ce filtre ne peut s'employer que dans un contexte où la variable de filtrage *nom* est définie. Il ne filtre que des expressions égales (par `equal`) à la valeur attribuée à *nom*.

`(**cons` *filtre*$_1$ *filtre*$_2$`)`

Ce filtre n'accepte que des paires pointées dont les `car` et `cdr` sont filtrables par ses deux arguments.

`(**ssetq-append` *nom filtre*$_1$ *filtre*$_2$`)`

Ce filtre n'accepte que des listes filtrables par `(*list` *filtre*$_1$ `(*tail` *filtre*$_2$`))`. Toutefois, dès que le début de l'expression est filtrable par *filtre*$_1$, ce début d'expression est lié à la variable de segment *nom* et le filtrage du reste de l'expression par rapport à *filtre*$_2$ est effectué dans cet environnement enrichi. Une marque syntaxique marque la fin de *filtre*$_1$: `**end-ssetq`.

`(**end-ssetq` *nom*`)`

Ce filtre marque la fin d'un filtre définissant une variable de segment. Ce filtre est purement technique et commande la liaison effective de *nom* à sa valeur. Donnons un exemple :

```
(**ssetq-append words
   (**cons (**quote foo)
           (**cons (**quote bar)
                   (**end-ssetq words) ) )
   (**cons (**quote hux) (**quote ())) )
```

Si ce filtre est confronté à l'expression `(foo bar hux)` alors il lie la variable de segment `words` au segment "`foo bar` ...". En d'autres termes encore, `**end-ssetq` correspond à la fonction syntaxiques des points de suspension présents dans le segment "`foo bar` ...".

`(**eval-append` *nom filtre*`)`

Ce filtre ne peut être employé que dans un contexte où la variable de segment *nom* est valuée, il accepte toute liste dont le début est filtrable par la valeur de *nom* et dont la fin est filtrable par *filtre*.

`(**labelled-times` *nom filtre*$_1$ *filtre*$_2$`)`

Ce filtre n'accepte que les listes dont le début est un certain nombre de fois filtrable par $filtre_1$ et dont la fin est filtrable par $filtre_2$. Le filtre de premier argument $filtre_1$ doit s'achever par la marque syntaxique `**end-times`.

(`**end-times` *nom*)

Ce filtre marque un point de choix pour une éventuelle répétition du premier filtre argument du `**labelled-times` englobant (et qui doit bien sûr porter le même *nom*). Donnons un exemple, le filtre `(??- foo)` sera traduit en :

```
(**labelled-times g43
     (**cons (**sexp) (**end-times g43))
     (**quote (foo)) )
```

Il est simple de se représenter ce filtre comme la grammaire équivalente

```
<g43> ::=  ( ?-  .  <g43>  )
       |   ( foo )
```

qui en est le strict décalque.

(`**success` *forme*)

Ce filtre évalue *forme* dans le contexte lexical courant, c'est-à-dire dans un environnement où les variables de filtrage valuées sont transformées en variables lexicales semblablement valuées. La réussite ou l'échec du filtre `**success` est conditionnée par la valeur de *forme*.

Une mauvaise appréciation de ce qui est valué ou pas peut conduire à des captures inopinées de variables incongrues. Par exemple,

```
(let ((x "valeur biscornue")
      (y "valeur biscornue") )
   (match-all-lambda (*or ?x ?y)
        (print ‘(x = ,x et y = ,y)) ) )
```

est une expression qui confrontée à `(foo bar)` imprimera :

```
(X = (FOO BAR) ET Y = "valeur biscornue")
(X = "valeur biscornue" ET Y = (FOO BAR))
```

Le filtre traduit de `(*or ?x ?y)` sera quant à lui :

```
(**or2 (**and2 (**setq x (**sexp))
               (**success (print ‘(x = ,x et y = ,y))) )
       (**and2 (**setq y (**sexp))
               (**success (print ‘(x = ,x et y = ,y))) )
```

Le premier des filtres `**success` s'évalue dans un contexte qui ne connaît pas la variable de filtrage `y` tandis que le second ignore `x`. Un avertissement écrit sera proféré à l'encontre de l'insolent qui oserait braver la recommandation : "Tes branches d'alternatives équilibreras". Ce qui, en clair, signifie qu'il vaut mieux, pour éviter d'intempestives captures, s'assurer que toutes les façons de filtrer définissent les mêmes ensembles de variables de filtrage. Cette vérification sera assurée par le normalisateur qui sanctionnera ces incartades en signalant une erreur.

La seule structure ici permise est la paire pointée mais comme suggéré dans l'exercice 3.8 d'autres structures pourraient, elles aussi, être désossées. Aussi a-t'on créé `**struct`

(`**struct` *type champ filtre* ...)

Ce filtre accepte toute instance de *type* (ou de ses sous-types) dont les valeurs des *champs* satisfont les *filtres* associés. Si la paire pointée était la structure doté des champs `car` et `cdr`, alors (`*cons` $filtre_1$ $filtre_2$) serait analogue à (`*struct cons :car` $filtre_1$ `:cdr` $filtre_2$). L'ordre des champs de la structure n'a pas à être respecté dans le filtre `**struct`, par contre l'ordre des filtres importe. Le fameux filtre `*xcons` (cf. page 52) n'est autre que (`*struct cons :cdr` $filtre_2$ `:car` $filtre_1$) où le `cdr` est filtré avant le `car`.

Enfin le dernier des filtres rudimentaires est dû à des considérations bassement lispiennes et tente d'éviter les tests fastidieux de fin de listes. Un filtre segmentiel terminal comme `reste` dans `(?x ?y ??reste)` impose que `reste` soit une vraie liste et donc que son ultime `cdr` soit bien `nil`. Si l'on était dans un langage typé comme ML, on n'aurait pas à se soucier de ce détail puisque les listes y sont homogènes. En Lisp et en revanche, on tolère souvent d'obscènes queues de listes. Introduisons donc `???`*variable* pour `**ssetq-terminal` *variable*) qui réalise une liaison de type segment qui accepte tout ce qui satisfait à `listp` c'est-à-dire `nil` ou une paire pointée et dans ce dernier cas sans même vérifier que c'est bien une vraie liste[3]. Toute tentative d'utilisation de la valeur ainsi liée provoquera l'émission d'un message d'avertissement et, à l'exécution, toute comparaison avec une valeur pointée sera signalée par un vigoureux message d'erreur. Telle est la sémantique de :

(**ssetq-terminal *nom*)

Comme dans tout choix une certaine part d'arbitraire a régné et un temps assez long a été nécessaire pour dégager ce jeu de filtres. Mais il est suffisamment réduit, orthogonal, simple et plaisant pour que l'on puisse envisager sereinement de travailler avec.

5.6 Les représentations

Avant d'aborder le moteur rudimentaire, donnons quelques indications sur les représentations que nous avons choisies pour cette dernière variante. Elles ne diffèrent que peu de celles élues pour `filtrer5` aussi les passerons-nous rapidement en revue.

5.6.1 Les continuations

Deux sortes peuvent être distinguées suivant qu'elles concernent l'une ou l'autre des deux grandes phases du compilateur. Interprète et compilateur usent de :

```
(defmacro et-puis6 (variables . filtrage)                      et-puis6
  `#'(lambda ,variables . ,filtrage) )
(defmacro ou-sinon6 (filtrage)                                 ou-sinon6
  `#'(lambda () ,filtrage) )
(defmacro reussir6 (e env succ alt)                            reusir6
  `(funcall ,succ ,e ,env ,alt) )
(defmacro reprendre6 (alt)                                     reprendre6
  `(funcall ,alt) )
```

On y remarquera l'absence de toute précaution, comme pratiquées dans la version expérimentale `filtrer5`, puisque nous présentons maintenant le code définitif, désencombré des verrues de mise au point. Certaines continuations sont plus courantes que d'autres ainsi :

```
(defvar ou-sinon6nil)                                          ou-sinon6nil
(setq ou-sinon6nil (ou-sinon6 'nil))

(defvar et-puis6t)                                             et-puis6t
(setq et-puis6t (et-puis6 (e1 env1 alt1) 't))
```

5.6.2 L'environnement

L'environnement est créé et manipulé par les fonctions qui suivent :

```
(defun initial6env () nil)                                     initial6env

;;; Rechercher la première liaison concernant NOM
```

[3] Similairement aux `???`*variable*, on introduira `???-` qui est l'attrape-tout pluriel non sûr. Il ne peut être utilisé qu'en position terminale et accepte donc toute queue de liste sans la vérifier. `???-` n'a guère besoin d'être primitif puisqu'il n'est autre que `(*tail (*sexp))`.

```
;;; et ayant un certain TYPE
(defun env6-liaison (nom env type)
    (find-if #'(lambda (lien)
                 (and (equal nom (liaison6-nom lien))
                      (subtypep (type-of lien) type) ))
             env ) )
```
env6-liaison

```
;;; Rechercher la liste des succès
(defun env6-succes (env)
   (env6-liaison 'succes env 'liaison6succes) )
```
env6-succes

```
;;; Retourner la liste des variables qui diffèrent dans ENV1 et ENV2.
(defun differer6env (env1 env2)
  (set-exclusive-or (env6-variables env1)
                    (env6-variables env2) ) )
```
differer6env

```
;;; S'assurer que ENV1 et ENV2 ont les mêmes variables.
(defun comparer6env-p (env1 env2)
   (null (differer6env env1 env2)) )
```
comparer6env-p

```
;;; Extraire les variables de l'environnement (en normalisation ou
;;; compilation). Remember, les variables de strockage sont en fin d'ENV.
(defun env6-variables (env)
   (mapcan
     #'(lambda (lien)
           (typecase lien
             ;;; Sert pendant la normalisation
             (liaison6element
              (if (liaison6element-variable lien)
                nil
                (list (liaison6-nom lien)) ) )
             ;;; Sert pendant la normalisation
             (liaison6segment
              (if (liaison6segment-debut-ref lien)
                nil
                (list (liaison6-nom lien)) ) )
             ;;; Sert pendant la compilation
             (liaison6variable (list (liaison6-nom lien)))
             (t nil) ) )
     env ) )
```
env6-variables

```
;;; Fusionner (lors de la normalisation) deux environnements
(defun fusionner6env (env1 env2)
  (reduce #'(lambda (lien env)
              (ajouter6 env lien) )
          env1
          :initial-value env2
          :from-end t) )
```
fusionner6env

Les environnements sont des listes de liaisons associant un nom et diverses informations liées. Les liaisons adoptent une structure d'objet.

```
(defclass liaison6
  :slots (nom) )
(defclass liaison6variable :super liaison6
  :methods (engendrer6acces) )
```
liaison6

liaison6variable

liaison6variable-terminale

```
(defclass liaison6variable-terminale
```

```
    :super liaison6variable
    :methods (engendrer6acces) )
```

liaison6variable-presque-terminale

```
  (defclass liaison6variable-presque-terminale
    :super liaison6variable-terminale )
```

liaison6element

```
  (defclass liaison6element :super liaison6variable
    :slots (variable)
    :methods (engendrer6acces) )
```

liaison6segment

```
  (defclass liaison6segment :super liaison6variable
    :slots (debut-ref fin-ref)
    :methods (engendrer6acces) )

  ;;; Pour la compilation des boucles **labelled-times
```

liaison6boucle

```
  (defclass liaison6boucle :super liaison6
    :slots (fonction) )

  ;;; Pour la compilation des fonctions de succès
  ;;; La Alist est une liste de type succes6 (cf plus bas)
```

liaison6succes

```
  (defclass liaison6succes :super liaison6
    :slots (alist) )

  ;;; Pour enregistrer le résultat de tests booléns
```

liaison6property

```
  (defclass liaison6property :super liaison6
    :slots (bool) )
  ;;; Déclare que (consp NOM) = BOOL
```

liaison6consp

```
  (defclass liaison6consp :super liaison6property)
  ;;; Déclare que (equal NOM VALUE) = BOOL
```

liaison6equal

```
  (defclass liaison6equal :super liaison6property
    :slots (value) )

  ;;; Pour la compilation des **setq-append
```

liaison6ssetq

```
  (defclass liaison6ssetq :super liaison6
    :slots (succ occurrences) )
```

Quelques types de liaisons sont nouveaux comme **`liaison6property`** qui permet de représenter une qualité booléenne, par exemple une instance de **`liaison6consp`** assure que son *nom* répond *bool* au prédicat **`consp`**. Cette liaison est utile lors de certaines améliorations surfaciques de génération de code et élimine quelques tests superflus.

La liaison **`liaison6succes`** rassemble la liste de tous les contextes lexicaux de succès déjà rencontrés et permet ainsi de partager ceux-ci. Enfin la liaison **`liaison6boucle`** sert à la génération de code des répétitions.

Des fonctions spécialisées permettent d'ajouter des liaisons à un environnement, ou de les comparer.

```
  ;;; Pendant la compilation, enregistre que la variable NOM est liée
```

ajouter6variable

```
  (defun ajouter6variable (env nom)
    (ajouter6 env (make-liaison6variable :nom nom)) )

  ;;; Pendant la compilation enregistre que la variable NOM est liée mais
  ;;; était située en position segmentielle terminale
```

ajouter6variable-terminale

```
  (defun ajouter6variable-terminale (env nom)
    (ajouter6 env (make-liaison6variable-terminale
                    :nom nom )) )
```

```
;;; Pendant la compilation enregistre que la variable NOM est liée mais
;;; était située en position segmentielle terminale. Par contre il n'a pas
;;; été vérifié qu'elle satisfait (endp (cdr (last .))) ! Il est donc
;;; déconseillé de l'utiliser pour sa valeur dans le filtrage.
```

ajouter6variable-presque-terminale

```
(defun ajouter6variable-presque-terminale (env nom)
  (ajouter6 env (make-liaison6variable-presque-terminale
                 :nom nom )) )
```

```
;;; Pendant la normalisation signifie que la variable d'élément NOM est liée
;;; (variable est alors NIL). Pendant la compilation enregistre la variable
;;; de stockage associée. La liaison est alors marquée par liaison6variable.
```

ajouter6element

```
(defun ajouter6element (env nom var)
    (ajouter6 env (make-liaison6element
                   :nom nom :variable var )) )
```

```
;;; Pendant la normalisation siginife que la variable de segment NOM est liée
;;; (debut-ref et fin-ref sont alors NIL). Pendant la compilation, enregistre
;;; les variables de stockage associées. Pour une variable segmentielle
;;; terminale, seule la variable de stockage debut-ref sert, l'autre
;;; déclare alors un avertissement "var lex inutilisée".
```

ajouter6segment

```
(defun ajouter6segment (env nom debut fin)
    (ajouter6 env (make-liaison6segment :nom nom
                                 :debut-ref debut
                                 :fin-ref fin ) ) )
```

ajouter6boucle

```
(defun ajouter6boucle (env nom fn)
    (ajouter6 env (make-liaison6boucle
                 :nom nom :fonction fn )) )
```

ajouter6consp

```
(defun ajouter6consp (env nom)
   (ajouter6 env (make-liaison6consp :nom nom :bool t)) )
```

ajouter6equal

```
(defun ajouter6equal (env nom value bool)
  (ajouter6 env (make-liaison6equal :nom nom
               :bool bool :value value )) )
```

ajouter6succes

```
(defun ajouter6succes (env)
  (ajouter6 env (make-liaison6succes :nom 'succes
               :alist nil )) )
```

ajouter6ssetq

```
(defun ajouter6ssetq (env nom succ)
  (ajouter6 env (make-liaison6ssetq :nom nom
                 :succ succ )) )
```

```
;;; La fonction générique d'ajout
```

ajouter6

```
(defun ajouter6 (env lien)
   (adjoin lien env :test #'comparer6liaison) )
```

comparer6liaison

```
(defun comparer6liaison (lien item)
  (and (equal (liaison6-nom lien) (liaison6-nom item))
       (equal (type-of lien) (type-of item)) ) )
```

Quelques fonctions supplémentaires permettent de lire l'environnement mieux que ne le fait `env6-liaison`.

```
;;; Détermine si une variable est liée (en normalisation ou compilation)
```

variable6liee-p

```
(defun variable6liee-p (var env)
  (let ((lien (env6-liaison var env 'liaison6variable)))
```

```
    (if lien
      (etypecase lien
        (liaison6element
           (not (liaison6element-variable lien)) )
        (liaison6segment
           (not (liaison6segment-debut-ref lien)) )
        (liaison6variable t) ) ) ) )
```

stockage

```
;;; retourne le lien contenant les noms des variables de stockage
(defun stockage (var env)
  (let ((lien1 (env6-liaison var env 'liaison6element))
        (lien2 (env6-liaison var env 'liaison6segment)) )
    (or lien1 lien2) ) )
```

variable6terminale-p

```
;;; Détermine si NOM est une variable segmentielle terminale
(defun variable6terminale-p (nom env)
  (env6-liaison nom env 'liaison6variable-terminale) )
```

variable6presque-terminale-p

```
(defun variable6presque-terminale-p (nom env)
  (env6-liaison nom env
    'liaison6variable-presque-terminale ) )
```

5.6.3 La représentation des filtres

Les filtres seront comme naguère représentés par des objets.

filtre6

```
(defclass filtre6
  :slots (nom definitions syntaxe variables filtre) )
```

executable6

element6

segment6

```
(defclass executable6    :super filtre6
  :slots (normalisation) )
(defclass element6       :super executable6)
(defclass segment6       :super executable6)
```

rudimentaire6

```
(defclass rudimentaire6  :super filtre6
  :slots (compilation interpretation
          amelioration extraction ) )
```

macro6

macro-element6

macro-segment6

```
(defclass macro6         :super filtre6
  :slots (expansion) )
(defclass macro-element6 :super macro6)
(defclass macro-segment6 :super macro6)
```

Les filtres exécutables sont les filtres réservés à l'utilisateur et se scindent toujours en deux catégories `element` ou `segment`, tout comme les macro-filtres. Des filtres rudimentaires, il n'y en aura qu'une seule sorte mais capable d'être compilée, interprétée voire améliorée. Les filtres de l'utilisateur pourront seulement être normalisés.

Enfin les filtres seront classiquement archivés et récupérés comme suit :

*filtre

```
;;; La table où seront archivés tous les filtres
(defvar *filtres*)

(setq *filtres* (make-hash-table))
```

obtenir6filtre

```
;;; convertir un nom en un filtre ou
;;; répondre NIL si ce n'est pas un filtre.
(defun obtenir6filtre (nom)
   (gethash nom *filtres*) )
```

```
;;; reconnaître si une expression est un filtre
(defun filtre6type-p (exp type)
   (and (consp exp)
        (symbolp (car exp))
        (obtenir6filtre (car exp))
        (typep (obtenir6filtre (car exp)) type)) )
```

filtre6type-p

5.7 Le moteur rudimentaire

Les filtres rudimentaires respectent une certaine syntaxe malaisée à décrire sous forme d'une grammaire, fût-elle en forme de Backus-Naur ! Par exemple `(**cons (**eval foo) (**quote ()))` n'a pas de sens puisque `foo` n'est pas valuée. Pour préciser la sémantique des filtres rien ne vaut une bonne sémantique opérationnelle et pour cela un programme fonctionnel sans effet de bord ! Voici donc un nouveau moteur à filtrer spécialisé dans le traitement des filtres rudimentaires et qui fixera, dans le marbre des octets, la sémantique de ces filtres.

filtrer6

```
(defun filtrer6 (e f env succ alt segvarenv timesvarenv)
  (let ((filtre (obtenir6filtre (car f))))
    (unless filtre
      (error "Je ne connais pas de filtre nomme ~S."
             (car f) ) )
    (let ((fn (rudimentaire6-interpretation filtre)))
       (if fn (funcall fn e f env succ alt
                       segvarenv timesvarenv )
           (error "je ne sais pas interpreter le filtre ~A."
                  (car f) ) ) ) ) )
```

Les filtres rudimentaires sont des objets de la classe `rudimentaire6` qui possèdent chacun une méthode d'interprétation mise en œuvre par `filtrer6`. Quelques variables nouvelles sont apparues qui ont pour nom `segvarenv` et `timesvarenv`. Les marques syntaxiques `**end-ssetq` et `**end-times` ne peuvent être employées qu'à bon escient : l'environnement `segvarenv` indique quel est le nom de la variable segmentielle en cours de définition (s'il y en a une) tandis que `timesvarenv` indique le nom de la répétition en cours (s'il y en a une). Ces environnements n'ont jamais qu'une profondeur unité puisque les filtres `ssetq-append` et `labelled-times` doivent être bien enchassés.

Les méthodes qui suivent définissent tous les filtres rudimentaires et précisent finement leur sémantique. Leur lecture est similaire à celles que l'on a pu faire des filtres des chapitres précédents sauf que ceux-ci sont beaucoup plus simples puisque leurs effets sont réduits. Une différence toutefois, comme il n'y a plus que des filtres d'éléments parmi les filtres rudimentaires, les continuations de succès invoquées par `reussir6` auront l'expression courante comme premier argument. Celle-ci ne sera utilisée que par les ex-filtres de segments : `**ssetq-append`, `**eval-append` et `**labelled-times`.

Voici donc les filtres élémentaires, simples et standard, sans autre commentaire que leur sempiternel code.

**sexp

```
(de-filtre6 (**sexp) rudimentaire6
  :interpretation
    (reussir6 e env succ alt) )
```

**quote

```
(de-filtre6 (**quote expression) rudimentaire6
  :interpretation
    (if (equal e expression)
      (reussir6 e env succ alt)
      (reprendre6 alt) ) )
```

**check

```
(de-filtre6 (**check predicat) rudimentaire6
  :interpretation
```

```
    (if (funcall predicat e)
      (reussir6 e env succ alt)
      (reprendre6 alt) ) )
```

Les inoubliables filtres booléens ne varient guère non plus.

```
(de-filtre6 (**or2 filtre1 filtre2) rudimentaire6                    **or2
  :interpretation
    (filtrer6-element e filtre1 env succ
       (ou-sinon6 (filtrer6-element e filtre2 env succ alt
                                    segvarenv timesvarenv ))
       segvarenv timesvarenv ) )
```

```
(de-filtre6 (**and2 filtre1 filtre2) rudimentaire6                   **and2
  :interpretation
    (filtrer6-element e filtre1 env
           (et-puis6 (e1 env1 alt1)
                (filtrer6-element e filtre2 env1 succ alt1
                                  segvarenv timesvarenv ) )
           alt segvarenv timesvarenv ) )
```

```
(de-filtre6 (**not filtre) rudimentaire6                             **not
  :interpretation
    (filtrer6-element e filtre env
        (et-puis6 (e1 env1 alt1) (reprendre6 alt))
        (ou-sinon6 (reussir6 e env succ alt))
        segvarenv timesvarenv ) )
```

Et maintenant, le traitement des variables d'éléments ! L'environnement **env** contient les valeurs des variables déjà valuées, il suffit de l'inspecter pour interdire d'affecter une variable qui le fût déjà. On relance bien sûr la continuation avec un environnement enrichi d'une liaison de type élément.

```
(de-filtre6 (**setq nom filtre) rudimentaire6                        **setq
  :interpretation
    (let ((lien (env6-liaison nom env 'liaison6variable)))
      (if lien
        (filtre6-erreur
          "MATCH: la variable ~A est deja liee." nom )
        (filtrer6 e filtre env
             (et-puis6 (e1 env1 alt1)
                (reussir6 e (ajouter6element env1 nom e)
                          succ alt1 ) )
             alt segvarenv timesvarenv ) ) )
```

On interdit cependant que soient mélangées variables de type élément ou segment. Ceci n'est pas une restriction de puissance puisque l'on peut toujours écrire :

... `(??v)` ... `??v` ... *à la place de* ... `?v` ... `??v` ...
et ... `??v` ... `(??v)` ... *à la place de* ... `??v` ... `?v` ...

Par contre les filtres sont ainsi mieux typés. Les variables segmentielles sont de type *liste de quelquechose* tandis que les variables de type élément ne sont pas restreintes. Voici donc le filtre **`**eval`**.

```
(de-filtre6 (**eval nom) rudimentaire6                               **eval
  :interpretation
    (if (let ((lien (env6-liaison nom env 'liaison6variable)))
          (if lien
            (etypecase lien
              (liaison6element
               (equal e (liaison6element-variable lien)) )
              (liaison6segment
               (filtre6-erreur
                "MATCH : ~A est une variable element"
```

```
              nom ) ) )
          (filtre6-erreur
           "MATCH : la variable ~A est segment."
           nom ) ) )
    (reussir6 e env succ alt)
    (reprendre6 alt) )
```

La reconnaissance des paires pointées est égale à elle-même.

`**cons`

```
(de-filtre6 (**cons filtre1 filtre2) rudimentaire6
  :interpretation
    (if (consp e)
      (filtrer6-element (car e) filtre1 env
         (et-puis6 (e1 env1 alt1)
           (filtrer6-element (cdr e) filtre2 env1
             succ alt1 segvarenv timesvarenv ) )
         alt (initial6env) (initial6env) )
      (reprendre6 alt) ) )
```

La reconnaissance des structures ne varie pas non plus.

`**struct`

```
(de-filtre6 (**struct type . champs) rudimentaire6
  :interpretation
    (flet ((concat (symba str symbb)
              (find-symbol (concatenate 'string
                   (symbol-name symba) str
                   (symbol-name symbb) ) ) ))
      (if (subtypep (type-of e) type)
        (labels
          ((analyser (champs env succ alt)
             (if (consp champs)
               (filtrer6
                 (funcall (symbol-function
                            (concat type "-" (car champs)) )
                           e )
                 (cadr champs) env
                 (et-puis6 (e1 env1 alt1)
                     (analyser (cddr champs) env1
                               succ alt1 ) )
                 alt (initial6env) (initial6env) )
               (reussir6 e env succ alt) ) ))
          (analyser champs env succ alt) )
        (reprendre6 alt) ) )
```

La fonction locale `concat` synthétise un nouveau nom d'accesseur à partir du nom du type, d'un tiret et du nom du champ. Il est bien sûr nécessaire que les noms standard aient été utilisés lors de la définition de la structure car il n'existe pas de moyen de les récupérer.

Le traitement des variables segmentielles est séparé des variables de type élément. La définition de `**ssetq-append` montre très précisément quand a lieu l'affectation : entre le filtrage de *filtre1* et celui de *filtre2*. On peut aussi apercevoir que le filtrage de *filtre1* est fait avec un `segvarenv` vide puisque les `**end-ssetq` ne peuvent figurer qu'en *filtre1*. Il est bien sûr aussi interdit de faire finir une répétition à l'intérieur de *filtre1* ce que marque l'environnement nul soumis en `timesvarenv`.

`**ssetq-append`

```
(de-filtre6 (**ssetq-append nom filtre1 filtre2) rudimentaire6
  :interpretation
    (let ((lien (env6-liaison nom env 'liaison6variable)))
      (if lien
        (filtre6-erreur
         "MATCH : la variable ~A est deja liee a un segment."
          nom )
        (filtrer6 e filtre1
```

```
          env
          (et-puis6 (e1 env1 alt1)
              (filtre6-erreur
               "MATCH : l'affectation a ~A est inachevee."
               nom ) )
          alt
          (ajouter6element (initial6env) nom
            (et-puis6 (e1 env1 alt1)
              (filtrer6 e1 filtre2
                 (ajouter6segment env1 nom e e1)
                 succ alt1 segvarenv timesvarenv ) ) )
          (initial6env) ) ) )
```

La marque syntaxique permet de relancer le filtrage après l'affectation ad hoc. La continuation de succès de la seconde partie du filtre `**ssetq-append` associé est conservée dans l'environnement `segvarenv` : elle relance le filtrage de *filtre2* en enregistrant la liaison de *nom* au segment débutant en `e` et s'achevant en `e1` exclu. La curieuse continuation de succès figurant en `**ssetq-append` ne sera déclenchée que s'il n'y a pas de `**end-ssetq` associé comme dans le filtre erroné `(**ssetq-append x (**quote foo) (**quote nil))`. La "vraie" continuation de succès est celle figurant dans `segvarenv` et invoquée par `**end-ssetq`.

```
(de-filtre6 (**end-ssetq nom) rudimentaire6                        **end-ssetq
  :interpretation
    (let ((lien (env6-liaison nom segvarenv
                              'liaison6element )))
      (if lien
        (reussir6 e env
          (liaison6element-variable lien) alt)
        (filtre6-erreur
         "MATCH : (**END-SSETQ ~A) indefini !"
         nom ) ) )
```

Le filtre lispien et technique `**ssetq-terminal` ne se satisfait guère que d'expressions satisfaisant `listp`.

```
(de-filtre6 (**ssetq-terminal nom) rudimentaire6
  :interpretation                                                  **ssetq-terminal
    (if (variable6liee-p nom env)
      (filtre6-erreur
       "MATCH : La variable ~A est deja liee." nom )
      (if (listp e)
        (reussir6 e (ajouter6segment env nom e nil)
                  succ alt )
        (reprendre6 alt) ) )
```

Le filtre `**eval-append` est la concaténation d'un filtre de segment valué et d'un filtre quelconque.

```
(de-filtre6 (**eval-append nom filtre) rudimentaire6
  :interpretation                                                  **eval-append
    (let ((lien (env6-liaison nom env 'liaison6variable)))
      (if lien
        (multiple-value-bind (bool arg)
           (typecase lien
              (liaison6element
               (filtre6-erreur
                "MATCH : Variable ~A non segmentielle."
                nom ) )
              (liaison6segment
               (executer6verifier-2liste
                e
                (liaison6segment-debut-ref lien)
                (liaison6segment-fin-ref lien) ) ) )
```

```
          (if bool (filtrer6 arg filtre env succ alt
                             segvarenv timesvarenv )
              (reprendre6 alt) ) )
        (filtre6-erreur
         "MATCH : la variable ~A n'est pas liee a un segment."
         nom ) ) )
```

La vérification que l'expression `e` débute bien par la valeur de la variable *nom* est effectuée par la fonction de bibliothèque `executer6verifier-2liste` qui retourne un résultat multiple composé d'un booléen et du reste de l'expression non consommée.

executer6verifier-2liste

```
(defun executer6verifier-2liste (e debut fin)
   (if (and (consp debut) (not (eq debut fin)))
       (if (consp e)
           (if (equal (car e) (car debut))
               (executer6verifier-2liste (cdr e)
                          (cdr debut) fin )
               (values nil e) ) )
       (values t e) ) )
```

La répétition est introduite par `**labelled-times` et achevée par `**end-times`. Ce couple use de l'environnement `timesvarenv` pour leur communication privée.

**labelled-times

```
(de-filtre6 (**labelled-times nom filtre1 filtre2)
  rudimentaire6 :interpretation
    (if (extraire6 filtre1)
      (filtre6-erreur
       "MATCH : Affectations dans boucle ~S."
       (extraire6 filtre1) )
      (labels
        ((try (e env alt)
           (filtrer6 e filtre2 env succ
              (ou-sinon6
                (filtrer6 e filtre1 env
                   (et-puis6 (e1 env1 alt1)
                        (try e1 env1 alt1) )
                   alt
                   (initial6env)
                   (ajouter6boucle (initial6env)
                                   nom #'try ) ) )
                segvarenv
                timesvarenv ) ))
        (try e env alt) ) )
```

**end-times

```
(de-filtre6 (**end-times nom) rudimentaire6
  :interpretation
    (let ((lien (env6-liaison nom timesvarenv
                              'liaison6boucle )))
      (if lien
        (funcall (liaison6boucle-fonction lien) e env alt)
        (filtre6-erreur
         "Match : (**END-TIMES ~A) indefini !" nom) ) )
```

Enfin, der des der, le filtre technique qui accepte tout mais calcule au passage. Ce filtre est utile pour manipuler les continuations initiales de succès. Par exemple, `(match-lambda` *filtre forme*`)` se traduit en `(**and` *filtre* `(**success (return-from` *label forme*`)))` Si le filtre réussit, la *forme* de succès sera évaluée et l'échappement approprié (`return-from` *label*) sera exécuté. Si le filtrage échoue, le résultat est naturellement `nil`. Le filtre `**success` permet de réifier l'endroit où se situe le lancement d'un calcul extérieur au filtrage, puisqu'en Lisp, et donc de le manipuler comme un filtre.

Pour convertir les variables de filtrage en variables lexicales, on usera de **eval** qui se justifie ici, par le côté interprétatif de cette définition. Cette définition n'est toutefois qu'une approximation puisque **eval** évalue son argument non pas dans le contexte lexical courant mais dans un contexte nul. La génération de code corrigera ce défaut.

**success

```
(de-filtre6 (**success forme) rudimentaire6
  :interpretation
    (if
     (eval
      `(let
        ,(mapcar
           #'(lambda (lien)
               (etypecase lien
                 (liaison6element
                  `(,(liaison6-nom lien)
                     ',(liaison6element-variable lien) ) )
                 (liaison6segment
                  `(,(liaison6-nom lien)
                    (executer6oter
                      ',(liaison6segment-debut-ref lien)
                      ',(liaison6segment-fin-ref lien))))))
           env )
         ,forme ))
      (reussir6 e env succ alt)
      (reprendre6 alt) )
```

Voici donc achevée la sémantique des filtres rudimentaires. Cette définition pourrait à coup sûr se traduire en une sémantique dénotationnelle bourrée de petites lettres grecques. La forme Lisp de ces équations est cependant intéressante puisque le moteur ci-dessus exposé est exécutable, peut être testé et sera utilisé comme étalon vis-à-vis de la génération de code qui sera bientôt faite.

5.8 Génération de code

À la base de toute génération de code, reposent quelques idées préconçues sur la structure que doivent revêtir les résultats. Une implantation d'un langage tel que Lisp est, dans une large mesure, déterminée par les choix de représentations des données et par la structure de la pile d'évaluation. Nous donnerons donc pour chaque filtre rudimentaire sa traduction en Lisp. Cette traduction s'appuie sur un certain nombre de préjugés relatifs à l'efficacité du véritable compilateur Lisp qui digèrera le résultat. Les grands choix concernent l'environnement (comment sont mémorisées les valeurs des variables de filtrage) et l'invocation des succès (comment évaluer dans un contexte lexical correct). Plusieurs stratégies étaient possibles, ont été essayées puis rejetées. Illustrons la stratégie retenue sur un exemple. Soit la forme

```
(match-lambda (*or (?x) (?x ?x))
     `(x = ,x) )
```

La normalisation du filtre conduit à[4] :

```
(**and2 (**cons (**setq x (**sexp))
                (**or2 (**quote nil)
                       (**cons (**eval x) (**quote nil)) ) )
        (**success `(x = ,x)) )
```

Le code engendré sera :

```
(LAMBDA (G39)
  (BLOCK G40
    (FLET ((G42 (X) (RETURN-FROM G40     ;;fonction de succès
                       `(X = ,X) )))
```

[4] On notera en passant la mise en facteur du fragment "(??x ..."

```
    (LET (G41)                          ;;variable locale pour x
      (AND (CONSP G39)
           (PROGN
            (SETQ G41 (CAR G39))        ;;(**setq X ?-)
            (OR (AND (NULL (CDR G39)) ;;(**quote nil)
                     (G42 G41))         ;;Gagné !
                (AND (CONSP (CDR G39))
                     (EQUAL (CADR G39) G41) ;;(**eval X)
                     (NULL (CDDR G39)) ;;(**quote nil)
                     (G42 G41))))))))) ;;Caramba encore gagné!
```

Le code du filtrage proprement dit est enchâssé dans un *berceau* formé

- d'une forme `flet` introduisant toutes les fonctions de succès, chacune affublée d'une liste de variables conforme aux variables de filtrage valuées lors de son (ou ses) appel(s); il n'y en a usuellement qu'une seule mais plusieurs sont possibles,
- d'une forme `let` introduisant des variables locales et anonymes dans lesquelles seront sauvegardées les valeurs que peuvent prendre les variables de filtrage.

D'autres générations de code sont possibles mais celle-ci offre quelques avantages. Tous les succès sont condensés en un seul `flet` et sous la forme de fonctions. Chaque succès dans le filtrage est marqué par un appel à une de ces fonctions avec les valeurs des variables de filtrage valuées comme arguments. Toutes les variables sont introduites en une seule fois, en un seul `let`. Cette forme `let` est placée après la forme `flet` pour aider l'analyse des fermetures du compilateur Lisp sous-jacent. Les variables locales sont anonymes pour empêcher l'utilisation précoce des variables de filtres par les prédicats arguments de `*check`. Une amélioration consistera, lorsque la fonction de succès n'est invoquée qu'une unique fois, à la placer sous la forme d'un `let` à l'endroit correspondant. Par exemple, la fonction `nodup`, extraite de [Peyton Jones 86].

```
(def-match-function nodup
  ( ()            => ())
  ( (?x)          => `(,x))
  ( (?x ?x ???y) => (nodup `(,x . ,y)))
  ( (?x ???y)     => `(,x . ,(nodup y))) )
```

engendrera

```
(LAMBDA (G77)
   (BLOCK G76
      (LET (G78 G79 G80)
        (OR (AND (NULL G77) (RETURN-FROM G76 NIL))
            (AND (CONSP G77)
                 (PROGN (SETQ G80 (CAR G77))
                        (OR (AND (NULL (CDR G77))
                                 (LET ((X G80))
                                   (RETURN-FROM G76
                                     (LIST X) ))
                            (AND (CONSP (CDR G77))
                                 (EQUAL (CADR G77) G80)
                                 (LISTP (CDDR G77))
                                 (PROGN
                                  (SETQ G78 (CDDR G77))
                                  (LET ((Y G78)
                                        (X G80))
                                    (RETURN-FROM G76
                                      (NODUP (CONS X Y)))))
                            (AND (LISTP (CDR G77))
                                 (PROGN
                                  (SETQ G78 (CDR G77))
                                  (LET ((Y G78)
                                        (X G80))
```

```
(RETURN-FROM G76
   (CONS X (NODUP Y))
)))))))))))
```

Les filtres **and, **or et autre **not seront naturellement représentés par des and, or et not. Enfin les combinaisons de filtres que procurent *list, *cons ou *progn useront de if et progn pour être enchaînés. Il apparaîtra clairement que le filtre le plus compliqué à traduire est *times mais que **ssetq-append et **success sont les plus complexes à améliorer.

La génération de code est effectuée grâce à compiler6.

```
(defun compiler6 (e f env succ alt segvarenv timesvarenv)
  (let ((filtre (obtenir6filtre (car f))))
    (unless filtre
      (error "Je ne connais pas ~S." (car f)) )
    (let ((fn (rudimentaire6-compilation filtre)))
      (if fn (funcall fn e f env succ alt
                     segvarenv timesvarenv )
          (error "Le filtre ~A ne sait pas se compiler."
                 (car f) ) ) ) ) )
```

5.8.1 Les générateurs de base

La thèse de Neidl veut que la génération de code soit effectuée par modification des fonctions de bibliothèques. Rappelons que le but du compilateur est de traduire le filtre en un code exécutable. Là où l'interprétation du filtre exigeait une décision (introduite par if) il faudra produire du code qui, quand il sera exécuté, prendra la décision. L'écriture des méthodes de compilation des filtres rudimentaires est donc un peu comme l'écriture d'une macro où l'on doit engendrer un if plutôt que de l'exécuter lors de l'expansion (mais si l'on peut c'est mieux). Nous aurons donc des générateurs particuliers nommés si, et, ou, egal ... exprimant les comparaison et combinaisons de comparaisons qui seront à effectuer. Leur définition est immédiate :

```
(defun si (co th el)           `(if ,co ,th ,el))
(defun et (a b)                `(and ,a ,b))
(defun ou (a b)                `(or ,a ,b))
(defun egal (a b)              `(equal ,a ,b))
(defun un-consp (c)            `(consp ,c))
(defun le-car (a)              `(car ,a))
(defun le-cdr (a)              `(cdr ,a))
(defun faire (&rest args)      `(progn . ,args))
(defun e-faire (arg)           (apply #'faire arg))
(defun affecter (variable valeur) `(setq ,variable ,valeur))
(defun sechapper (label forme)
  `(return-from ,label ,forme) )
```

Pareillement devront être synthétisés des chemins d'accès aux divers constituants de l'expression scrutée : le-car et le-cdr traitent cela :

```
(defun le-car (a)              `(car ,a))
(defun le-cdr (a)              `(cdr ,a))
```

Enfin lorsque paraissait dans les méthodes d'interprétation un appel à reussir6 ou à reprendre6, il faudra maintenant comprendre qu'il faut engendrer le code correspondant au succès ou à l'échec.

Tous ces générateurs seront améliorés plus tard pour, d'une part, rendre le code engendré plus lisible et d'autre part supprimer quelques instructions inutiles. Ces améliorations seront programmées par filtrage aussi devons-nous attendre encore qu'il soit mûr !

5.8.2 La compilation des filtres d'éléments simples

La théorie veut donc que la méthode de compilation soit semblable à celle d'interprétation sauf que les if auront été remplacés par des si, les and par des et, etc. Quelques accents graves (backquote) de synthèse surgiront aussi çà et là.

`*sexp` est le plus simple de tous les filtres : il correspond au succès instantané et sera tout simplement

**sexp

```
(de-filtre6 (**sexp) rudimentaire6
  :compilation
    (reussir6 e env succ alt) )
```

Le filtre `*quote` compare son argument avec l'expression. Si la réponse est déjà connue dans l'environnement alors cette réponse est substituée directement en place.

**quote

```
(de-filtre6 (**quote expression) rudimentaire6
  :compilation
    (si (let ((lien (env6-liaison e env 'liaison6equal)))
          (if (and lien (equal expression
                               (liaison6equal-value lien) ))
            (liaison6equal-bool lien)
            (egal e '',expression) ) )
        (reussir6 e env succ alt)
        (reprendre6 alt) ) )
```

Le filtre `**check` est non moins trivial et se compile en :

**check

```
(de-filtre6 (**check predicat) rudimentaire6
  :compilation
    (si '(,predicat ,e)
        (reussir6 e env succ alt)
        (reprendre6 alt) ) )
```

Le lecteur est invité à comparer les méthodes d'interprétation et de compilation de ces trois premiers filtres. Il pourra ainsi constater le surprenant décalque.

5.8.3 Filtres booléens

Ils se décalquent assez bien de leur sémantique sauf que `filtrer6` devient `compiler6`. Voici la conjonction, parfaitement conforme à son modèle.

**and2

```
(de-filtre6 (**and2 filtre1 filtre2) rudimentaire6
  :compilation
    (compiler6 e filtre1 env
           (et-puis6 (e1 env1 alt1)
                (compiler6 e filtre2 env1 succ alt1
                                   segvarenv timesvarenv ) )
           alt segvarenv timesvarenv ) )
```

La disjonction est un peu modifiée. Nous avons pris le parti de partout coder l'échec par `nil`. Il faut donc non moins partout s'arranger pour que le retour de `nil` corresponde à l'évaluation du code relançant le filtrage. C'est possible et cela procure un autre avantage qui sera vu lors de la compilation de `**labelled-times`. Le filtre `**or2` sera donc naturellement compilé en une conjonction `or`, engendrée par `ou`.

**or2

```
(de-filtre6 (**or2 filtre1 filtre2) rudimentaire6
  :compilation
    (ou (compiler6 e filtre1 env succ ou-sinon6nil
                           segvarenv timesvarenv )
        (compiler6 e filtre2 env succ alt
                           segvarenv timesvarenv ) ) )
```

Du fait du choix associant l'échec et `nil` et de l'abandon de l'environnement résultant d'un `**not`, ce dernier sera compilé grâce à une négation `not` enveloppée dans une forme `let` qui introduit les variables locales du `**not`. Ces variables sont extraites grâce à `extraire6` qui retourne un environnement que l'on transforme pour allouer les variables de stockage correspondantes et que l'on fusionne avec l'environnement ambiant afin de pouvoir compiler le filtre argument de `**not`. La fonction `enchasser6variables` enveloppe le code dans une forme `let` s'il y a des variables à y placer.

```
(de-filtre6 (**not filtre) rudimentaire6                          **not
  :compilation
   (let* ((varenv (extraire6 filtre))
          (env (fusionner6env (transformer6env varenv) env))
          (code (si (compiler6 e filtre env
                              et-puis6t ou-sinon6nil
                              segvarenv timesvarenv )
                    (reprendre6 alt)
                    (reussir6 e env succ alt) )) )
     (enchasser6variables varenv code) ) )
```

```
(defun enchasser6variables (env code)                enchasser6variables
  (let ((vars (allouer6variables env)))
    (if vars `(let ,vars ,code)
        code ) ) )
```

Donnons un exemple, encore succinct, des ces précédents filtres. `(*or foo (*not bar))` sera compilé en :

```
(or (if (equal e 'foo) succès nil)
    (if (not (equal e 'bar))
        succès échec ) )
```

On se rend compte dans cet exemple, pourtant immédiat, que de nombreuses circonlocutions ont été engendrées que l'on s'attachera plus loin à faire s'évanouir. On pourra remarquer aussi que de multiples exemplaires de *succès* sont apparus, c'est lié à la stratégie de compilation mais est atténué par le choix que les succès seront invoqués comme des fonctions dont le code sera ainsi factorisé.

5.8.4 Extraction des variables affectées

Les variables locales affectées c'est-à-dire faisant l'objet d'un `**setq`, d'un `**ssetq-append` ou d'un `**ssetq-terminal` sont extraites de tout filtre grâce à `extraire6`.

```
(defun extraire6 (f)
  (let ((filtre (obtenir6filtre (car f))))
    (unless filtre
      (error "Je ne connnais pas ~S." (car f)) )
    (let ((fn (rudimentaire6-extraction filtre)))
      (if fn (funcall fn f)
         (error "Le filtre ~A ne sait pas extraire."
                (car f) ) ) ) ) )
```

Celle-ci met en œuvre les méthodes d'extraction propres à chacun des filtres et que voici en vrac :

```
(de-filtre6 (**sexp) rudimentaire6 :extraction                   **sexp
    (initial6env) )

(de-filtre6 (**quote expression) rudimentaire6                  **quote
  :extraction (initial6env) )

(de-filtre6 (**check predicat) rudimentaire6                    **check
  :extraction (initial6env) )

(de-filtre6 (**or2 filtre1 filtre2) rudimentaire6                 **or2
  :extraction (fusionner6env (extraire6 filtre1)
                             (extraire6 filtre2) ) )

(de-filtre6 (**and2 filtre1 filtre2) rudimentaire6               **and2
  :extraction (fusionner6env (extraire6 filtre1)
                             (extraire6 filtre2) ) )
```

****not**

```
(de-filtre6 (**not filtre) rudimentaire6
  :extraction (initial6env) )
```

****setq**

```
(de-filtre6 (**setq nom filtre) rudimentaire6
  :extraction
    (ajouter6element (extraire6 filtre) nom nil) )
```

****eval**

```
(de-filtre6 (**eval nom) rudimentaire6
  :extraction (initial6env) )
```

****cons**

```
(de-filtre6 (**cons filtre1 filtre2) rudimentaire6
  :extraction (fusionner6env (extraire6 filtre1)
                             (extraire6 filtre2) ) )
```

****ssetq-append**

```
(de-filtre6 (**ssetq-append nom filtre1 filtre2)
  rudimentaire6 :extraction
   (ajouter6segment
     (fusionner6env (extraire6 filtre1) (extraire6 filtre2))
     nom nil nil ) )
```

****end-ssetq**

```
(de-filtre6 (**end-ssetq nom) rudimentaire6
  :extraction (initial6env) )
```

****ssetq-terminal**

```
(de-filtre6 (**ssetq-terminal nom) rudimentaire6
  :extraction
    (ajouter6segment (initial6env) nom nil nil) )
```

****eval-append**

```
(de-filtre6 (**eval-afppend nom filtre) rudimentaire6
  :extraction
    (extraire6 filtre) )
```

****labelled-times**

```
(de-filtre6 (**labelled-times nom filtre1 filtre2)
  rudimentaire6 :extraction
   (fusionner6env (extraire6 filtre1) (extraire6 filtre2)) )
```

****end-times**

```
(de-filtre6 (**end-times nom) rudimentaire6
 :extraction (initial6env) )
```

****success**

```
(de-filtre6 (**success forme) rudimentaire6
  :extraction (initial6env) )
```

****struct**

```
(de-filtre6 (**struct type . champs) rudimentaire6
  :extraction
    (labels ((analyser (champs resultat)
                (if (consp champs)
                  (analyser (cddr champs)
                            (fusionner6env
                              (extraire6 (cadr champs))
                              resultat  ) )
                  resultat ) ))
      (analyser champs (initial6env)) ) )
```

La valeur retournée est un environnement comportant des liaisons de type `liaison6element` ou `liaison6segment` suivant la nature des variables découvertes.

5.8.5 Les paires pointées

Le filtre `**cons` s'occupe des paires pointées. On doit d'abord s'assurer que l'expression est un doublet pour ensuite filtrer ses `car` et `cdr`. Fidèle à sa sémantique, `**cons` se compile comme :

`**cons`

```
(de-filtre6 (**cons filtre1 filtre2) rudimentaire6
  :compilation
    (si (let ((lien (env6-liaison e env 'liaison6consp)))
          (if lien (liaison6consp-bool lien) (un-consp e)) )
        (compiler6 (le-car e) filtre1 env
            (et-puis6 (e1 env1 alt1)
              (compiler6 (le-cdr e) filtre2 env1 succ alt1
                         segvarenv timesvarenv ) )
            alt (initial6env) (initial6env) )
        (reprendre6 alt) ) )
```

On voit ici les premiers emplois significatifs de l'expression courante `e`. `e` n'est bien sûr pas l'expression mais une forme qui, à l'exécution, aura pour valeur l'expression courante. Les générateurs `le-car` et `le-cdr` changent l'expression courante en ses `car` et `cdr`.

Pour illustrer ces mécanismes où l'on ne parle jamais des objets directement mais par l'intermédiaire de formes qui, si elles étaient évaluées, ramèneraient ces valeurs, intéressons-nous au filtre :

```
(**cons (**or (**quote foo) (**quote bar)) ;;((*or foo bar) ???-)
        (**sexp) )
```

Supposons que `succ` soit `"Succès"` et que `alt` soit `"Échec"`. Alors le résultat de compilation sera :

```
(if (consp e)                  ;;**cons
  (or                          ;;**or
      (if (equal (car e) 'foo)     ;;**quote foo
          (if t                    ;;**sexp
              "Succès" nil )
          nil )
      (if (equal (car e) 'bar)     ;;**quote bar
          (if t                    ;;**sexp
              "Succès" nil )
          "Échec" ) )
  "Échec" )
```

Ce résultat, dont l'aspect extérieur sera affiné, est fait d'un grand nombre de morceaux composés dont la provenance est rappelée en commentaire.

L'autre filtre structurel est `**struct` dont le généré épouse la définition interprétative.

`**struct`

```
(de-filtre6 (**struct type . champs) rudimentaire6
  :compilation
    (flet ((concat (symba str symbb)
             (find-symbol (concatenate 'string
              (symbol-name symba) str (symbol-name symbb)))))
      (si `(subtypep (type-of ,e) ',type)
        (labels ((analyser (champs env succ alt)
                   (if (consp champs)
                     (compiler6
                      `(,(concat type "-" (car champs)) ,e)
                      (cadr champs) env
                      (et-puis6 (e1 env1 alt1)
                        (analyser (cddr champs)
                                  env1 succ alt1 ) )
                      alt (initial6env) (initial6env) )
                     (reussir6 e env succ alt) ) ))
           (analyser champs env succ alt) )
        (reprendre6 alt) ) ) )
```

Donnons un court exemple avec le filtre :

```
(*struct liaison6element :variable ?x :nom ?x)
```

qui engendrera

```
(LET (G63)
  (AND (SUBTYPEP (TYPE-OF G62) 'LIAISON6ELEMENT)
       (PROGN (SETQ G63 (LIAISON6ELEMENT-VARIABLE G62))
              (AND (EQUAL (LIAISON6ELEMENT-NOM G62) G63)
                   (LET ((X G63)) Succès)))))
```

5.8.6 Les variables de filtrage de type élément

Abordons les filtres `**setq` et `**eval` qui lient et usent des variables de filtrage de type élément. Ces variables de filtrage seront représentées par des variables dont le nom sera synthétisé par **`gensym`** de manière à éviter tout masquage de variable. Ces noms abstrus protégeront aussi le code engendré contre toute perversion que tenterait tout utilisateur indélicat par le biais de prédicat très particuliers qu'il soumettrait via `**check`. Le seul moyen légal d'accéder aux valeurs des variables de filtrage depuis Lisp est d'employer le pseudo-filtre `**success`.

Une variable de type élément sera représentée par une simple variable située dans le **`let`** englobant du berceau. Le filtre `**setq` l'affectera tandis que `**eval` l'évaluera. Comme il nous faut associer le nom de la variable de filtrage et le nom de la variable lexicale qui stocke sa valeur, on supposera trouver ce lien (matérialisé par une liaison de type `liaison6element`) dans l'environnement **`env`** grâce à la fonction `stockage`. Par contre, comme il nous faut aussi déterminer l'ensemble des variables de filtrage liées avant de lancer `**success`, il nous faut aussi enregistrer que la variable est liée dans **`env`**; ce sera le rôle d'un lien de type `liaison6variable`. Il n'est pas nécessaire d'enregistrer la nature de la variable puisqu'elle figure déjà dans le lien de stockage.

$\boxed{\texttt{**setq}}$

```
(de-filtre6 (**setq nom filtre) rudimentaire6
  :compilation
    (if (variable6liee-p nom env)
      (filtre6-erreur "MATCH: ~A est deja liee." nom)
      (let ((lien (stockage nom env)))
        (compiler6 e filtre env
          (et-puis6 (e1 env1 alt1)
           (faire
             (affecter (liaison6element-variable lien) e)
             (reussir6 e (ajouter6variable env1 nom)
                       succ alt1 ) ) )
          alt segvarenv timesvarenv ) ) ) )
```

$\boxed{\texttt{**eval}}$

```
(de-filtre6 (**eval nom) rudimentaire6
  :compilation
    (if (variable6liee-p nom env)
      (let ((lien (stockage nom env)))
        (if (subtypep (type-of lien) 'liaison6element)
          (si (egal e (liaison6element-variable lien))
              (reussir6 e env succ alt)
              (reprendre6 alt) )
          (filtre6-erreur "MATCH : ~A n'est pas element."
                          (liaison6alias-vrai-nom nom) ) ) )
      (filtre6-erreur "MATCH : ~A n'est pas liee." nom) ) )
```

Le filtre `(**cons (**setq x (**sexp)) (**cons (**eval x) (**quote nil)))`, version normalisée de `(?x ?x)`, sera donc compilé en :

```
(if (consp g12)     ;;g13 est le substitut de x
    (progn (setq g13 (car g12))
           (if (consp (cdr g12))
              (if (equal (car (cdr g12)) g13)
                 (if (equal (cdr (cdr g12)) 'nil)
```

```
                    "Succès" "Échec" )
                  "Échec" )
                "Échec" ) )
      "Échec" )
```

5.8.7 Les filtres segments

Les variables de segment apparaissent dans les filtres normalisés sous la forme d'une affectation réalisée par `**ssetq-append`, filtre dans lequel le premier argument s'achève par la marque syntaxique spéciale `**end-ssetq`. Prenons le filtre suivant où `*ssetq` est à `*setq` ce que les segments sont aux éléments.

```
(*list (*ssetq x (*or foo bar) (*check numberp)) ??x)
```

Ce filtre reconnaît `(foo 4 foo 4)` et `(bar 1 bar 1)` mais pas `(foo 3 bar 3)`. La version normalisée est :

```
(**SSETQ-APPEND X
                (**CONS (**OR2 (**QUOTE FOO) (**QUOTE BAR))
                        (**CONS (**CHECK NUMBERP)
                                (**END-SSETQ X)))
                (**EVAL-APPEND X (**QUOTE NIL)))
```

Les variables de segment seront représentées par deux variables lexicales, l'une désignant la tête du segment, la seconde désignant sa queue. Comme pour les variables de type élément, l'association liant la variable de type segment au couple de variables lexicales de stockage se trouvera dans l'environnement dans une liaison de type `liaison6segment` et sera récupérée par la fonction `stockage`. On notera, toujours dans l'environnement `env`, que l'on a affecté une variable grâce à `ajouter6variable`. Les vérifications usuelles concernant l'existence d'une éventuelle précédente liaison sont bien sûr effectuées avant toute autre chose.

**ssetq-append

```
(de-filtre6 (**ssetq-append nom filtre1 filtre2) rudimentaire6
  :compilation
    (if (variable6liee-p nom env)
      (filtre6-erreur "MATCH : ~A est deja liee." nom)
      (let* ((lien (stockage nom env))
             (debut (liaison6segment-debut-ref lien))
             (fin (liaison6segment-fin-ref lien))
             (nsegvarenv
              (ajouter6ssetq (initial6env) nom
                 (et-puis6 (e1 env1 alt1)
                   (if (equal filtre2 '(**quote nil))
                     (compiler6 e1 filtre2
                       (ajouter6variable-terminale env1 nom)
                       succ alt1 segvarenv timesvarenv )
                     (faire (affecter fin e1)
                            (compiler6 e1 filtre2
                             (ajouter6variable env1 nom)
                             succ alt1
                             segvarenv timesvarenv ) ) ) ) ))
             (code (compiler6 e filtre1
                     env
                     (et-puis6 (e1 env1 alt1)
                       (filtre6-erreur
                        "MATCH : ~A est inachevee."
                        nom ) )
                     alt nsegvarenv (initial6env) )) )
        (faire
         (affecter debut e)
         `(flet
           ,(let ((mem (env6-liaison nom nsegvarenv
                                     'liaison6ssetq )))
```

```
                (mapcar #'(lambda (occ)
                           `(,(occurrence6ssetq-nom occ)
                             (,(occurrence6ssetq-variable occ))
                             ,(occurrence6ssetq-code occ) ) )
                        (liaison6ssetq-occurrences mem) ) )
             ,code ) ) ) ) )
```

La fin du segment est signalée par la marque syntaxique **`**end-ssetq`**. D'ailleurs si on ne la trouve pas, la continuation menant à l'erreur **`"MATCH : l'affectation a ~A est inachevee."`** sera invoquée. Le filtre **`**end-ssetq`** relance le filtrage qui suit, c'est-à-dire le second filtre argument de **`**ssetq-append`**. Ce filtrage, ou plutôt cette génération de code est archivée dans l'environnement **`segvarenv`** qui désigne la variable de type segment en cours de détermination.

`**end-ssetq`

```
(de-filtre6 (**end-ssetq nom) rudimentaire6
  (let ((lien (env6-liaison nom segvarenv 'liaison6ssetq)))
    (if lien
      (let ((occ (find env (liaison6ssetq-occurrences lien)
                        :test #'comparer6env-p
                        :key #'occurrence6ssetq-env )) )
        (if occ (progn (incf (occurrence6ssetq-nombre occ))
                        `(,(occurrence6ssetq-nom occ) ,e) )
            (let ((g (gensym))
                  (arg (gensym))
                  (code 'ignore) )
            (setq code
                  (reussir6 arg env
                       (liaison6ssetq-succ lien) alt ) )
            (setf (liaison6ssetq-occurrences lien)
                  (cons (make-occurrence6ssetq
                          :env env :nom g :nombre 1
                          :variable arg :code code )
                         (liaison6ssetq-occurrences lien) ) )
            `(,g ,e) ) ) )
      (filtre6-erreur
        "MATCH : (**END-SSETQ ~A) indefini !" nom ) ) ) )
```

Quelques explications sont nécessaires. Lorsque des alternatives (`**or2`) sont présentes dans le premier filtre de **`**ssetq-append`**, de nombreuses occurrences de **`**end-ssetq`** sont à craindre. Une technique simple pour compiler ces derniers est d'insérer à leur place le code correspondant au second filtre de **`**ssetq-append`**. Il faut alors s'attendre à une inflation de la taille du code engendré. Ainsi le filtre précédent pris en exemple :

```
(**SSETQ-APPEND X
                (**CONS (**OR2 (**QUOTE FOO) (**QUOTE BAR))
                        (**CONS (**CHECK NUMBERP)
                                (**END-SSETQ X)))
                (**EVAL-APPEND X (**QUOTE NIL)))
```

engendrerait-il, selon cette technique,

```
(LAMBDA (G64)
  (BLOCK G63
    (FLET ((G71 (X) (RETURN-FROM G63 Succès)))
      (LET (G65 G66)
        (SETQ G65 G64)
        (AND (CONSP G64)
             (OR (AND (EQ (CAR G64) 'FOO)
                      (CONSP (CDR G64))     ;;partie 1
                      (NUMBERP (CADR G64)) ;;isomorphe
                      (PROGN
                       (SETQ G66 (CDDR G64))
```

```
                (MULTIPLE-VALUE-BIND (G68 G67)
                  (EXECUTER6VERIFIER-2LISTE
                     (CDDR G64) G65 G66 )
                 (AND G68 (NULL G67)
                  (G71 (EXECUTER6OTER G65 G66))))))
          (AND (EQ (CAR G64) 'BAR)
               (CONSP (CDR G64))     ;;partie 2
               (NUMBERP (CADR G64)) ;;isomorphe
               (PROGN
                (SETQ G66 (CDDR G64))
                (MULTIPLE-VALUE-BIND (G73 G72)
                  (EXECUTER6VERIFIER-2LISTE
                     (CDDR G64) G65 G66 )
                 (AND G73
                      (NULL G72)
                      (G71 (EXECUTER6OTER G65 G66))
                       )))))))))))
```

La technique ici programmée consiste à factoriser le code commun en le transformant en une fonction que l'on placera dans un `flet` englobant. Ainsi obtiendra-t'on toujours avec le précédent exemple :

```
(LAMBDA (G29)
  (BLOCK G28
    (LET (G30 G31)
      (SETQ G30 G29)
      (FLET ((G32 (G33)
               (SETQ G31 G33)
               (MULTIPLE-VALUE-BIND (G35 G34)
                   (EXECUTER6VERIFIER-2LISTE G33 G30 G31)
                (AND G35 (NULL G34)
                     (LET ((X (EXECUTER6OTER G30 G31)))
                       (RETURN-FROM :G26428 Succès))))))
       (AND (CONSP G29)
            (OR (AND (EQ (CAR G29) 'FOO)
                     (CONSP (CDR G29))
                     (NUMBERP (CADR G29))
                     (G32 (CDDR G29)))
                (AND (EQ (CAR G29) 'BAR)
                     (CONSP (CDR G29))
                     (NUMBERP (CADR G29))
                     (G32 (CDDR G29))))))))))
```

Quelques subtilités existent au cas (interdit par la normalisation) où les `**end-ssetq` seraient dans des contextes lexicaux différents comme dans le filtre

```
(**ssetq-append X (**cons (**or2 (**setq Y (**quote foo))
                                 (**quote bar) )
                          (**end-ssetq x) )
                 (**quote nil) )
```

Ce filtre reconnaît `(foo)` et `(bar)` mais dans le premier des cas lie `X` et `Y` tandis qu'il ne lie que `X` dans le second des cas. Cet exemple est quelque peu un cas d'école car s'il peut être permissible d'affecter une variable dans une branche de `**or2`, il n'est plus envisageable d'utiliser cette même variable. En effet, remplacer `(**quote nil)` par `(**setq Y (**sexp))` réaffecterait `Y` dans une des branches de l'alternative, ce qui est une erreur; remplacer par `(**eval Y)` conduirait en revanche à l'erreur "variable indéfinie" dans l'autre branche de l'alternative. Quoiqu'il en soit et par pure coquetterie programmative, voici le code engendré par l'étrange filtre précédent :

```
(LAMBDA (G86)
  (BLOCK G85
    (FLET ((G94 (X)
```

```
            (RETURN-FROM G85 Succès avec X lié))
          (G92 (X Y)
            (RETURN-FROM G85 Succès avec X et Y liés)))
     (LET (G87 G88 G89)
        (SETQ G87 G86)
        (AND (CONSP G86)
             (OR (AND (EQ (CAR G86) 'FOO)
                      (PROGN (SETQ G89 (CAR G86))
                             (AND (NULL (CDR G86))
                                  (G92 G87 G89))))
                 (AND (EQ (CAR G86) 'BAR)
                      (NULL (CDR G86))
                      (G94 G87))))))))
```

On y reconnaît les deux possibilités de succès avec `Y` liée ou pas. Cet effet est réalisé comme suit : chaque occurrence de `**end-ssetq` est classée avec son environnement lexical associé. Lorsque toutes les occurrences sont recensées, on engendre autant de fonctions qu'il y a d'environnements lexicaux différents.

Une dernière amélioration reste à dévoiler. Les variables segmentielles sont repérées par un couple de variables désignant sa tête et sa terminaison. En cas de succès le segment est construit grâce à la fonction de bibliothèque `executer6oter`. Lorsque le segment est terminal, c'est-à-dire s'achève par `nil`, il est inutile de reconstruire sa valeur puisque l'expression est égale au segment désiré. On a donc introduit la notion de variable terminale qui représente un segment de liste repéré par sa tête seule. Il faut enrichir l'environnement pour l'indiquer car l'obtention de la valeur du segment est particulière : c'est le but de la fonction `ajouter6variable-terminale`.

L'autre moyen d'affecter une variable de segment est `**ssetq-terminal`. Celle-ci se compile simplement.

`**ssetq-terminal`

```
(de-filtre6 (**ssetq-terminal nom) rudimentaire6
  :compilation
    (if (variable6liee-p nom env)
      (filtre6-erreur "MATCH : ~A est deja liee." nom)
      (let* ((lien (stockage nom env))
             (debut (liaison6segment-debut-ref lien)) )
        (si '(listp ,e)
            (faire
              (affecter debut e)
              (reussir6 e
                (ajouter6variable-presque-terminale env nom)
                succ alt ) )
            (reprendre6 alt) ) ) ) )
```

Tout comme les variables segmentielles terminales, ces variables que l'on qualifiera de "presque terminales" seront repérées par leur tête seule, ce que l'on indiquera dans l'environnement par `ajouter6variable-presque-terminale`.

On peut utiliser une variable de segment valuée grâce au filtre `**eval-append`. La comparaison demandant un code constant un peu trop important pour être intégralement recopié chaque fois qu'on le nécessite, on en fera bien sûr une fonction de bibliothèque : `executer6verifier-2liste` déjà vue page 73.

`**eval-append`

```
(de-filtre6 (**eval-append nom filtre) rudimentaire6
    (if (variable6liee-p nom env)
      (let ((lien (stockage nom env)))
        (if (subtypep (type-of lien) 'liaison6segment)
          (let ((arg (gensym))
                (bool (gensym)) ) ;pour ne rien masquer !!!
            '(multiple-value-bind (,bool ,arg)
               ,(cond
```

```
                ((variable6presque-terminale-p nom env)
                 (format
                   *show-match-expansion-output-stream*
                   "~&MATCH : ~S presque terminale est ~
                     utilisee pour sa valeur !" nom )
                 `(executer6verifier-exp
                   ,e
                   ,(liaison6segment-debut-ref lien) ) )
                ((variable6terminale-p nom env)
                 `(executer6verifier-exp
                   ,e
                   ,(liaison6segment-debut-ref lien) ) )
                (t `(executer6verifier-2liste
                     ,e
                     ,(liaison6segment-debut-ref lien)
                     ,(liaison6segment-fin-ref lien) ) ) )
           ,(si bool (compiler6 arg filtre env succ alt
                                segvarenv timesvarenv )
                     (reprendre6 alt) ) ) )
         (filtre6-erreur "MATCH : ~A non segment."
                         nom ) ) )
      (filtre6-erreur "MATCH : ~A n'est pas liee." nom) ) )
```

Ainsi s'achève le traitement des variables de type segment.

5.8.8 Les répétitions

Les répétitions apparaissent sous forme normalisée semblablement aux variables de segment. Une marque syntaxique fixe l'endroit où peut avoir lieu la répétition. Prenons par exemple le filtre `(*any)` dont on sait qu'il n'est autre que `(*times (*sexp))`. Sa forme normalisée est

```
(**LABELLED-TIMES G93
                  (**CONS (**SEXP)
                          (**END-TIMES G93))
                  ... )
```

Les répétitions sont les plus difficiles à compiler. Fort heureusement les quelques décisions déjà prises aident à le traiter convenablement. La difficulté est que le filtre terminal `filtre2` ne doit être compilé qu'une seule et unique fois et non une infinité de fois. Or le premier filtre `filtre1` peut induire des liaisons. Prenons par exemple le filtre

```
(*list (*times ?x))
```

Cet innocent filtre est en fait équivalent à (et c'est le résultat de sa normalisation) :

```
(**OR2 (**QUOTE NIL)
       (**CONS (**SETQ X (**SEXP))
               (**LABELLED-TIMES G75
                      (**CONS (**EVAL X) (**END-TIMES G75))
                      (**QUOTE NIL) ) ) )
```

C'est-à-dire à :

```
(*or () (*list ?x (*times (*eval x))))
```

Dans ces deux filtres équivalents, deux phénomènes sont apparus. Le premier est que la variable `x` peut ne pas être liée par `*times` et induit donc une disjonction déséquilibrée. Le second phénomène est que `*times` ne peut souffrir comme argument un filtre qui contient une affectation puisque, comme ce filtre peut être amené à être répété, une réaffectation serait à craindre, ce qui est interdit. Il faut donc que le filtre argument de `*times` ne comporte pas d'affectation, en conséquence tous les appels à la marque syntaxique `**end-times` sont donc tous effectués dans un même environnement et une même continuation d'échec (`nil`). Tous les appels récursifs peuvent donc tous être factorisés en une même fonction.

Voici donc comment se compile `**labelled-times`.

**labelled-times

```
(de-filtre6 (**labelled-times nom filtre1 filtre2)
  rudimentaire6 :compilation
    (if (extraire6 filtre1)
      (filtre6-erreur "MATCH : Affectations dans ~S."
                      (extraire6 filtre1) )
      (let ((try (gensym))
            (ge (gensym)) )
        `(labels
          ((,try (,ge)
             ,(ou (compiler6 ge filtre2 env succ
                    ou-sinon6nil segvarenv timesvarenv )
                  (compiler6 ge filtre1 env
                   (et-puis6 (e1 env1 alt1)
                     (filtre6-erreur
                       "MATCH : Repetition en cours !" ) )
                   alt (initial6env)
                   (ajouter6boucle (initial6env)
                                   nom try ) ) ) ) ) )
         (,try ,e) ) ) ) )
```

Comme pour le filtre **ssetq-append, la continuation un peu bizarre de succès qui apparaît ci-dessus traque l'éventuelle absence (erronée) d'un **end-times approprié. La véritable continuation de succès est mise en œuvre par **end-times.

**end-times

```
(de-filtre6 (**end-times nom) rudimentaire6
  :compilation
    (let ((lien (env6-liaison nom timesvarenv
                              'liaison6boucle )))
      (if lien `(,(liaison6boucle-fonction lien) ,e)
          (filtre6-erreur
           "MATCH : (**END-TIMES ~A) indefini !" nom) ) ) )
```

Donnons un exemple de ces deux filtres à première vue indigeste. Soit le filtre

```
(??- ?x foo ???-)
```

dont la forme normalisée est :

```
(**LABELLED-TIMES G46
                  (**CONS (**SEXP) (**END-TIMES G46))
                  (**CONS (**SETQ X (**SEXP))
                          (**CONS (**QUOTE FOO) (**SEXP))))
```

On remarquera que la queue inimportante de la liste est non pas spécifiée par "??-" mais bien par "???-". La différence est que dans le premier des cas on doit vérifier que l'expression est bien une liste, une vraie liste terminée par nil, alors que l'on accepte tout dans le second cas sans autre forme de procès. Le code engendré est

```
(LAMBDA (G02)
  (BLOCK G00
    (LET (G03)
      (LABELS ((G04 (G05)
                 (AND (CONSP G05)
                      (OR (PROGN (SETQ G03 (CAR G05))
                                 (AND (CONSP (CDR G05))
                                      (EQ (CADR G05) 'FOO)
                                      (LET ((X G03))
                                        (RETURN-FROM G00
                                          "Succès" ))))
                          (G04 (CDR G05))))))
        (G04 G02))))))
```

La fonction locale *try* est récursive terminale ce qui garantit qu'un bon compilateur saurait la compiler de façon itérative. Un filtre tel que `**any` est alors compilé en quelques instructions seulement : quelque chose comme

```
(prog ((G26 e))
   success
        (,success ,gr)
   retry
       (if (consp G26) (go cdr))
       (return nil)                  ;;Échec
   cdr (setq G26 (cdr G26))
       (go success) )
```

Exercice 5.18 : Si votre compilateur favori ne dérécursive pas le généré de `**labelled-times`, concoctez une nouvelle méthode de compilation usant des ignobles `prog` et autres `go`.

5.8.9 L'invocation des succès

Le pseudo-filtre `**success` sert à indiquer les endroits où le filtrage a définitivement réussi. Marquer explicitement ces endroits permet de les manipuler convenablement. On verra par exemple que quelques puissent être les filtres ϕ_1, ϕ_2 et ϕ_3, on peut toujours transformer

`(**and2 (**or2` ϕ_1 ϕ_2`)` ϕ_3`)`

en

`(**or2 (**and2` ϕ_1 ϕ_3`) (**and2` ϕ_2 ϕ_3`))`

Une des difficultés de `**success` tient à sa définition qui est d'évaluer une forme Lisp dans le contexte instantané du filtrage. Il faut donc tenir à jour l'environnement de manière à pouvoir synthétiser convenablement la transformation entre variables de filtrage et variables lexicales. La synthèse de l'environnement lexical à partir de l'environnement de filtrage est effectuée lors de l'appel au code de succès qui sera transformé en une λ-expression dont les variables auront pour nom les noms des variables de filtrage valuées.

La plus simple des traductions de `(**success` *code*`)` est

`(let ((` *variable accès* `) ...)` *code*`)`

où *variable* représente les noms des variables de filtrage valuées tandis que *accès* représente les formes les calculant. Ces formes sont soit la simple nomination d'une variable de stockage dans le cas d'un élément ou d'un segment terminal soit un appel à `executer6oter` dans le cas d'un segment.

On l'a vu sur les nombreux exemples précédents, de nombreuses formes de succès peuvent apparaître, aussi serait-il ridicule de ne pas tenter de les factoriser. Toutes les occurrences de `**success` sont donc engrangées dans une structure (`liaison6succes`) située dans l'environnement et accessible par `env6-succes`. Elles sont ensuite classées par forme, dans des structures de type `succes6` et au sein de chaque forme par environnement, dans des structures `occurrence6`. Quelques compteurs y figurent, comptabilisant les occurrences. Un succès qui ne survient qu'en un seul endroit sera naturellement traduit en une forme `let` et non par un appel fonctionnel. Cette amélioration permet de ne pas trop présumer du compilateur sous-jacent.

La difficulté est maintenant qu'on ne sait plus traduire à la volée les filtres `**success` puisqu'il faut attendre d'avoir parcouru tout le filtre pour savoir comment les traduire. Plutôt que d'opérer de délicats actes chirurgicaux au sein du code engendré, nous avons opté pour des macros locales dont les définitions ne seront données qu'après coup. Donnons quelques exemples mais rappelons auparavant que `(match-lambda` *filtre code*`)` correspond au filtre `(*and` *filtre* `(*success` *code*`))`.

`(match-lambda foo 1)`

Un seul succès donc une insertion directe du code de succès à l'endroit de succès.

```
(LAMBDA (G72)
   (BLOCK G71
      (MACROLET ((G73 NIL '(RETURN-FROM G71 1)))
         (AND (EQ G72 'FOO) (G73)))))
```

`(match-lambda (?x ?x) 2)`

Un seul succès donc une insertion directe. Comme l'environnement n'est pas nul, une forme `let` est engendrée.

```
(LAMBDA (G81)
  (BLOCK G80
    (MACROLET
      ((G83 NIL '(LET ((X G82)) (RETURN-FROM G80 2))))
     (LET (G82)
       (AND (CONSP G81)
            (PROGN (SETQ G82 (CAR G81))
                   (AND (CONSP (CDR G81))
                        (EQUAL (CADR G81) G82)
                        (NULL (CDDR G81))
                        (G83))))))))
```

`(match-lambda (*or (?x ?y)(?y ?x)) 3)`

Deux endroits de succès mais dans des environnements compatibles.

```
(LAMBDA (G91)
  (BLOCK G90
    (FLET ((G96 (Y X) (RETURN-FROM G90 3)))
      (MACROLET ((G94 NIL '(G96 G93 G92)))
        (LET (G92 G93)
          (AND (CONSP G91)
               (OR (PROGN
                     (SETQ G92 (CAR G91))
                     (AND (CONSP (CDR G91))
                          (PROGN
                           (SETQ G93 (CADR G91))
                           (AND (NULL (CDDR G91))
                                (G94)))))
                   (PROGN
                     (SETQ G93 (CAR G91))
                     (AND (CONSP (CDR G91))
                          (PROGN
                           (SETQ G92 (CADR G91))
                           (AND (NULL (CDDR G91))
                                (G94))))))))))))
```

`(match-case-lambda ((?x ?x) 4) (?y 5))`

Deux succès différents.

```
(LAMBDA (G07)
  (BLOCK G06
     (MACROLET
         ((G13 NIL '(LET ((Y G09))
                         (RETURN-FROM G04 5) ))
          (G10 NIL '(LET ((X G08))
                         (RETURN-FROM G04 4) )) )
       (LET (G08 G09)
        (OR (AND (CONSP G07)
                 (PROGN (SETQ G08 (CAR G07))
                        (AND (CONSP (CDR G07))
                             (EQUAL (CADR G07) G08)
                             (NULL (CDDR G07))
                             (G10))))
            (PROGN (SETQ G09 G07)
                   (G13)))))))
```

Voici donc les fonctions compilant les succès :

**success

```
(de-filtre6 (**success forme) rudimentaire6
  :compilation
    (let* ((succes (env6-succes env))
           (bool (and (consp forme)
                      (eq (car forme) 'return-from) ))
           (blist (find forme (liaison6succes-alist succes)
                        :test #'eq :key #'succes6-forme )) )
      (if blist
        (let ((s (find env (succes6-occurrences blist)
                       :test #'comparer6env-p
                       :key #'occurrence6-env )))
          (incf (succes6-nombre-total blist))
          (if s (progn (incf (occurrence6-nombre s))
                       (if bool `(,(occurrence6-nom s))
                         (si `(,(occurrence6-nom s))
                             (reussir6 e env succ alt)
                             (reprendre6 alt) ) ) )
              (let ((g (gensym)))
                (setf (succes6-occurrences blist)
                      (cons (make-occurrence6
                             :env env :nom g
                             :nom-fonction (gensym)
                             :nombre 1)
                            (succes6-occurrences blist) ) )
                (if bool `(,g)
                    (si `(,g)
                        (reussir6 e env succ alt)
                        (reprendre6 alt) ) ) ) ) ) )
        (let ((g (gensym)))
          (setf (liaison6succes-alist succes)
                (cons (make-succes6 :forme forme
                       :nom (gensym) :nombre-total 1
                       :occurrences
                         (list (make-occurrence6
                                :env env :nom g
                                :nom-fonction (gensym)
                                :nombre 1 )) )
                      (liaison6succes-alist succes) ) )
          (if bool `(,g)
              (si `(,g)
                  (reussir6 e env succ alt)
                  (reprendre6 alt) ) ) ) ) ) )
```

enchasser6succes

```
(defun enchasser6succes (env code)
  (let* ((alist (liaison6succes-alist (env6-succes env)))
         (fns (allouer6succes alist))
         (macros (allouer6macro-succes alist)) )
    (if fns
      `(flet ,fns (macrolet ,macros ,code))
      `(macrolet ,macros ,code) ) ) )
```

```
;;; Détermine les fonctions à engendrer. Si une forme de succès
;;; est appelée plus d'une fois alors elle est fonctionnalisée.
```

allouer6succes

```
(defun allouer6succes (alist)
  (mapcan
   #'(lambda (blist)
      (if (= 1 (succes6-nombre-total blist))
        nil
```

```
      (mapcar #'(lambda (occ)
                  `(,(occurrence6-nom-fonction occ)
                    ,(env6-variables (occurrence6-env occ))
                    ,(succes6-forme blist) ) )
              (succes6-occurrences blist) ) ) )
   alist ) )

;;; Engendre l'expansion des macros. Si une forme n'est appelée qu'une
;;; unique fois, on l'insère sous la forme d'un let.
(defun allouer6macro-succes (alist)
  (mapcan
   #'(lambda (blist)
      (if (= 1 (succes6-nombre-total blist))
        (let* ((occ (car (succes6-occurrences blist)))
               (bindings
                (mapcar
                 #'(lambda (var)
                     `(,var
                       ,(engendrer6acces
                         (stockage var
                                   (occurrence6-env occ) )
                       (occurrence6-env occ) )) )
                 (env6-variables (occurrence6-env occ)) )) )
          (setq bindings
                (mapcan #'(lambda (bind)
                            (if (eq (car bind) (cadr bind))
                              nil (list bind) ) )
                        bindings ) )
          `((,(occurrence6-nom occ) ()
             ,(if (consp vars)
                `'(let ,vars ,(succes6-forme blist))
                `',(succes6-forme blist) ) )) )
        (mapcar
         #'(lambda (occ)
             `(,(occurrence6-nom occ) ()
               ',(engendrer6succes (occurrence6-env occ)
                       (occurrence6-nom-fonction occ)
                       (occurrence6-env occ) ) ) )
          (succes6-occurrences blist) ) ) )
   alist ) )
```

allouer6macro-succes

Résumé

Le compilateur traduit un filtre en un code Lisp exécutable. Diverses stratégies sont possibles suivant les moyens linguistiques que l'on s'autorise à mettre en œuvre. Ici le compilateur a fait le choix de confiner les expressions de succès dans des fonctions dont les variables ont pour nom les noms des variables de filtrage : ainsi est réalisée la traduction entre variable de filtrage et variable lexicale.

Un autre choix est de toujours coder l'échec par `nil` et donc d'adopter une architecture où la continuation d'échec est toujours en pile et relancé par des formes `or`.

Un dernier choix est de concentrer toutes les variables de travail nécessaires à l'accomplissement du filtrage en un seul endroit, dans une forme `let` initiale du berceau qui enchâsse le code engendré tout entier.

Enfin on remarquera la persistance de l'architecture du moteur à filtrer ici devenu moteur à compiler.

Exercice 5.19 : On aura remarqué l'allocation d'une nouvelle variable booléenne à chaque compilation de `**eval-append`. Modifier le code présenté afin de les fondre toutes en une unique variable allouée dans le berceau.

5.9 Normalisation

La première phase du compilateur traduit un filtre en un filtre équivalent mais exprimé dans un sous-ensemble particulièrement réduit. Cette phase n'est pas à proprement parler une "normalisation" puisque il n'existe pas une unique forme normalisée. Nous profiterons donc de cette traduction pour effectuer quelques améliorations sur le filtre afin d'éviter des branches mortes, des filtrages redondants etc. Cette normalisation s'effectue sous la forme d'un parcours du filtre original et sa traduction sera faite au cours de son parcours. Cette traduction est encore et bien sûr un avatar du moteur à filtrer. Il s'exprime sous la forme maintenant bien connue du couple fondamental **`normaliser6-element`** et **`normaliser6-segment`** et d'une fonction d'interface **`normaliser6`**. Les macro-filtres sont expansés et les filtres implicites **`*quote`** et **`*cons`** ajoutés.

normaliser6-element

```
(defun normaliser6-element (f env succ)
  (if (filtre6type-p f 'macro-element6)
    (normaliser6-element
      (expanser6 (obtenir6filtre (car f)) f) env succ )
    (if (filtre6type-p f 'element6)
      (normaliser6 (obtenir6filtre (car f)) f env succ)
      (if (filtre6type-p f 'rudimentaire6)
        (retourner6 f env succ)
        (if (atom f)
          (normaliser6-element `(*quote ,f) env succ)
          (normaliser6-segment f env succ) ) ) ) ) )
```

normaliser6-segment

```
(defun normaliser6-segment (f env succ)
  (if (consp f)
   (if (filtre6type-p (car f) 'macro-segment6)
     (normaliser6-segment
       (expanser6 (obtenir6filtre (caar f)) f) env succ )
     (if (filtre6type-p (car f) 'segment6)
       (normaliser6 (obtenir6filtre (caar f)) f env succ)
       ; Traitement par défaut :
       (normaliser6-element
        `(*cons ,(car f) (*list . ,(cdr f))) env succ) ) )
   (normaliser6-element `(*quote ,f) env succ) ) )
```

normaliser6

```
(defun normaliser6 (filtre f env succ)
  (let ((fn (executable6-normalisation filtre)))
    (if fn (funcall fn f env succ)
        (retourner6 f env succ) ) ) )
```

Le moteur à normaliser ne comporte que trois variables: **`f`** le filtre à normaliser, **`env`** l'environnement tenant compte des liaisons et de la nature des variables de filtre, enfin **`succ`** qui est la continuation de succès qui détermine ce que deviendra le filtre traduit et l'environnement résultant. Il n'y a pas de continuation d'échec car la traduction s'effectue sans à-coups, les erreurs étant signalées par **`filtrer6-erreur`**. Le retour à la continuation s'effectue par **`retourner6`** tandis que celles-ci seront construites par **`et-encore6`**.

et-encore6

```
(defmacro et-encore6 (variables . filtrage)
  `#'(lambda ,variables . ,filtrage) )
```

retourner6

```
(defun retourner6 (f env succ)
  (funcall succ (ameliorer6-element f env) env) )
```

Ces fonctions ressemblent, au nom près, à leurs équivalents **`et-puis6`** et **`reussir6`**. La continuation initiale de succès, telle qu'elle sera fournie par les macros d'interface **`match-lambda`** ou **`match-all-lambda`**, sera de fournir le filtre normalisé et un environnement dans lequel on pourra trouver le nom et la nature de toutes les variables nécessaires au filtrage.

Il ne reste plus qu'à passer en revue tous les schémas de normalisation des filtres. Ceux-ci seront classés en quatre catégories : macro-élément et macro-segment, ou élément et segment. Compte tenu de la définition du jeu réduit, le nombre de macro-filtres a singulièrement augmenté. Seuls sont restés de type segment ou élément ceux qui avaient à inspecter l'environnement pour déterminer leur traduction.

5.9.1 Les filtres simples

Ce ne sont plus que des macro-filtres à traduction immédiate.

```
(de-filtre6 (*sexp) macro-element6 :expansion                    *sexp
    `(**sexp) )

(de-filtre6 (*quote expression) macro-element6                   *quote
  :expansion `(**quote ,expression) )

(de-filtre6 (*check predicat) macro-element6                     *check
  :expansion `(**check ,predicat) )
```

Rien de plus n'est à dire à leur sujet, ils ne sont que les strictes démarques de leurs équivalents du jeu réduit.

5.9.2 Les filtres booléens

Le premier que nous traiterons sera le plus simple : `*not`.

```
(de-filtre6 (*not filtre) element6 :normalisation                *not
    (normaliser6-element filtre env
       (et-encore6 (f env1)
           (retourner6 `(**not ,f) env succ) ) ) )
```

On remarque aisément l'abandon de l'environnement interne au `*not` conformément à sa définition interprétative.

Le travail lié aux filtres `*or` et `*and` est plus complexe. Il faut en effet grouper deux par deux les filtres puisque `**or2` et `**and2` sont binaires. De multiples choix sont possibles mais certains sont meilleurs que d'autres. Étudions, par exemple, le filtre `(*or (?x) (?x ?x) (?x ?x ?x))`. La bonne façon, intuitive, de le normaliser est de le transformer en

```
(**or2 (**cons (**setq x (**sexp)) (**quote nil))
       (**or2 (**cons (**setq x (**sexp))
                      (**cons (**eval x) (**quote nil)) )
              (**cons (**setq x (**sexp))
                      (**cons (**eval x)
                              (**cons (**eval x)
                                      (**quote nil))))))
```

Car ainsi les termes qui se rapprochent le plus sont les plus à même d'être combinés en

```
(**cons (**setq x (**sexp))
        (**or2 (**quote nil))
               (**cons (**eval x)
                       (**or2 (**quote nil)
                              (**cons (**eval x)
                                      (**quote nil))))))
```

Cette nécessité de regroupement complique fortement le traitement des conjonctions et disjonctions.

Il faut de plus, dans le cas de `**or2`, s'enquérir des environnements produits par chacune des branches afin de voir si elles sont bien équilibrées et produire un message d'erreur dans le cas contraire. La fonction `normaliser6or` s'occupe de cette vérification tandis que le regroupement est l'œuvre de `normaliser6groupe` qui sera ainsi commune à `normaliser6or` et `normaliser6and`. Le regroupement est fondé sur l'identité des filtres de tête : les `**quote` avec les `**quote` et les `**cons` avec les `**cons` (sic). Lorsque les regroupements dépassent deux unités, on approfondit si possible leurs différences avec leurs filtres de premier argument (voir la fonction `approfondir6difference`).

```
(de-filtre6 (*or . filtres) element6 :normalisation              *or
```

```
    (normaliser6or filtres env succ t) )

;;;Semblable à *or mais ne hurle pas lorsque
;;;les branches sont déséquilibrées.
(de-filtre6 (*xor . filtres) element6 :normalisation
    (normaliser6or filtres env succ nil) )
```

*xor

normaliser6or

```
(defun normaliser6or (filtres env succ warn-p)
    (if (consp filtres)
      (if (consp (cdr filtres))
        (let (nfs nenvs finalenv)
          ;;Normalise les filtres arguments
          (dolist (f (reverse filtres))
            (normaliser6-element f env
               (et-encore6 (nf nenv) (push nf nfs)
                                     (push nenv nenvs) ) ) ) )
          ;;Compare les environnements des branches
          (setq finalenv (car nenvs))
          (do ((envs (cdr nenvs) (cdr envs))
               (fs (cdr filtres) (cdr fs)) )
              ( (null fs) 'ignore)
            (setq finalenv
                  (fusionner6env finalenv (car envs)) )
            ;;Hurler si un quelconque déséquilibre est constaté
            (when warn-p
              (unless (comparer6env-p (car nenvs)
                                      (car envs) )
                (filtre6-erreur
                 "MATCH : alternative non congruente en ~
                  ~A :~%~A~%~A."
                (differer6env (car nenvs) (car envs))
                (car filtres) (car fs) ) ) ) ) )
          (normaliser6groupe nfs finalenv succ '**or2) )
        ;;(*or filtre) est filtre
        (normaliser6-element (car filtres) env succ) )
      ;;(*or) est l'échec.
      (retourner6 '(**not (**sexp)) env succ) ) )

;;;regrouper les filtres de FS par similitude
```

normaliser6groupe

```
(defun normaliser6groupe (fs env succ foncteur)
  (labels ((grouper (fs1 fs2)
             (ameliorer6 `(,foncteur ,fs1 ,fs2) env) )
           (analyser (fs fna orgs)  ;;(assume (consp fs))
             (if (consp (cdr fs))
               (if (consp (cddr fs))
                 (scruter (cdr fs)     ;;trier si > 2
                          (list (funcall fna (car fs)))
                          fna
                          (cdr orgs)
                          (list (car orgs)) )
                                       ;;grouper si =2
                 (grouper (car orgs) (cadr orgs)))
               (car orgs) ) )          ;;tel quel si =1
           (scruter (fs residus fna orgs resorgs)
             (if (consp fs)
               (let ((fr (funcall fna (car fs))))
                 (if (similaire6-p fr (car residus))
```

```
                    (scruter (cdr fs) (cons fr residus)
                             fna (cdr orgs)
                             (cons (car orgs) resorgs) )
                    (let* ((fnp (approfondir6difference
                                 (car residus) ))
                           (deuze
                            (if fnp
                              (analyser (reverse residus)
                                fnp (reverse resorgs))
                              (reduce #'grouper
                                 (reverse resorgs)) )) )
                      (if fs (grouper deuze
                                 (analyser fs fna orgs) )
                          deuze ) ) ) )
                (let ((fnp (approfondir6difference
                            (car residus) )))
                  (if fnp
                    (analyser (reverse residus)
                              fnp (reverse resorgs) )
                    (reduce #'grouper
                            (reverse resorgs) ) ) ) ) ) )
    (retourner6 (analyser fs #'identity fs) env succ) ) )
```

similaire6-p

```
;;; Deux filtres sont similaires s'ils sont de même nature
(defun similaire6-p (f1 f2)
  (eq (car f1) (car f2)) )
```

approfondir6diffference

```
;;; Retourne le sélecteur qui permet d'approfondir l'analyse
(defun approfondir6difference (filtre)
  (case (car filtre)
    ((**or2 **and2 **not **cons) #'cadr)
    ((**setq **ssetq-append **eval-append) #'caddr)
    (t nil) ) )
```

*and

```
(de-filtre6 (*and . filtres) element6
  :normalisation
    (normaliser6and filtres env succ) )
```

normaliser6and

```
(defun normaliser6and (filtres env succ)
  (if (consp filtres)
    (if (consp (cdr filtres))
      (let (nfs (env env))
        (dolist (f (reverse filtres))
          (normaliser6-element f env
                     (et-encore6 (nf nenv)
                                 (setq env nenv)
                                 (push nf nfs) ) ) )
        (normaliser6groupe nfs env succ '**and2) )
      (normaliser6-element (car filtres) env succ) )
    (retourner6 '(**sexp) env succ) ) )
```

Donnons pour illustrer ces regroupements quelques exemples.

```
(*or a b (?x) (?x ?x) c)
```

sera normalisé en :

```
(**OR2
 (**OR2 (**QUOTE A) (**QUOTE B))
 (**OR2 (**CONS (**SETQ X (**SEXP)) (**QUOTE NIL))
        (**CONS (**SETQ X (**SEXP))
                (**CONS (**EVAL X) (**QUOTE NIL))) )
```

tandis que

```
(*or ((?x) foo) (bar ?x) (hux ?x))
```

le sera en :

```
(**OR2 (**CONS (**CONS (**SETQ X (**SEXP))
                       (**QUOTE NIL) )
               (**CONS (**QUOTE FOO) (**QUOTE NIL)) )
       (**OR2 (**CONS (*QUOTE BAR)
                      (**CONS (**SETQ X (**SEXP))
                              (**QUOTE NIL) ) )
              (**CONS (*QUOTE HUX)
                      (**CONS (**SETQ X (**SEXP))
                              (**QUOTE NIL) ) ) ) )
```

5.9.3 Les listes, paires pointées ou autres structures

Les filtres **`*list`**, **`*cons`** et **`*struct`** se traiteront facilement. Le premier n'est qu'un appel explicite à **`normaliser6-segment`** tandis que les suivants se dérivent mot pour mot de leur sémantique.

```
(de-filtre6 (*list . filtres) element6                    *list
  :normalisation
    (normaliser6-segment filtres env succ) )
```

```
(de-filtre6 (*cons filtre1 filtre2) element6              *cons
  :normalisation
    (normaliser6-element filtre1 env
       (et-encore6 (a env)
          (normaliser6-element filtre2 env
              (et-encore6 (d env)
                 (retourner6 (ameliorer6-element
                               `(**cons ,a ,d) env)
                             env succ ) ) ) ) ) )
```

```
(de-filtre6 (*struct type . champs) element6              *struct
  :normalisation
    (labels ((analyser (champs env resultat)
                (if (consp champs)
                  (if (keywordp (car champs))
                     (if (consp (cdr champs))
                       (normaliser6-element
                         (cadr champs) env
                         (et-encore6 (f nenv)
                           (analyser (cddr champs) nenv
                             `(,f ,(car champs)
                                . ,resultat ) ) ) ) )
                       (filtre6-erreur
                        "MATCH : manque un filtre apres ~
                         ~le mot-clef ~S."
                        (car champs) ) )
                     (filtre6-erreur
                      "MATCH : Pas un mot-clef ~S."
```

```
                    (car champs) ) )
                (retourner6 (reverse resultat)
                            env succ ) ) ))
        (analyser champs env `(,type **struct)) ) )
```

5.9.4 Les variables d'élément

On recense trois filtres pour ce faire : `*setq`, `*eval` et l'abréviation polyvalente `*term`. Leur programmation est simple. Tous trois inspectent leur environnement pour savoir si la variable de même nom est déjà liée ainsi que sa nature (élément ou segment). La traduction est alors immédiate ou alors un message d'erreur est émis.

`*setq`

```
(de-filtre6 (*setq nom filtre) element6
  :normalisation
    (normaliser6-element filtre env
      (et-encore6 (f env)
        (if (variable6liee-p nom env)
          (filtre6-erreur
           "MATCH : la variable ~A est redefinie !"  nom)
          (retourner6 `(**and2 ,f (**setq ,nom (**sexp)))
                       (ajouter6element env nom nil)
                       succ ) ) ) ) )
```

`*eval`

```
(de-filtre6 (*eval nom) element6
  :normalisation
    (if (variable6liee-p nom env)
      (let ((lien (env6-liaison nom env 'liaison6variable)))
        (if (subtypep (type-of lien) 'liaison6element)
          (retourner6 `(**eval ,nom) env succ)
          (filtre6-erreur
           "MATCH : ~A n'est pas de type element."
           nom ) ) )
      (filtre6-erreur
       "MATCH : ~A n'est pas definie." nom) ) )
```

`*term`

```
(de-filtre6 (*term nom) element6
  :normalisation
    (if (variable6liee-p nom env)
      (normaliser6-element `(*eval ,nom) env succ)
      (normaliser6-element
        `(*setq ,nom (*sexp)) env succ) ) )
```

5.9.5 Les variables de segments

Deux filtres aussi sont possibles : le bien connu `*terms` ainsi que `*ssetq` qui permet de lier une variable de segment à un segment spécifié par un filtre idoine. On écrira, par exemple, `(*list (*ssetq x (*times (*check numberp))) ??-)` pour récupérer la liste des nombres débutant une liste sous le nom `X`.

Encore une fois leur programmation est simple. Ces filtres s'enquièrent de l'environnement afin de savoir si la variable est libre ou liée et du bon type (segment et non élément) puis se traduisent dans le jeu réduit. Une petite subtilité existe quant à la normalisation de `*ssetq` qui utilise une marque syntaxique `*end-ssetq`.

`*ssetq`

```
(de-filtre6 ((*ssetq nom . filtres) . reste) segment6
  :normalisation
    (normaliser6-segment
      `((*progn . ,filtres) (*end-ssetq ,nom)) env
      (et-encore6 (f env)
         (if (variable6liee-p nom env)
```

```
          (filtre6-erreur
           "MATCH : La variable ~A est deja definie." nom )
          (normaliser6-segment reste
               (ajouter6segment env nom nil nil)
               (et-encore6 (r env2)
                    (retourner6 `(**ssetq-append ,nom ,f ,r)
                                env2
                                succ ) ) ) ) ) ) )
```

*end-ssetq

```
(de-filtre6 ((*end-ssetq nom) . reste) segment6
  :normalisation
    (if reste
      (filtre6-erreur
       "MATCH : (*END-SSETQ ~A) non terminal car suivi de~%~
        ~S." nom reste )
      (retourner6 `(**end-ssetq ,nom) env succ) ) )
```

*terms

```
(de-filtre6 ((*terms nom) . reste) segment6
  :normalisation
    (if (variable6liee-p nom env)
      (let ((lien (env6-liaison nom env 'liaison6variable)))
        (if (subtypep (type-of lien) 'liaison6segment)
          (normaliser6-segment reste env
               (et-encore6 (r env)
                 (retourner6
                  `(**eval-append ,nom ,r) env succ) ) )
          (filtre6-erreur
           "MATCH : ~A n'est pas de type segment."
           nom ) ) )
      (if (or reste (dynamic-ref *match-prefer-true-lists*))
        (normaliser6-segment
         `((*ssetq ,nom (*any)) . ,reste) env succ)
        (normaliser6-segment
         `((*unsafe-ssetq ,nom)) env succ) ) ) )
```

On remarquera que dans le cas où `*terms` est libre, le filtre (`*terms` *nom*) n'est autre que (`*ssetq` *nom* `(*any)`). Lorsque un filtre segmentiel d'affectation est terminal et si la variable globale `*match-prefer-true-list*` est fausse, alors la normalisation utilise la traduction peu sûre `*unsafe-ssetq` que l'on peut toujours forcer par l'emploi direct des `???`*variable*.

*unsafe-ssetq

```
(de-filtre6 ((*unsafe-ssetq nom) . reste) segment6
  :normalisation
    (if (variable6liee-p nom env)
      (filtre6-erreur
        "MATCH : La variable ~S est deja liee." nom )
      (if reste
        (filtre6-erreur
          "MATCH : La variable ~S n'est pas en queue de :~%~
           ~S." nom `((*unsafe-ssetq ,nom) . ,reste) )
        (retourner6 `(**ssetq-terminal ,nom)
                    (ajouter6segment env nom nil nil)
                    succ ) ) ) )
```

5.9.6 Les répétitions

Quoique complexe, le filtre `*times` se prête assez simplement à être normalisé. Il faut toutefois faire attention aux affectations qui pourraient se glisser dans la partie répétitive. Quel pourrait, en effet, être le sens de

```
(*list (*times (*setq x (*sexp))))
```

où la réaffectation de `x` est itérée tant et plus. Aucune répétition n'est effectuée par exemple lorsque ce filtre est confronté à la liste vide, et `X` n'est alors pas liée. Si par contre, ce même filtre est confronté à la liste `(foo bar hux)`, trois affectations vont être commises. Que vaudra `X` en final : la dernière valeur affectée, la première si l'on suppose que les deux autres sont superflues puisque survenant lorsque la variable est valuée, ou bien signalera-t'on une erreur lors de la seconde affectation (pour simple réaffectation ou réaffectation à une valeur différente) ? Beaucoup de questions qui assaillent la sémantique de `*times` dans un délicat domaine. Pour les résoudre, un seul moyen : trancher ! Et donc interdire les affectations sous `*times`. C'est ce qu'effectuera le code suivant :

```
(de-filtre6 ((*times . filtres) . reste) segment6                        *times
  :normalisation
    (let ((loop (gensym)))
      (normaliser6-segment
        `((*progn . ,filtres) (*end-times ,loop)) env
        (et-encore6 (f env1)
           (if (comparer6env-p env env1)
               (normaliser6-segment reste
                        env1 (et-encore6 (r env)
                                (retourner6
                                   `(**labelled-times
                                      ,loop ,f ,r )
                                   env
                                   succ ) ) )
               (progn
                 (format
                   *show-match-expansion-output-stream*
                   "~&;;; Inconsistence locale en ~S, ~
                    j'expanse (*times~{ ~S~}) !~%"
                   (differer6env env env1) filtres )
                 (normaliser6-segment
                  `((*oorr
                      (*progn)
                      (*progn (*progn . ,filtres)
                              (*times . ,filtres) ) )
                     . ,reste )
                  env succ ) ) ) ) ) ) )
```

```
(de-filtre6 ((*end-times nom) . reste) segment6                      *end-times
  :normalisation
    (if reste
      (filtre6-erreur
        "MATCH : (*END-TIMES ~A) non terminal, suivi de~%~
         ~S." nom reste )
      (retourner6 `(**end-times ,nom) env succ) ) )
```

Quelques fioritures restent à décrire. Notamment si le filtre contient une affectation, alors il est tenté une expansion qui peut peut-être le sauver. Le filtre `(*list (*times ?x))` n'accepte que des listes formées du même terme. Certes, mais il accepte aussi la liste vide qui n'est formée de rien. De façon formelle, `(*list (*times ?x))` n'est autre que `(*or () (*list (*setq x (*sexp)) (*times (*eval x))))` qui lui comporte un filtre `*times` admissible puisque indemne de toute affectation interne. En revanche nous venons de violer l'équilibre de la forme `*or` englobante. Est-ce la fin de l'expansion que nous venons de perpétrer ? Pas tout à fait si l'on considère le filtre plus large suivant :

```
(*and (?- ??-)
      ((*times ?x)) )
```

qui lui exclut la liste vide et donc souffre que soit expansé `*times`. Bien sûr, il faut alors un normalisateur malin, en tous cas plus malin que celui présenté ici, car la décision de normalisation ne dépend plus seulement du contexte des variables de filtrage mais bien des valeurs possibles des expressions confrontables. Cette expansion s'accomoderait bien des techniques d'évaluation partielle. Mais ne rabaissons pas notre travail, les améliorations sur les filtres que nous présenterons bientôt permettent de traiter correctement ce cas et donc de le bien compiler.

5.9.7 Les macro-filtres

Ils sont légion ! Mais leur nature de macro les rend limpides. Il suffit donc de les énoncer.

*even

```
(de-filtre6 (*even) macro-element6
  :expansion `(**and2 (**check numberp)
                      (**check evenp) ) )
```

*among

```
(de-filtre6 (*among . expressions) macro-element6
  :expansion
    `(*or . ,(mapcar #'(lambda (e) `(*quote ,e))
                     expressions ) ) )
```

*proper-list

```
(de-filtre6 (*proper-list filtre) macro-element6
  :expansion
    `(*and ,filtre
           (*list (*times (*sexp)) (*tail (*quote ())))) ) )
```

*fail

```
(de-filtre6 (*fail) macro-element6
  :expansion `(*not (*sexp)) )
```

*any

```
(de-filtre6 ((*any) . reste) macro-segment6
  :expansion
    (if (or reste (dynamic-ref *match-prefer-true-lists*))
      `((*times (*sexp)) . ,reste)
      `((*unsafe-any)) ) )
```

*unsafe-any

```
(de-filtre6 ((*unsafe-any) . reste) macro-segment6
  :expansion
    (if (endp reste) `((*tail (*sexp)))
      (filtre6-erreur
       "MATCH : Le filtre ???- n'est pas terminal.") ) )
```

*mcons

```
(de-filtre6 (*mcons filtre . filtres) macro-element6
  :expansion
   (if (consp filtres) `(*cons ,filtre (*mcons . ,filtres))
       filtre ) )
```

*progn

```
(de-filtre6 ((*progn . filtres) . reste) macro-segment6
  :expansion (append filtres reste) )
```

*oorr

```
(de-filtre6 ((*oorr . filtres) . reste) macro-segment6
  :expansion
   `((*tail
       (*or . ,(mapcar #'(lambda (f) `(*list ,f . ,reste))
                       filtres )) )) )
```

*maybe

```
(de-filtre6 ((*maybe . filtres) . reste) macro-segment6
  :expansion
   `((*oorr (*progn . ,filtres) (*progn)) . ,reste) )
```

*listwhere

```
(de-filtre6 (*listwhere . filtres) macro-element6
  :expansion
```

```
  `(*and . ,(mapcar #'(lambda (filtre)
                        `((*any) ,filtre (*tail (*sexp))))
                    filtres )) )
```

Il ne reste plus pour conclure que le filtre **`*tail`** qui n'est pas un macro-filtre car il permet de revenir à **`normaliser6-element`** depuis **`normaliser6-segment`**.

`*tail`

```
(de-filtre6 ((*tail filtre) . reste) segment6
  :normalisation
   (if reste
       (filtre6-erreur
         "MATCH : le filtre ((*TAIL ~A) ~{~A ~}) ~
          n'est pas terminal."
         filtre reste )
       (normaliser6-element filtre env succ) ) )
```

Divers exemples sont déjà apparus et d'autres apparaîtront encore, aussi conclurons-nous ici le processus de normalisation.

Résumé

La normalisation traduit un filtre écrit dans un langage riche et profus en un filtre équivalent qui, lui, appartient à un langage restreint. Ce langage est restreint à la fois en nombre de primitives et en ce sens que pour un effet souhaité il n'existe que peu de façons différentes de l'obtenir.

La traduction inspecte la syntaxe et vérifie les grandes contraintes : équilibrage des alternatives, absence d'affectations dans les boucles et macro-expansion bornée.

Quelques éléments de choix non triviaux apparaissent déjà comme le regroupement de filtres similaires en disjonctions ou conjonctions.

Enfin on remarquera la persistence de l'architecture du moteur à filtrer qui est ici devenu moteur à traduire.

Exercice 5.20 : Imaginer et programmer un nouveau filtre de haut niveau et donner sa méthode de normalisation.

5.10 L'amorce

Tout ce qui précède définit un compilateur de filtres fonctionnellement intéressant mais manque encore le définisseur de filtres `de-filtre6`. Loin d'être une banale extrapolation de ces illustres prédécesseurs, nous allons le révéler dans toute sa complexité.

Le définisseur de filtres, comme son nom l'indique, définit les filtres, leur syntaxe (arguments, structure) et leurs méthodes propres (compilation, interprétation, normalisation). La forme d'un appel à `de-filtre6` est le suivant :

(de-filtre6 *appel type*
  :*facette forme*)

Les types possibles sont `rudimentaire6`, `element6`, `segment6`, `macro-segment6` ou `macro-element6`. Les facettes possibles sont `:compilation`, `:interpretation`, `:expansion`, `:normalisation` et `:extraction`. On verra plus loin qu'existe aussi la facette `:amelioration`. Les *formes* peuvent utiliser les variables implicites qui ont pour nom `e`, `env`, `succ`, `alt`, `segvarenv` et `timesvarenv` mais toutes ne sont pas disponibles suivant les facettes. La forme d'appel peut revêtir deux structures suivant que l'on use ou pas du contexte droit. Ce peut être

(*nom-du-filtre variables-du-filtre*)
ou ((*nom-du-filtre variables-du-filtre*) *variables-de-contexte*)

Jamais auparavant n'avions-nous vérifié la syntaxe des filtres aussi souhaitons-nous réparer cet oubli. Vérifier la syntaxe suppose de vérifier que tout appel à un filtre satisfait la structure indiquée par la forme d'appel lors de sa définition : c'est typiquement un problème de filtrage ! L'ennui est que justement nous souhaitons nous doter de cet outil mais pour l'heure nous n'en disposons pas encore. Le problème d'amorçage commence de se dessiner mais fort heureusement les formes d'appel ont une structure très simple que l'on peut vérifier ou traduire en un filtre grâce à `verifier-appel` et `appel->filtre`.

```
;;; Vérifier que la forme d'appel est (nom variables [ . variable])
;;; ou encore ((nom variables [ . variable]) variables [ . variable]).
(defun verifier-appel (appel)                                  verifier-appel
   (labels ((tous-symboles (liste)
              (or (symbolp liste)
                  (null liste)
                  (and (symbolp (car liste))
                       (tous-symboles (cdr liste)) ) ) ) ))
     (or (tous-symboles appel)
         (and (tous-symboles (car appel))
              (tous-symboles (cdr appel)) ) ) ) )

;;; Transformer une forme d'appel en un filtre normalisé.
;;; On supprime le test du nom du filtre au passage.
;;; Le *CONS de tête est de trop mais ne sera pas testé.
(defun appel->filtre (appel)                                   appel->filtre
  (labels ((traduire (exp) ;traduit une liste de symboles
             (cond ((consp exp)
                    `(**cons (**setq ,(car exp) (**sexp))
                             ,(traduire (cdr exp)) ))
                   ((null exp)    `(**quote nil))
                   ((symbolp exp)
                    `(**setq ,exp (**sexp)) ) ) ) ))
    (if (consp (car appel))
        `(**cons (**cons (**sexp) ,(traduire (cdar appel)))
                 ,(traduire (cdr appel)) )
        `(**cons (**sexp) ,(traduire (cdr appel))) ) ) )
```

Il apparaît donc que dès lors que l'on sait compiler les filtres `**cons`, `**quote`, `**sexp` et `**setq` alors tous les appels de filtres sont vérifiables. Nous reprendrons donc la technique des précédents définisseurs de filtres qui combinaient perversement `apply` et `&rest` pour définir un `de-filtre6` temporaire.

```
;;; DE-FILTRE version temporaire, juste pour définir les quatre filtres
;;; de base qui suivent. DE-FILTRE est très, très simplifié !!!
```

de-filtre6

```
(defmacro de-filtre6 (appel type &key compilation)
  (unless (member type '(rudimentaire6))
    (error "DE-FILTRE6: erreur dans l'amorce. ~
            Type de filtre inconnu ~A." type ) )
  (let ((code `(setf (gethash ',(car appel) *filtres*)
                     (make-rudimentaire6
                      :nom ',(car appel)
                      :compilation
                      #'(lambda (e f env succ alt
                                 segvarenv timesvarenv )
                          (apply #'(lambda ,(cdr appel)
                                         ,compilation)
                                 (cdr f) ) ) ) ) ) ))
    (when *show-pattern-definition*
      (format *show-pattern-definition-output-stream*
           "~&;; Definition du filtre rudimentaire ~A :~%"
           (car appel) )
      (write code
         :stream *show-pattern-definition-output-stream*
         :pretty t )
      (terpri *show-pattern-definition-output-stream*) )
    code ) )
```

C'est simple mais suffisant pour avaler maintenant les quatre filtres qui suivent.

```
(de-filtre6 (**sexp) rudimentaire6
  :compilation
    (reussir6 e env succ alt) )

(de-filtre6 (**quote expression) rudimentaire6
  :compilation
    (si (egal e `',expression)
        (reussir6 e env succ alt)
        (reprendre6 alt) ) )

(de-filtre6 (**cons filtre1 filtre2) rudimentaire6
  :compilation
   (si `(consp ,e)
       (compiler6-element (le-car e) filtre1
         (ajouter6consp env e)
         (et-puis6 (e1 env1 alt1)
           (compiler6-element (le-cdr e) filtre2 env1 succ
                              alt1 segvarenv timesvarenv ) )
         alt (initial6env) (initial6env) )
       (reprendre6 alt) ) )

(de-filtre6 (**setq nom filtre) rudimentaire6
  :compilation
   (compiler6-element e filtre env
     (et-puis6 (e1 env1 alt1)
       (faire (affecter nom e)
              (reussir6 e1 (ajouter6element env1 nom nom)
                        succ alt1 ) ) )
     alt segvarenv timesvarenv ) )
```

À peine plus simples que leurs originaux, ces filtres suffisent à compiler les résultats de `appel->filtre` : on sait donc engendrer le code correspondant à la vérification de la syntaxe des filtres. Il ne reste donc plus qu'à engendrer le vrai `de-filtre6`.

de-filtre6

```
(defmacro de-filtre6 (&whole def
    appel type
    &key (compilation       'ignore si-compilation)
         (interpretation    'ignore si-interpretation)
         (normalisation     'ignore si-normalisation)
         (amelioration      'ignore si-amelioration)
         (expansion         'ignore si-expansion)
         (extraction        'ignore si-extraction) )
  (unless (subtypep type 'filtre6)
    (error "DE-FILTRE : Le type ~A est inconnu." type) )
  (unless (verifier-appel appel)
    (error "DE-FILTRE : La liste des variables ~A ~
            est incorrecte." appel) )
  (let* ((nom (if (consp (car appel))
                  (caar appel) (car appel) ))
         (filtre (chercher6filtre nom type appel def)) )
   `(progn (chercher6filtre ',nom ',type ',appel ',def)
     (format *show-pattern-definition-output-stream*
             "~&; DE-FILTRE : Je charge ~A."
             ',nom )
     ,(when si-compilation
       (engendrer6methode-compilation
         filtre compilation ) )
     ,(when si-interpretation
       (engendrer6methode-interpretation
         filtre interpretation ) )
     ,(when si-normalisation
       (engendrer6methode-normalisation
         filtre normalisation ) )
     ,(when si-expansion
       (engendrer6methode-expansion
         filtre expansion ) )
     ,(when si-amelioration
       (engendrer6methode-amelioration
         filtre amelioration ) )
     ,(when si-extraction
       (engendrer6methode-extraction
         filtre extraction ) )
     ',nom ) ) )
```

La macro a une forme d'appel complexe à la mesure de toutes les facettes qu'elle sait traiter mais son corps est simple et consiste à créer un objet filtre de la bonne nature grâce à **`chercher6filtre`** et à l'enrichir de ses méthodes associées.

chercher6filtre

```
(defun chercher6filtre (nom type appel def)
  (declare (ignore def))
  (let ((le-filtre (or (obtenir6filtre nom)
                       (creer6filtre type nom) )))
    (unless (eq type (type-of le-filtre))
      (format *show-pattern-definition-output-stream*
        "~&; DE-FILTRE : Le filtre ~A change de type ~
         (~A -> ~A) !!!"
        nom (type-of le-filtre) type )
      (setq le-filtre (creer6filtre type nom)) )
    (unless (verifier-congruence le-filtre appel)
      (format *show-pattern-definition-output-stream*
        "~&; DE-FILTRE : Le filtre ~A change de ~
          syntaxe !!!~%; ~A -> ~A."
```

```
        nom (filtre6-syntaxe le-filtre) appel )
      (setq le-filtre (creer6filtre type nom)) )
    (setf (filtre6-syntaxe le-filtre) appel)
    (setf (filtre6-variables le-filtre)
          (aplatir-appel appel) )
    (setf (filtre6-filtre le-filtre) (appel->filtre appel))
    le-filtre ) )
```

creer6filtre

```
(defun creer6filtre (type nom)
   (let ((f (make-instance (symbol-class type) :nom nom)))
        (setf (gethash nom *filtres*) f)
        f ) )
```

La fonction `chercher6filtre` est appelée deux fois dans `de-filtre6`. La première fois dans l'expansion, la seconde dans l'expansé. La raison est que l'objet filtre est nécessaire pour l'expansion notamment si l'on définit les facettes les unes après les autres. Il faut alors s'assurer que le filtre n'a pas changé de nature, ni de liste de variables. L'objet filtre est aussi nécessaire dans la forme expansée car sa compilation exige que l'on sache extraire de lui ses méthodes appropriées. Les méthodes engendrant les facettes sont toutes semblables et donc toutes factorisées en `engendrer6methode`.

```
(defun engendrer6methode-compilation (filtre corps)
  (engendrer6methode filtre corps
        'rudimentaire6 'compilation ) )

(defun engendrer6methode-interpretation (filtre corps)
  (engendrer6methode filtre corps
        'rudimentaire6 'interpretation ) )

(defun engendrer6methode-normalisation (filtre corps)
  (engendrer6methode filtre corps
        'executable6 'normalisation ) )

(defun engendrer6methode-amelioration (filtre corps)
  (engendrer6methode filtre corps
        'rudimentaire6 'amelioration ) )

(defun engendrer6methode-expansion (filtre corps)
  (engendrer6methode filtre corps
        'macro6 'expansion) )

(defun engendrer6methode-extraction (filtre corps)
  (engendrer6methode filtre corps
        'rudimentaire6 'extraction) )
```

engendrer6methode

```
(defun engendrer6methode (filtre corps nom-type nom-methode)
  (unless (subtypep (type-of filtre) nom-type)
    (error "DE-FILTRE6: le type de filtre ~A n'admet ~
            pas de methode de ~(~A~)."
           (type-of filtre) nom-methode) )
  (let* ((nom (filtre6-nom filtre))
         (variables (filtre6-variables filtre))
         (appel (filtre6-syntaxe filtre))
         (fn
          (let* ((arg (gensym))
                 (env (reduce
                       #'(lambda (var env)
                           (ajouter6element env var var) )
                       variables
                       :from-end t
```

```
                    :initial-value
                    (ajouter6consp (initial6env) arg t))))
          (when (consp (car appel))
            (setq env (ajouter6consp env (le-car arg) t)))
          `(lambda ,(variables6facette arg nom-methode)
             ,(symbol-name nom)    ;;doc-string
             (let ,(mapcar #'(lambda (var) `(,var nil))
                           variables )
                 (if ,(compiler6 arg
                          (filtre6-filtre filtre)
                          env et-puis6t ou-sinon6nil
                          (initial6env) (initial6env) )
                   ,corps
                   (filtre6incorrect ',nom ,arg) ) ) ) )))
  (when *show-pattern-definition*
    (format *show-pattern-definition-output-stream*
      "~&;; je definis ~A (methode de ~(~A~)):~%"
      (filtre6-nom filtre) nom-methode )
    (write
      `(setf (,(concat nom-type "-" nom-methode)
               (obtenir6filtre ',(filtre6-nom filtre)))
             #',fn )
      :stream *show-pattern-definition-output-stream*
      :pretty t )
    (terpri *show-pattern-definition-output-stream*) )
  `(setf (,(concat nom-type "-" nom-methode)
          (obtenir6filtre ',(filtre6-nom filtre)) )
         #',fn ) ) )
```

La mise en forme de la méthode définie est la suivante : on teste d'abord si l'appel est conforme puis on le traite ou l'on appelle la fonction `filtre6incorrect` qui émettra un message d'erreur. La fonction `filtre6incorrect` factorise le code traitant les erreurs de syntaxe.

```
;;;Signaler une syntaxe de filtre incorrecte
(defun filtre6incorrect (nom arg)                                filtre6incorrect
  (let ((filtre (obtenir6filtre nom)))
    (filtre6-erreur
      "Le filtre ~A est syntaxiquement incorrect :~%~
       Syntaxe attendue : ~S~%~
       Syntaxe recue    : ~S."
      (filtre6-nom filtre) (filtre6-filtre filtre) arg ) ) )
```

```
;;;La fonction qui imprime les erreurs
(defun filtre6-erreur (fmt &rest args)                           filtre6-erreur
  (apply #'error fmt args) )
```

Quant à `filtre6-erreur`, elle encapsule le traitement des erreurs (que l'on a toujours intérêt à confiner ne serait-ce que pour la redéfinir pour automatiser les tests).

Donnons un exemple de la définition d'un filtre avec ce nouveau `de-filtre6`, par exemple, le filtre `*ssetq`.

```
(de-filtre6 ((*ssetq nom . filtres) . reste) segment6
  :normalisation
  ... )
```

sera expansé en

```
(SETF (EXECUTABLE6-NORMALISATION (OBTENIR6FILTRE '*SSSETQ))
 #'(LAMBDA (G8 ENV SUCC) "*SSSETQ"
    (LET ((NOM NIL)        ;;(**cons
          (FILTRES NIL)    ;;(**cons (**setq nom (**sexp))
```

```
   (RESTE NIL))      ;;(**setq filtres (**sexp)) )
(IF (AND (CONSP (CDAR G8))  ;;(**setq reste (**sexp)) )
         (PROGN
          (SETQ NOM (CADAR G8))
          (SETQ FILTRES (CDDAR G8))
          (SETQ RESTE (CDR G8))
          T))
    ...
    (FILTRE6INCORRECT '*SSSETQ G8)))))
```

On remarquera que les tests `(CONSP G8)` et `(CONSP (CAR G8))` ont été escamotés puisque naturellement vérifiés par `normaliser6`. C'est la fonction `engendrer6methode` qui a donc produit les `ajouter6consp` adéquats.

On pourra objurguer qu'une traduction plus correcte de `((*ssetq nom . filtres) . reste)` est `((?-?nom ??filtres) ??reste)` qui teste que les segments sont de vraies listes. Là encore on voit le caractère insidieux des paires pointées réapparaître. On pourra, si on le souhaite, poursuivre l'amorce, une fois chargés tous les filtres, en redéfinissant la fonction `appel->filtre` et en recompilant toutes les définitions des filtres. Pour notre part nous continuerons de perpétrer les errements en vigueur et en Lisp et ne point nous mettre martel en tête de ces, peut-être, gênantes fins de listes.

Ayant atteint notre but de disposer d'un bon définisseur de filtres, on peut maintenant charger toutes les méthodes de compilation, d'interprétation (pour les tests), de normalisation et d'expansion des filtres. Toutes ces méthodes ont été vues et commentées dans les sections précédentes. Ceci fait, il ne reste plus qu'à définir les macros d'interface. `match-lambda` et `match-all-lambda` partagent en fait le même code

```
;;; Retourne la première des solutions.
(defmacro match-lambda (&whole call)                    match-lambda
  (flet ((traiter (label code) (sechapper label code)))
    (make-match-lambda call #'traiter nil) ) )

;;; Effectue tous les filtrages
(defmacro match-all-lambda (&whole call)                match-all-lambda
  (flet ((traiter (label code)
            `(progn ,code nil) ) )
    (make-match-lambda call #'traiter nil) ) )
```

La seule différence apparente est le traitement du code d'un succès. Dans le premier cas on retourne la valeur ramenée grâce à un échappement bien senti. Dans le second on évalue et on retourne `nil` qui force l'échec et permet d'obtenir les autres solutions (s'il y en a).

```
                                                        make-match-lambda
(defun make-match-lambda (call traiter arite)
  (let ((org-filtre 'ignore) ;;filtre original
        (filtre 'ignore)     ;;filtre normalisé
        (doc-string 'ignore) ;;chaine de documentation
        (faire 'ignore)      ;;corps de la macro
        (args 'ignore)       ;;nom des variables engendrées
        (label 'ignore)      ;;label du return-from engendré
        (nenv 'ignore)       ;;env après normalisation
        (env 'ignore)        ;;env de compilation
        (fn 'ignore)         ;;lambda engendrée
        (code 'ignore) )     ;;corps de la lambda engendrée
    (dynamic-let ((*match-lambda-mode* (car call)))
      (multiple-value-setq (doc-string org-filtre faire)
                  (analyse-match-lambda (cdr call)) )
      (when *show-match-expansion*
        (format *show-match-expansion-output-stream*
          "~&; A : filtre a normaliser : (write org-filtre
         :stream *show-match-expansion-output-stream*
         :pretty t ) )
```

```
      (setq label (gensym))
      (normaliser6-element
       `(*and ,org-filtre
              (*success
               ,(funcall traiter label (e-faire faire)) ) )
       (initial6env)
       (et-encore6 (f env)
                   (setq filtre f)
                   (setq nenv env) ) )
      (if arite
        (setq args `(list . ,(let ((result nil))
                               (dotimes (i arite)
                                 (push (gensym) result) )
                               result )))
        (setq args (gensym)) )
      (setq env (ajouter6succes (transformer6env nenv)))
      (when arite
        (do ((i 0 (1+ i))
             (path args (le-cdr path)) )
            ( (= i arite)
              (setq env (ajouter6equal env path nil t)) )
          (setq env (ajouter6consp env path t)) ) )
      (when *show-match-expansion*
        (format *show-match-expansion-output-stream*
         "~&; A : resultat de normalisation : (write filtre
         :stream *show-match-expansion-output-stream*
         :pretty t ) )
      (setq code (compiler6 args filtre env
                  (et-puis6 (e env alt) (reprendre6 alt))
                  ou-sinon6nil
                  (initial6env)
                  (initial6env) ))
      (setq fn `(lambda ,(if arite (cdr args) (list args))
                  ,@(when doc-string (list doc-string))
                  (block ,label
                    ,(enchasser6succes env
                       (enchasser6variables env code) ) ) ))
      (when *show-match-expansion*
        (format *show-match-expansion-output-stream*
         "~&; A : resultat d'expansion : (write fn
         :stream *show-match-expansion-output-stream*
         :pretty t ) )
      `#',fn ) ) )
```

La forme d'appel est tout d'abord décomposée afin qu'en soient extraits la chaîne de documentation optionnelle, le filtre et les formes à évaluer en cas de succès du filtre. Cette analyse est effectuée par :

```
(defun analyse-match-lambda (parameters)
  (if (stringp (car parameters))
    (values (car parameters) (cadr parameters)
            (cddr parameters) )
    (values nil (car parameters) (cdr parameters)) ) )
```

analyse-match-lambda

Cette analyse est bien légère mais a le mérite d'être simple à programmer. Toutefois nous allons bientôt disposer d'un outil permettant de mieux faire : un filtreur ! Aussi dès que nous en disposerons, redéfinirons-nous cette fonction.

Le filtre est alors normalisé. Les formes composant le corps de la macro sont insérées dans le filtre grâce à `**success` et `**and2`. Ainsi `(match-lambda` *filtre faire*`)` est-il converti en `(**and2` *filtre* `(**success` *faire*`))` tandis que `(match-case-lambda (`$filtre_i$ $faire_i$`) ...)` se transforme en `(*xor (*and` *filtre$_i$* `(*success` *faire$_i$*`))` ...). Le résultat est alors compilé puis utilisé pour synthétiser une fonction.

Nous avions utilisé des variables de stockage à nom original afin d'éviter d'occultes captures via les prédicats qu'un utilisateur mal intentionné aurait pu commettre. Toutefois dans la majorité des cas ces précautions sont superflues. Une variable globale `*match-prefer-anonymous-variables*` gouverne la génération des noms des variables de stockage.

```
;;; Transforme l'environnement reçu du normalisateur en environnement
;;; pour le compilateur. Détermine les variables de stockage associées.
(defun transformer6env (env)                                  transformer6env
  (dolist (lien env)
    (typecase lien
      (liaison6element
       (if *match-prefer-anonymous-variables*
         (let ((g (gensym)))
           (setf (liaison6element-variable lien) g) )
         (setf (liaison6element-variable lien)
               (liaison6-nom lien) ) ) )
      (liaison6segment
       (if *match-prefer-anonymous-variables*
         (let ((g1 (gensym)) (g2 (gensym)))
           (setf (liaison6segment-debut-ref lien) g1)
           (setf (liaison6segment-fin-ref lien) g2) )
         (let ((g (gensym)))
           (setf (liaison6segment-debut-ref lien)
                 (liaison6-nom lien) )
           (setf (liaison6segment-fin-ref lien) g) ) ) )
      (t nil) ) )
  env )

;;; Allouer toutes les variables de stockage, Il n'est pas nécessaire
;;; qu'elles soient initialisées car elles le seront dans le code engendré.
;;; En fait il suffit de les extraire de l'environnement
(defun allouer6variables (env)                              allouer6variables
  (mapcan #'(lambda (lien)
              (typecase lien
                (liaison6element
                 (list (liaison6element-variable lien)) )
                (liaison6segment
                 (list (liaison6segment-debut-ref lien)
                       (liaison6segment-fin-ref lien) ) )
                (t nil) ) )
          env ) )
```

Quelques fioritures supplémentaires apparaissent dans `make-match-lambda` relatives à l'arité de la fonction engendrée et seront analysées lors du traitement des fonctions d'arité n définies par filtrage.

Lorsque ces macros sont définies, le moteur est enfin opérationnel. On peut donc en profiter pour redéfinir derechef `analyse-match-lambda`.

```
(defun analyse-match-lambda (parameters)
  (funcall                                               analyse-match-lambda
   (match-lambda ((*ssetq doc-string
                          (*maybe (*check stringp)) )
                  (*setq filtre (*sexp))
                  (*ssetq faire (*sexp) (*any)) )
    (return-from analyse-match-lambda
     (values (when doc-string (car doc-string))
             filtre faire ) ) )
   parameters ) )
```

Le filtre est maintenant particulièrement clair et indique bien la syntaxe que doivent arborer les appels à `match-lambda`. En guise d'exemple et parce qu'il n'est pas si laid, voici l'expansion de la nouvelle définition de `analyse-match-lambda`.

```
(LAMBDA (G54)
  (BLOCK G52
    (MACROLET
     ((G66 NIL '(LET ((FAIRE G55)
                      (FILTRE G57)
                      (DOC-STRING (EXECUTER6OTER G58 G59)))
                 (RETURN-FROM ANALYSE-MATCH-LAMBDA
                   (VALUES (WHEN DOC-STRING
                                 (CAR DOC-STRING) )
                           FILTRE
                           FAIRE)))))
     (LET (G55 G56 G57 G58 G59)
      (SETQ G58 G54)
      (FLET ((G60 (G61)
               (SETQ G59 G61)
               (AND (CONSP G61)
                    (PROGN
                     (SETQ G57 (CAR G61))
                     (SETQ G55 (CDR G61))
                     (FLET ((G64 (G65)
                              (AND (NULL G65)
                                   (PROGN (G66) NIL))))
                       (AND (CONSP (CDR G61))
                            (LABELS
                               ((G62 (G63)
                                 (OR (G64 G63)
                                     (AND
                                      (CONSP G63)
                                      (G62 (CDR G63))))))
                              (G62 (CDDR G61)))))))))
        (OR (AND (CONSP G54)
                 (STRINGP (CAR G54))
                 (G60 (CDR G54)))
            (G60 G54)))))))
```

Il est temps maintenant d'étoffer notre arsenal de macros. Dans un premier temps, définissons des macros permettant de définir des fonctions ou des macros mettant en œuvre un filtrage. On a pu remarquer que, nonobstant l'urgence qu'il y avait à redéfinir `analyse-match-lambda`, celle-ci n'était qu'à une η-conversion près la fonction synthétisée par le `match-lambda` interne. Dans ce cas, autant encapsuler ce tour de main et créer `def-match-macro` et `def-match-function`.

`def-match-macro`

```
;;; Une macro dont les paramètres sont obtenus par filtrage.
;;; Tout appel incorrect provoque une erreur automatiquement.
;;; Syntaxe : (def-match-macro (nom . filtres...) corps...)
(defmacro def-match-macro ((nom . filtres) . faire)
  (let ((g (gensym)))
    `(defmacro ,nom (&rest ,g)
       (funcall (match-lambda ,filtres
                  ,(sechapper nom (apply #'faire faire)) )
                ,g )
       (error "~A : Syntaxe incorrecte :~%~
               *** Forme attendue      : ~A~%~
               *** Parametres actuels  : ~A."
              ',nom ',filtres ,g )) ) )

;;; On redéfinit DEF-MATCH-MACRO avec lui-même mais de façon plus sûre.
```

def-match-macro

```
(def-match-macro
  (def-match-macro ((*setq nom (*check symbolp))
                    (*terms parms) )
        (*terms faire) )
   (let ((g (gensym)))
     `(defmacro ,nom (&whole ,g)
        (funcall (match-lambda ,parms
                    ,(sechapper nom (e-faire faire)) )
                 (cdr ,g) )
        (error "~A : Syntaxe incorrecte :~%~
                *** Forme attendue     : ~A~%~
                *** Parametres actuels : ~A."
               ',nom ',parms (cdr ,g) )) ) )
```

Si l'on s'en souvient, le définisseur de fonctions décrites par filtrage autorisait deux syntaxes suivant que l'on désirait une fonction d'arité fixe ou variable. La fonction `ek1` suivante est binaire et compare ses deux arguments.

```
(def-match-function ek1
   ( ?x ?x => t)
   ( ?- ?- => nil) )
```

De cette définition on peut déduire celle-ci, qui est son équivalente sauf que l'arité est maintenant variable.

```
(def-match-function ek2
   ( (?x ?x) t)
   ( (?- ?-) nil) )
```

On ne voit ici qu'un `match-case-lambda` déguisé correspondant au filtre :

```
(*xor (*and (*cons (*setq x (*sexp))
                   (*cons (*eval x) (*quote nil)) )
            (*success t) )
      (*and (*cons (*sexp)
                   (*cons (*sexp) (*quote nil)) )
            (*success nil) ) )
```

Notez le `*xor` initial qui permet d'esquiver ici la vérification de déséquilibrage, naturelle pour une discrimination par cas. La différence entre la version binaire et n-aire est que les filtres mis en italiques ne seront pas vérifiés dans la version binaire puisque nécessairement satisfaits. Le regroupement des variables en une unique liste, comme si obtenue par `&rest` n'est qu'un artifice commode pour traiter les fonctions d'arité fixe sans perte d'efficacité.

Voici donc `def-match-function` accompagnée des ses utilitaires propres.

def-match-function

```
(def-match-macro
   (def-match-function (*setq nom (*check symbolp))
            (*ssetq doc-string (*maybe (*check stringp)))
            (*ssetq regles (*times (?- ?- ???-))) )
   (let ((arite (chercher6arite regles)))
     (if arite
       (match-let
          (function (lambda ?variables ??faire))
          (make-match-lambda
            `(match-lambda
               (*xor . ,(assembler6regles regles arite nom))
               (error "Erroneous arguments for ~A."
                      ',nom ) )
            #'sechapper
            arite )
         `(defun ,nom ,variables ,@doc-string . ,faire) )
       (let ((g (gensym)))
```

```
    `(defun ,nom (&rest ,g) ,@doc-string
           (match-case ,g
               ,@regles
               ((*sexp)
                (error "Incorrect arguments for ~A."
                       ',nom ) ) ) ) ) ) ) )
```

```
;;; Assembler les filtres en un seul.
(defun assembler6regles (regles arite nom)
  (mapcar #'(lambda (regle)
             `(*and ,(reduce #'(lambda (filtre filtres)
                                 `(*cons ,filtre ,filtres) )
                             (subseq regle 0 arite)
                             :initial-value `(*quote nil)
                             :from-end t )
                    (*success
                     ,(sechapper nom
                       (e-faire
                         (subseq regle (1+ arite))) )) ) )
          regles ) )
```

assembler6regles

```
;;; Chercher l'arité d'un ensemble de règles grâce à la première puis
;;; vérifier les autres par rapport.
(defun chercher6arite (regles)
  (match-let (??filtres ?'=> ???-)
             (car regles)
        (let ((arite (length filtres)))
          (dolist (regle (cdr regles))
            (match-let (??filtres => ???-)
                       regle
              (unless (= arite (length filtres))
                (filtre6-erreur
                 "DEF-MATCH-FUNCTION : arité fluctuante !~%~
                  La clause suivante ne devrait comporter ~
                  que ~D filtre(s):~%~S."
                 arite filtres ) ) ) )
          arite ) ) )
```

chercher6arite

Nous verrons par la suite de nombreux autres exemples de ces somptueux définisseurs de fonctions ou macros.

La macro `match-case-lambda` annoncée dans le préambule de ce chapitre est à même maintenant d'être définie.

```
(def-match-macro
       (match-case-lambda
              (*ssetq doc-string (*maybe (*check stringp)))
              (*ssetq rules (*times (?- ???-))) )
   (let ((arg (gensym))
         (label (gensym))
         (filtre 'ignore)
         (fn 'ignore) )
     (setq filtre
          `(*xor ,@(mapcar
                    #'(lambda (rule)
                       `(*and ,(car rule)
                              (*success
                               ,(sechapper label
                                 (e-faire (cdr rule)) )) ) )
                    rules )) )
```

match-case-lambda

```
    (setq fn
          `(lambda (,arg)
             ,@doc-string
             (block ,label
               (match-let ,filtre ,arg
                 (error "MATCH-CASE-LAMBDA : ~
                         No case found" ) ) ) ) ) )
    `#',fn ) )
```

Enfin pour parachever l'amorce, le macro-caractère préfixant les filtres peut profiter de l'arsenal mis en place.

```
(set-macro-character #\? #'(lambda (stream void-ch)
   ;; ? reconnait *sexp, *quote, *term, *any, *terms et *check.
   (let ((exp (read stream)))
     (match-case exp
       ((*quote |-|)                  `(*sexp))
       ((*check symbolp)              `(*term ,exp))
       ((quote (*term e))             `(*quote ,e))
       ((*quote (*sexp))              `(*any))
       (((*quote *term) (*term e))    `(*terms ,e))
       (((*quote *terms) (*term e))   `(*unsafe-ssetq ,e))
       (((*quote *any))               `(*unsafe-any))
       ((*sexp)
        (error "Le filtre ?~A est incorrect." exp)) ) ) ))
```

?

On notera que la discrimination est faite par filtrage et donc exclut les fautes de syntaxe. Si l'on avait voulu écrire ce qui précède à la main, n'aurait-on pas négligé quelques vérifications terminales ? On en jugera sur l'expansion qui suit :

```
(LAMBDA (G3)
  (BLOCK G2
    (MACROLET
       ((G26 NIL
         '(RETURN-FROM G0
            (ERROR "Le filtre ?~A est incorrect." EXP)))
       (G23 NIL '(RETURN-FROM G0 '(*UNSAFE-ANY)))
       (G20 NIL
         '(LET ((E G4))
            (RETURN-FROM G0
                   (CONS '*UNSAFE-SSETQ (LIST E)) )))
       (G17 NIL
         '(LET ((E G4))
            (RETURN-FROM G0 (CONS '*TERMS (LIST E)))))
       (G14 NIL '(RETURN-FROM G0 '(*ANY)))
       (G11 NIL
         '(LET ((E G4))
            (RETURN-FROM G0 (CONS '*QUOTE (LIST E)))))
       (G8 NIL '(RETURN-FROM G0
                   (CONS '*TERM (LIST EXP)) ))
       (G5 NIL '(RETURN-FROM G0 '(*SEXP))))
      (LET (G4)
        (OR (AND (EQ G3 '-) (PROGN (G5) NIL))    ;;?-
            (AND (SYMBOLP G3) (PROGN (G8) NIL))  ;;?x
            (AND (CONSP G3)
                 (OR (AND (EQ (CAR G3) 'QUOTE)       ;;?'
                          (CONSP (CDR G3))
                          (PROGN
                            (SETQ G4 (CADR G3))
                            (AND (NULL (CDDR G3))
```

```
                         (PROGN (G11) NIL))))
                 (AND (EQ (CAR G3) '*SEXP)       ;;??-
                      (NULL (CDR G3))
                      (PROGN (G14) NIL))
                 (AND (EQ (CAR G3) '*TERM)       ;;??x
                      (CONSP (CDR G3))
                      (PROGN
                        (SETQ G4 (CADR G3))
                        (AND (NULL (CDDR G3))
                             (PROGN (G17) NIL))))
                 (AND (EQ (CAR G3) '*TERMS)      ;;???x
                      (CONSP (CDR G3))
                      (PROGN
                       (SETQ G4 (CADR G3))
                       (AND (NULL (CDDR G3))
                            (PROGN (G20) NIL))))
                 (AND (EQ (CAR G3) '*ANY)        ;;???-
                      (NULL (CDR G3))
                      (PROGN (G23) NIL))))
        (PROGN (G26) NIL))))))
```

On notera aussi l'exquise suavité qu'il y a à écrire un filtre plutôt que l'embrouillamini précédent. L'ajout ou la modification sont simples sur les filtres par comparaison au charcutage dangereux de code qui lui est équivalent. Dieu nous garde d'écrire de pareilles choses maintenant qu'existe le compilateur de filtres !

5.11 Les améliorations

En fait, il nous faut maintenant l'avouer, nous avons beaucoup menti dans ce qui précède. Tout lecteur scrupuleux ayant vérifié à la main (ou de tête) les expansions qui étaient données en exemples l'aura remarqué : elles étaient vraiment mieux que ce qui était réellement obtenu. L'essentiel dans un premier temps était de disposer d'un filtreur fonctionnellement correct. Sa puissance est telle que maintenant nous allons pouvoir l'appliquer à lui-même et systématiquement améliorer ce qui se peut.

Les améliorations que nous allons passer en revue concernent soit les générateur de code de bas niveau, soit les méthodes de normalisation. Commençons par les générateurs de code.

5.11.1 Les améliorations sur les générateurs de code

L'expansion réelle de `(match-lambda (?- ?-) "Succès")` est

```
(LAMBDA (G92)
   (BLOCK G93
      (FLET ((G94 NIL (RETURN-FROM G93 (PROGN "Succès"))))
            (IF (CONSP G92)
                (IF (CONSP (CDR G92))
                    (IF (EQUAL (CDR (CDR G92)) 'NIL)
                        (G94)
                        NIL)
                    NIL)
                NIL))))
```

On voit bien les mutiples améliorations que l'on peut apporter à ce code. Ces améliorations peuvent être visuelles : il reste important de pouvoir lire le code engendré à la fois pour mettre au point mais aussi pour se persuader que le compilateur marche bien. Les améliorations peuvent aussi supprimer des portions de code inatteignables, éliminer des sous-expressions communes ...

Dans le fragment précédent, on peut remarquer la forme `progn` à un seul paramètre, les formes `if` emboitées avec un `nil` en troisième paramètre. Ces deux défauts ne sont valables que pour la lecture car un compilateur Lisp sous-jacent pourrait très bien normaliser le code en une autre syntaxe qui nous semblerait peut-être lourde à lire.

On peut aussi remarquer le test `equal` avec `nil` qui sera plus efficacement converti en un `eq`, ou plus lisiblement en un `null`. La fusion des deux formes `cdr` en `cddr` serait aussi appréciable.

Toutes ces remarques militent donc vers une redéfinition des générateurs de base afin d'améliorer leur efficacité et lisibilité. Le premier à être ainsi modifié est :

si

```
(def-match-function si "Si"
  (?pred nil t                => `(not ,pred))
  (?pred t ?sinon             => (ou pred sinon))
  (?pred ?alors nil           => (et pred alors))
  (?pred1 (if ?pred2 ?alors ?sinon) ?sinon
                   => (si (et pred1 pred2) alors sinon) )
  ((*or t 't) ?alors ?-       => alors)
  (nil ?- ?sinon              => sinon)
  ((not ?pred) ?alors ?sinon  => (si pred sinon alors))
  (?pred ?alors ?sinon
      => `(if ,pred ,alors ,sinon)) )
```

Les améliorations portant sur les conditionnelles sont purement syntaxiques et visent à supprimer les termes inesthétiques et diminuer l'enclavage parenthétique. Les générateurs `et` et `ou` se livrent à la même gymnastique mais avec une contrainte nouvelle, qui est de respecter l'ordre et la sémantique d'évaluation. En effet, `(et (#:g23) nil)` n'est pas simplifiable en `nil` car ce serait nier l'évaluation du premier terme qui peut provoquer un échappement. On peut seulement dans ce dernier cas constater l'inanité de la valeur retournée par `(#:g23)` et simplifier l'expression totale en `(progn (#:g23) nil)`. Les générateurs `et` et `ou` font aussi jouer une intéressante source d'amélioration qui est la mise en facteur de préfixes communs. L'expression `(or (and` *qqch* `...) (and` *qqch* `...))` peut être raccourcie en `(progn` *qqch* `(or (and ...) (and ...)))`, ce qui à la fois diminue la taille du code et améliore son efficacité.

super-et

```
(def-match-function super-et "Super-et"
  ((??d (and ???m) ???e)
        => `(,@d ,@m ,@e) )
  ((?exp) => exp)
  (?exps  => `(and . ,exps)) )
```

et

```
(def-match-function et "Et"
  (t ?b                    => b)
  (?a nil                  => (faire a nil))
  (?a t                    => a)
  (nil ?-                  => 'nil)
  ((and ??aa) (and ??bb)   => (super-et `(,@aa ,@bb)))
  (?a (and ??bb)           => (super-et `(,a ,@bb)))
  ((and ??aa) ?b           => (super-et `(,@aa ,b)))
  (?a ?d                   => `(and ,a ,d)) )
```

ou

```
(def-match-function ou "Ou"
  (?a nil                 => a)
  (?a t                   => (faire a t))
  (t ?-                   => 't)
  (nil ?b                 => b)
  ((or ??aa) (or ??bb)    => `(or ,@aa ,@bb))
  ((or ??aa) ?b           => `(or ,@aa ,b))
  (?a (or ??bb)           => `(or ,a ,@bb))
  ((and (*and ?a ((*or eq equal null consp setq) ???-))
        ???bb )
   (and ?a ???aa)
            => (et a (ou (super-et bb) (super-et aa))))
  ((and (progn (*and ?a
                    ((*or eq equal null consp setq) ???-))
                    ???bb) ???cc)
   (and (progn ?a ???aa) ???dd)
```

```
          => `(progn ,a
                ,(ou (super-et `(,(e-faire bb) . ,cc))
                     (super-et `(,(e-faire aa) . ,dd)) ) ) )
   ((progn (*and ?a
                 ((*or eq equal null consp setq) ???-))
                 ?b1 ???cc)
    (progn ?a ?b2 ???aa)
          => (faire a (ou (e-faire `(,b1 . ,cc))
                          (e-faire `(,b2 . ,aa)) )) )
   (?a ?d                 => `(or ,a ,d)) )
```

Pour `egal` on se contentera de "dégénérifier" `equal` en un prédicat plus approprié, plus lisible et plus efficace.

```
 (def-match-function egal "Egal"                                    egal
   (?e 'nil  =>        `(null ,e))
   (?e (*and ?f '(*check symbolp))
             =>        `(eq ,e ,f) )
   (?e ?f    =>        `(equal ,e ,f)) )
```

Les accesseurs `le-car` et `le-cdr` agglutinent les `car` et `cdr` et ainsi améliorent la lisibilité. Les clauses concernent aussi `list` qui sert à empaqueter les arguments des fonctions définies par filtrage et d'arité fixe. Si l'aspect visuel est grandement amélioré, il est en revanche dubitable que soit améliorée la compilation de ces formes qui, de toute manière, auront à suivre les combinaisons de `car` et de `cdr` indiquées.

```
 (def-match-function le-car "Le-CAR"                                le-car
   ((cdr ?e)      => `(cadr ,e))
   ((car ?e)      => `(caar ,e))
   ((cddr ?e)     => `(caddr ,e))
   ((cadr ?e)     => `(caadr ,e))
   ((cdar ?e)     => `(cadar ,e))
   ((caar ?e)     => `(caaar ,e))
   ((caaar ?e)    => `(caaaar ,e))
   ((caadr ?e)    => `(caaadr ,e))
   ((list ?a ???-) => a)
   ((list)
      => (filtre6-erreur
           "MATCH : Erreur interne dans Le-car." ) )
   (?e            => `(car ,e)) )
```

```
 (def-match-function le-cdr "Le-CDR"                                le-cdr
   ((cdr ?e)       => `(cddr ,e))
   ((car ?e)       => `(cdar ,e))
   ((cddr ?e)      => `(cdddr ,e))
   ((cadr ?e)      => `(cdadr ,e))
   ((cdar ?e)      => `(cddar ,e))
   ((caar ?e)      => `(cdaar ,e))
   ((caaar ?e)     => `(cdaaar ,e))
   ((caadr ?e)     => `(cdaadr ,e))
   ((list ?- ???d) => `(list . ,d))
   ((list)
      => (filtre6-erreur
           "MATCH : Erreur interne dans Le-cdr." ) )
   (?e             => `(cdr ,e)) )
```

On remarquera dans ce qui précède quelques règles de simplification comme `(car (list` α β`...))` en α dont l'esprit est hérité de la méta-évaluation symbolique [Goossens 80].

Enfin `faire` élimine quelques évaluations constantes et aplatit les formes `progn`.

```
 (def-match-function faire "Faire"                                  faire
   ((?arg)                 arg)
```

```
  ((t ???args)            (e-faire args))
  ((nil ???args)          (e-faire args))
  (((quote ?-) ???args)   (e-faire args))
  ((??d (progn ???m) ???e)  (e-faire `(,@d ,@m . ,e)))
  (?arg                   `(progn . ,arg)) )
```

Enfin, pour faire bon poids, voici deux autres améliorations :

affecter

```
(defun affecter (variable valeur)
  (if (eq variable valeur) 'NIL
      `(setq ,variable ,valeur) ) )
```

sechapper

```
(def-match-function sechapper "S'echapper"
  ( ?- (*and ?f (return-from ?- ?-))
    => f )
  ( ?- (*and ?f (progn ??- (return-from ?- ?-)))
    => f )
  ( ?label ?forme
    => `(return-from ,label ,forme) ) )
```

Toutes les précédentes fonctions se lisent simplement car ce ne sont que de triviales analyses de cas suivies d'expansion non moins simplistes. En revanche le résultat est loin d'être trivial car le généré des filtres est bien plus lisible et efficace.

Ce ne sont que les futures compilations qui bénéficieront des ces nouveaux générateurs et donc de ces nouveaux avantages. Rassurez-vous, tout sera recompilé mais auparavant nous raît quelques autres touches à apporter à l'ensemble.

Ceci clora les améliorations que nous pouvons apporter au code. Toutes auraient pu être effectuées par un compilateur sous-jacent très intelligent, ce qui serait très malin car cela pourrait s'exercer sur tout programme et non sur les seuls générés de filtres. Mais parier sur l'intelligence d'un compilateur relève d'un acte de foi qu'il est bon de ne pas commettre sur toute implantation. Aussi avons-nous pris le parti d'améliorer ce que l'on peut, c'est toujours cela de pris !

5.11.2 Les améliorations sur la normalisation

Un large domaine n'a pas encore été abordé, c'est celui de l'amélioration des filtres normalisés. Il est en effet simple de voir que :

(**or2 (**cons (**sexp) $filtre_1$)
       (**cons (**sexp) $filtre_2$))

n'est autre que

(**cons (**sexp) (**or2 $filtre_1$ $filtre_2$))

Ce sont ces règles que nous allons mettre en œuvre dans une phase dite d'amélioration des filtres.

Nous employons systématiquement le terme d'*amélioration* plutôt que d'optimisation car nous n'avons pas spécifié de critère vis-à-vis duquel nous optimisons. Comme les transformations auxquelles nous nous livrons sont elles-mêmes améliorables, nous nous gardons donc d'utiliser ce vocable.

En fait les améliorations les plus significatives portent sur trois filtres seulement et couvrent un large spectre qui, complété des améliorations de génération de code déjà expliquées, traite correctement tous les exemples sensibles de ce livre. On pourra apprécier dans le florilège qui clôt ce chapitre que, si l'on peut encore affiner les résultats, les moyens à mettre en œuvre pour ce faire augmentent singulièrement en complexité.

Les améliorations sont mises en œuvre par des méthodes spécialisées sur les filtres rudimentaires. Par défaut, il n'y a pas d'amélioration.

ameliorer6

```
(defun ameliorer6 (f env)
  (let ((filtre (obtenir6filtre (car f))))
    (unless filtre
      (error "Je ne connais pas de filtre nomme ~S." (car f)))
```

```
    (funcall (or (rudimentaire6-amelioration filtre)
                 #'(lambda (f env) f) )
             f env ) ) )
```

La première amélioration porte sur `**quote` et vise à ne pas permettre des constantes de type liste et donc à les décomposer en leur `car` et `cdr`.

```
(de-filtre6 (**quote expression) rudimentaire6                    **quote
  :amelioration
    (match-case expression
      ( (*cons ?a ?d)
        `(**cons ,(ameliorer6 `(**quote ,a) env)
                 ,(ameliorer6 `(**quote ,d) env) ) )
      ( ?- `(**quote ,expression)) ) )
```

Compte tenu du fait que les filtres ne recèlent que des données de faible taille, il est plus intéressant de décomposer les paires pointées et de les combiner si nécessaire. L'augmentation de volume de code généré est alors faible si l'on compare `(equal e '(foo bar))` et `(and (consp e) (eq (car e) 'foo) (consp (cdr e)) (eq (cadr e) 'bar) (null (cddr e)))`. On a en fait mué une donnée `(foo bar)` en du code exécutable.

Les autres améliorations portent sur les conjonctions et disjonctions.

```
(de-filtre6 (**and2 a d) rudimentaire6                             **and2
  :amelioration
    (peaufiner6and a d env) )
```

```
(de-filtre6 (**or2 a d) rudimentaire6                               **or2
  :amelioration
    (peaufiner6or a d env) )
```

Tout le savoir-faire est concentré en les fonctions `peaufiner6and` et `peaufiner6or`. Un effet nouveau apparaît, ces améliorations dépendent de la macro d'invocation ou plus exactement de l'éventuelle multiplicité des solutions souhaitées. Par exemple le filtre `(**or2` ϕ ϕ`)` est simplifiable en ϕ si au plus une seule solution est espérée tandis qu'il doit demeurer tel qu'il est lorsque l'on désire énumérer toutes les solutions. Cet effet joue sur de nombreuses règles.

Les règles qui suivent usent de filtres non triviaux, comportant de nombreuses conditions et sont de ce fait assez durs à lire. Le résultat est qu'un langage de filtres plus lisible que celui-ci reste encore à inventer. Il est toutefois gênant d'introduire en Lisp des sous-langages qui ne soient pas compatibles c'est-à-dire non intimement couplables à Lisp. Le problème reste donc ouvert d'inventer une syntaxe agréable, lisible pour spécifier harmonieusement des filtres.

Voici donc les règles concernant la conjonction.

```
(def-match-function peaufiner6and                             peaufiner6and
  ;;Distribuer les **and2 sur les **or2 en premier argument
  ((?'**or2 ?y ?z) ?x ?env
     => (ameliorer6 `(**or2 ,(ameliorer6 `(**and2 ,y ,x) env)
                            ,(ameliorer6 `(**and2 ,z ,x) env))
                    env ) )
  ;;Distribuer les **and2 sur les **or2 en second argument
  ;;mais que si le premier argument est trivial (ici un **quote)
  ((*and ?x (?'**quote ?-)) (?'**or2 ?y ?z) ?env
     => (ameliorer6 `(**or2 ,(ameliorer6 `(**and2 ,x ,y) env)
                            ,(ameliorer6 `(**and2 ,x ,z) env))
                    env ) )
  ;;repousser les **and2 imbriqués en second argument
  ((?'**and2 ?y ?z) ?x ?env
     => (ameliorer6 `(**and2 ,y
                           ,(ameliorer6 `(**and2 ,z ,x) env))
                    env ) )
  ;;(*and2 (*sexp) φ) ≡φ
```

```
  (?'(**sexp) ?x ?-   => x)
  ;;(*and2 φφ) ≡φ si unique solution
  (?x ?x  ?-
     => (ecase (dynamic-ref *match-lambda-mode*)
           (match-lambda x)
           (match-all-lambda `(**and2 ,x ,x)) ) )
  ;;(*and2 φ(*sexp)) ≡φ
  (?x ?'(**sexp) ?-   => x)
  ;;(*and A B) est impossible
  ((?'**quote ?exp) (?'**quote (*not ?exp)) ?-
     => `(**not (**sexp)) )
  ;;de même que (*and A (B . C))
  ((?'**quote ?-) (?'**cons ?- ?-) ?-
     => `(**not (**sexp)) )
  ;;et que (*and (A . B) C))
  ((?'**cons ?- ?-) (?'**quote ?-) ?-
     => `(**not (**sexp)) )
  ;;(*and (*fail) φ) ≡(*fail)
  (?'(**not (**sexp)) ?-  ?- => `(**not (**sexp)))
  ;;Distribuer les **and2 sur les **cons
  ((?'**cons ?a1 ?d1) (?'**cons ?a2 ?d2) ?env
     => (ameliorer6
          `(**cons ,(ameliorer6 `(**and2 ,a1 ,a2) env)
                   ,(ameliorer6 `(**and2 ,d1 ,d2) env) )
          env ) )
  ;;Repousser les **success en second argument de **cons
  ((?'**cons ?a1 ?d1) (*and ?s (?'**success ?-)) ?env
     =>`(**cons ,a1 ,(ameliorer6 `(**and2 ,d1 ,s) env)) )
  ;;On ne réchappe pas d'un succès par échappement
  ((*and ?s (?'**success (return-from ??-))) ?- ?-
     => s )
  ;;Remonter les **setq de second argument (mais pas de premier)
  (?x (?'**setq ?nom ?f) ?env
     => `(**setq ,nom ,(ameliorer6 `(**and2 ,x ,f) env)) )
  ;;Et si tout ce qui précède a échoué
  (?a ?d  ?-     =>     `(**and2 ,a ,d)) )
```

Le cas de la disjonction est un peu plus simple.

peaufiner6or

```
(def-match-function peaufiner6or
  ;;On ne réchappe pas d'un succès par échappement en premier argument
  ((*and ?s (?'**success (return-from ??-))) ?- ?-
      => s )
  (?'(**sexp) ?d ?-
      => (ecase (dynamic-ref *match-lambda-mode*)
             (match-lambda `(**sexp))
             (match-all-lambda `(**or2 (**sexp) ,d)) ) )
  ;;(*or φφ) ≡φ) si solution unique
  (?x ?x ?-
      => (ecase (dynamic-ref *match-lambda-mode*)
           (match-lambda x)
           (match-all-lambda `(**or2 ,x ,x)) ) )
  ;;(*or (*fail) φ) ≡φ)
  (?'(**not (**sexp)) ?x ?-  =>  x)
  ;;Distribuer les **or2 sur les **cons
  ((?'**cons ?a ?d1) (?'**cons ?a ?d2) ?env
      => `(**cons ,a ,(ameliorer6 `(**or2 ,d1 ,d2) env)) )
  ;;Remonter les car communs
  ((?'**and2 (?'**cons ?a ?d1) ?s1)
```

```
     (?'**and2 (?'**cons ?a ?d2) ?s2) ?env
         => `(**cons
              ,a ,(ameliorer6
                    `(**or2 ,(ameliorer6 `(**and2 ,d1 ,s1) env)
                            ,(ameliorer6 `(**and2 ,d2 ,s2) env))
                    env )) )
    ;;Remonter les cdr communs
    ((?'**cons ?a1 ?d) (?'**cons ?a2 ?d) ?env
         => `(**cons ,(ameliorer6 `(**or2 ,a1 ,a2) env) ,d) )
    ;;En désespoir de cause
    ( ?a ?d ?-   => `(**or2 ,a ,d)) )
```

Voici achevées les règles d'amélioration concernant les filtres normalisés. Elles suffisent dans la majorité des cas et sont capables de certaines prouesses qui seront dévoilées dans le florilège qui suit.

Résumé

Voici donc achevée la description du compilateur de filtres avec ses deux phases et les améliorations afférentes.

L'élaboration du compilateur profite d'elle-même puisque toutes les améliorations sont réalisées par filtrage.

Le résultat est un jeu de macro-fonctions non triviales puisque mettant en œuvre un véritable petit compilateur dont on a pu voir les tripes exposées.

Exercice 5.21 : Au lieu de revenir bêtement en arrière lors d'un échec, il serait judicieux de transférer de l'information quant aux causes de l'échec. En effet, et pour un filtre tel que `(??x fin de partie)`, dès que le segment initial `x` est trop long, il est inutile de chercher à l'allonger derechef. Cette utilisation astucieuse des raisons d'échec est l'essence même du retour arrière intelligent (*clever backtrack*) commun en Prolog. L'exercice consiste à améliorer le compilateur dans ce sens.

Florilège 6

De toutes les représentations de la béatitude imaginées par les peuples et les poètes, la plus grande et la plus profonde m'a toujours semblé être celle que l'on éprouve en écoutant l'harmonie des sphères. Mon rêve le plus intime, le plus précieux fut d'entendre, pour la durée d'un battement de cœur, résonner dans leur harmonie secrète toute la structure de l'Univers et toutes les manifestations de la vie.

HERMANN HESSE, *Gertrude*

Les temps sont venus d'exhiber quelques belles compilations prouvant que les 121 pages précédentes ont bien conduit quelque part. Les améliorations qui furent citées sont déjà des exemples consistants de filtrage mais sans doute trop importants en taille pour que même l'auteur ait regardé précisément ce qui avait été engendré, ce qui prouve au passage la confiance accordée au compilateur. Une erreur sournoise dans le compilateur pourrait engendrer une non moins sournoise erreur dans les améliorations, qui elle-même pourrait engendrer d'encore plus occultes dysfonctionnements. Mais foin de philosophie, passons au pétulant bouquet final !

A — Le filtre `(??- ?qui aime ?- ???-)`, cité page 26, qui extrait le mot qui précède le mot `aime` tout en assurant qu'au moins un autre mot le suit, conduirait à

```
(match-lambda (??- ?qui aime ?- ???-) qui)  ≡
(LAMBDA (G49)
  (BLOCK G47
    (MACROLET ((G52 NIL '(RETURN-FROM G47 QUI)))
      (LET (QUI)
        (LABELS ((G50 (G51)
                   (AND (CONSP G51)
                        (OR (PROGN
                              (SETQ QUI (CAR G51))
                              (AND (CONSP (CDR G51))
                                   (EQ (CADR G51) 'AIME)
                                   (CONSP (CDDR G51))
                                   (G52)))
                            (G50 (CDR G51))))))
          (G50 G49))))))
```

B — La fonction `analyse-match-lambda` (page 108) correspondait au filtre

```
((*ssetq doc-string (*maybe (*check stringp)))
 (*setq filtre (*sexp))
```

```
(*ssetq faire (*sexp) (*any)) )
```

On remarquera l'écriture `(*ssetq doc-string (*maybe (*check stringp)))` qui permet dans tous les cas de définir la variable `doc-string`, même si la chaîne de documentation est absente. La disjonction présente dans `*maybe` est équilibrée ce qui ne serait pas le cas de l'écriture a priori équivalente `(*maybe (*setq doc-string (*check stringp)))`.

Voici donc l'expansion :

```
(LAMBDA (G96)
   (BLOCK G94
      (MACROLET
         ((G07 ()
            '(LET ((DOC-STRING
                    (EXECUTER6OTER DOC-STRING G98) ))
               (RETURN-FROM ANALYSE-MATCH-LAMBDA
                   (VALUES (WHEN DOC-STRING
                                 (CAR DOC-STRING) )
                           FILTRE
                           FAIRE) ) ) ))
        (LET (FAIRE G97 FILTRE DOC-STRING G98)
          (PROGN
           (SETQ DOC-STRING G96)
           (FLET ((G99 (G00)
                   (PROGN
                    (SETQ G98 G00)
                    (AND
                     (CONSP G00)
                     (PROGN
                      (SETQ FILTRE (CAR G00))
                      (SETQ FAIRE (CDR G00))
                      (FLET ((G03 (G04)
                               (AND (NULL G04)
                                    (G07))))
                       (AND
                        (CONSP (CDR G00))
                        (LABELS
                           ((G01 (G02)
                             (OR (G03 G02)
                                 (AND (CONSP G02)
                                      (G01 (CDR G02))))))
                          (G01 (CDDR G00)))))))))))
          (OR (AND (CONSP G96)
                   (STRINGP (CAR G96))
                   (G99 (CDR G96)))
              (G99 G96))))))))
```

C — Les exemples les plus durs sont les fonctions définies par filtrage et dotées de multiples clauses où les travaux d'amélioration sont les plus cruciaux. Voici `nodup`, exemple extrait de [Peyton Jones 86] et ici traité à la Lisp.

```
(def-match-function nodup
  ( ()           => ())
  ( (?x)         => `(,x))
  ( (?x ?x ???y) => (nodup `(,x . ,y)))
  ( (?x ???y)    => `(,x . ,(nodup y))) )
```

Voici le code engendré :

```
(LAMBDA (G19)
   (BLOCK G18
```

```
(MACROLET
  ((G46 NIL '(LET ((Y G20)
                  (X G22))
              (RETURN-FROM NODUP
                 (CONS X (NODUP Y)))))
   (G39 NIL '(LET ((Y G20)
                  (X G22))
              (RETURN-FROM NODUP
                 (NODUP (CONS X Y)))))
   (G32 NIL '(LET ((X G22))
              (RETURN-FROM NODUP (LIST X))))
   (G25 NIL '(RETURN-FROM NODUP NIL)))
  (LET (G20 G21 G22)
    (OR (AND (NULL G19) (G25))
        (AND (CONSP G19)
             (PROGN
              (SETQ G22 (CAR G19))
              (OR (AND (NULL (CDR G19)) (G32))
                  (AND (CONSP (CDR G19))
                       (EQUAL (CADR G19) G22)
                       (LISTP (CDDR G19))
                       (PROGN
                        (SETQ G20 (CDDR G19))
                        (G39)))
                  (AND (LISTP (CDR G19))
                       (PROGN
                        (SETQ G20 (CDR G19))
                        (G46)))))))))))
```

Il est évident que dans ce genre de définition l'ordre des clauses importe plus que peu puisqu'il conditionne les possibilités de combinaisons que peuvent réaliser les améliorations des deux niveaux.

D — La fonction précédente était unaire et de ce fait plus simple à compiler que la suivante qui est ternaire. Il s'agit de `mappairs` toujours extraite de [Peyton Jones 86].

```
(def-match-function mappairs
  ( ?fn () ?-               => ())
  ( ?fn ?- ()               => ())
  ( ?fn (?x ???y) (?xx ???yy) =>
     `(,(funcall fn x xx) . ,(mappairs fn y yy))) )
```

Voici le code engendré :

```
(LAMBDA (G26 G25 G24)
   (BLOCK G23
      (MACROLET
         ((G45 NIL
              '(RETURN-FROM MAPPAIRS
                    (CONS (FUNCALL FN X XX)
                          (MAPPAIRS FN Y YY) )))
          (G38 NIL '(RETURN-FROM MAPPAIRS NIL))
          (G31 NIL '(RETURN-FROM MAPPAIRS NIL)))
        (LET (YY G27 XX Y G28 X FN)
          (PROGN
           (SETQ FN G26)
           (OR (AND (NULL G25) (G31))
               (AND (NULL G24) (G38))
               (AND (CONSP G25)
                    (PROGN
                     (SETQ X (CAR G25))
```

```
                  (AND
                   (LISTP (CDR G25))
                   (PROGN
                    (SETQ Y (CDR G25))
                    (AND
                     (CONSP G24)
                     (PROGN
                      (SETQ XX (CAR G24))
                      (AND
                       (LISTP (CDR G24))
                       (PROGN
                        (SETQ YY (CDR G24))
                        (G45))))))))))))))
```

E — N'escamotons pas quelques belles réalisations comme l'expansion de `(match-lambda (*and hux wrek) t)` qui mène à

```
(LAMBDA (G96) (BLOCK G95 (MACROLET NIL NIL)))
```

On peut bien constater ici la propagation de l'échec.

F — Un joli cas d'école maintenant, un peu plus compliqué que le célèbre `(??x ??x)` et redoutable à programmer à la main :

```
(match-lambda (??x ??x ??x) t)
```

Avant que n'apparaisse l'expansion, réfléchissez trente secondes sur la façon dont vous programmeriez cette fonction directement ... En guise de question subsidiaire, vous pouvez aussi vous interroger sur ce qui va être engendré par le compilateur de filtres. Écrivant ces lignes et n'ayant pas encore soumis l'expression, je n'imagine même pas ce qui va être produit bien que je connaisse tous les trucs de cette compilation. Voici donc l'outil dépassant son maître sur des exemples non triviaux. Voici donc l'expansion :

```
(LAMBDA (G58)
  (BLOCK G56
    (MACROLET
       ((G68 NIL '(LET ((X (EXECUTER6OTER X G59)))
                    (RETURN-FROM G56 T) )))
      (LET (X G59)
        (PROGN
         (SETQ X G58)
         (FLET ((G62 (G63)
                  (PROGN
                   (SETQ G59 G63)
                   (MULTIPLE-VALUE-BIND (G65 G64)
                      (EXECUTER6VERIFIER-2LISTE G63 X G59)
                    (AND G65
                         (MULTIPLE-VALUE-BIND (G67 G66)
                            (EXECUTER6VERIFIER-2LISTE
                              G64 X G59 )
                          (AND G67
                               (NULL G66)
                               (G68)))))))
      (LABELS ((G60 (G61)
                 (OR (G62 G61)
                     (AND (CONSP G61)
                          (G60 (CDR G61))))))
       (G60 G58))))))))
```

Pas mal, non ?

La forme **flet** introduit une fonction locale qui vérifie si la liste qui lui est soumise est une copie de `X` suivie de `X`. La fonction introduite par **labels** donne à `X` des longueurs successives croissantes. Notez qu'il n'y a qu'un seul endroit de succès car la variable `X` est toujours définie par ce filtre même confronté à la liste vide.

G — Un dernier exemple relatif cette fois à **match-all-lambda**. Soit à extraire tous les segments d'au moins deux termes figurant au moins deux fois dans une liste.

```
(funcall (match-all-lambda
              (??- (*ssetq x ?- ?- ??-) ??- ??x ???-)
            (print `(X = ,X)) )
         '(q u e j a i m e a f a i r e a p p r e n d r e
           u n n o m b r e u t i l e a u x s a g e s))
```

Voici donc l'expansion :

```
(LAMBDA (G77)
  (BLOCK G73
    (MACROLET
        ((G89 NIL '(LET ((X (EXECUTER6OTER X G78)))
                     (PROGN (PRINT (CONS 'X
                                       (CONS '= (LIST X))))
                            NIL))))
       (LET (X G78)
         (LABELS
           ((G79 (G80)
             (OR
              (PROGN
               (SETQ X G80)
               (FLET
                ((G83 (G84)
                  (PROGN
                   (SETQ G78 G84)
                   (LABELS
                     ((G85 (G86)
                        (OR (MULTIPLE-VALUE-BIND (G88 G87)
                               (EXECUTER6VERIFIER-2LISTE
                                  G86 X G78 )
                             (AND G88
                                  (PROGN (G89) NIL)))
                           (AND (CONSP G86)
                                (G85 (CDR G86))))))
                    (G85 G84)))))
                 (AND (CONSP G80)
                      (CONSP (CDR G80))
                      (LABELS ((G81 (G82)
                                  (OR (G83 G82)
                                      (AND (CONSP G82)
                                           (G81
                                            (CDR G82))))))
                        (G81 (CDDR G80))))))
                (AND (CONSP G80)
                     (G79 (CDR G80))))))
        (G79 G77))))))
```

Et voici les résultats :

```
(X = (A I))
(X = (E A))
(X = (E A))
```

```
(X = (R E))
(X = (R E))
(X = (R E))
(X = (E A))
(X = (R E))
(X = (R E))
(X = (R E))
(X = (R E U))
(X = (E U))
NIL
```

Résumé

La programmation par filtrage est simple et efficace. Les filtres épousent la forme des arguments et sont donc simples à lire et à écrire.

Toutefois une syntaxe plus agréable que celle ici présentée reste à inventer. Notez que cela n'affecterait que la phase de normalisation.

7 De l'intérêt du filtrage

L'acte d'écrire demande toujours un certain "sacrifice de l'intellect". On sait bien, par exemple, que les conditions de la lecture littéraire sont incompatibles avec une précision excessive du langage. L'intellect volontiers exigerait du langage commun des perfections et des puretés qui ne sont pas en sa puissance. Mais rares sont les lecteurs qui ne prennent leur plaisir que l'esprit tendu.

PAUL VALÉRY, *Monsieur Teste*

À quoi peut bien servir le filtrage ? Cette question, d'autant plus angoissante qu'elle survient à plus de 127 pages du début, mérite d'être considérée bien qu'une réponse soit déjà apparente dans tous les programmes précédemment exposés.

Le filtrage analyse des formes, compare des contenus, extrait des parties. Ces trois effets se conjuguent dans une description unique : le filtre. La mise en œuvre de ce dernier peut conduire à des solutions multiples.

Historiquement les données préférentiellement manipulées par les filtreurs sont des listes. La tendance moderne est d'utiliser des structures, qui disposent de représentation compacte en mémoire, d'accès rapide et qui, en outre, procurent certains avantages de la programmation par objets. La plupart des exemples de filtrage présents dans les manuels (Doctor, Eliza) ignorent les solutions multiples et se contentent de la solution gauche courte. À la fois les filtres et les expressions qui seront filtrées sont écrits de manière à ce qu'il n'y ait pas de multiples solutions. Enfin, le réflexe quasi génétique du `progn` implicite pousse à ce que les filtres de segments nommés soient en position terminale et donc immédiatement résolus sans retour arrière.

Si l'on relit les programmes de `filtrer6` à la lueur de ces indications, il apparaît bien que le filtrage sert principalement aux macros pour analyser leur forme d'appel (une liste) et la déstructurer de manière appropriée. Le filtrage se recommande particulièrement pour cet usage mais aussi pour une foule de fonctions comme ont pu le démontrer les améliorateurs. Le filtrage devient aussi, et de plus en plus, un trait des langages fonctionnels (ML) ou autres (Modula 3) autorisant des discriminations sur les types de données reçues plus que sur les valeurs de leurs champs.

Étant donné un filtre, un moteur à filtrer s'attellera sans préjugé à la vérification fastidieuse de toutes les informations que recèle ce filtre. L'écriture à la main des programmes équivalents conduit le paresseux qui sommeille en tout (?) programmeur à user d'a priori, d'approximation et de court-circuits qui, à la longue, peuvent s'avérer dangereux. Considérez, par exemple, ces deux définitions

```
(defmacro faire (&whole forme)
  `(progn . ,(cdr forme)) )
                  (def-match-macro (faire ??formes)
                     `(progn . ,formes) )
```

Appliquées à la forme d'appel suivante, elles révèleront leur différence.

```
(faire ceci puis . cela)
```

Dans le premier cas est engendrée la forme syntaxiquement erronée `(progn ceci puis . cela)`tandis que cette difformité est explicitement signalée dans le second cas. On peut donc, sous un même volume, programmer plus sûrement et écrire en toute sécurité On ne se soucie pas du code engendré et l'on peut se concentrer sur l'essence même de ce qu'on l'on imagine : la forme pure !

Le filtrage semble donc ne procurer qu'une façon agréable d'engendrer du code. L'emploi de filtres de segments nommés est généralement limité à leur liaison, sans aucune référence postérieure. La programmation en Lisp est un processus où l'on ne requiert habituellement nulle redondance de la part du programmeur, ce qui rend peu probable des filtres comme `(??defs ??defs)`. Le moteur à filtrer précédemment présenté est donc beaucoup plus riche que quotidiennement nécessaire : il n'exclut par contre point que soient découverts de nouveaux emplois. En revanche, la quasi-totalité des filtrages courants n'étant que des déstructurations avec vérifications, ils peuvent être mis en œuvre par des moteurs plus réduits.

Nous voici donc arrivés au terme de notre aventure, pourvus d'un moteur assez élaboré et riches d'une expérience programmatique buissonnée sur de nombreuses terres. Le résultat, nous semble-t'il, valait le voyage et mérite que l'on s'en remémore certains bonheurs ou accidents. L'auteur lui-même peut faire part des nombreuses joies qu'il a obtenues chemin faisant. La plus grande est probablement la foi qu'il a placée dans le moteur interprétatif à double continuation : c'est une bonne structure. Même aux pires moments de l'élaboration du compilateur (par exemple, la normalisation du filtre `*times` ou la compilation de `*success`) où d'inavouées tentatives de solutions rompaient l'architecture sous-jacente, l'épiphanie est venue, qui a confirmé la voie et la foi. L'application du compilateur à certaines parties de lui-même procura un autre puzzle, une autre joie que de débrouiller comment amorcer le moteur. Si enfin le résultat et les techniques qui permirent d'y aboutir étoffent votre arsenal personnel et vous encouragent à expérimenter de nouveaux styles, vous nous offrirez alors une dernière et profonde joie.

Bibliographie 8

Puis l'âge aidant, à la faveur des expériences accumulées, il apparut qu'ils prenaient un peu de champ à l'égard de leurs ferveurs les plus exacerbées. Ils surent attendre, et s'habituer. Leur goût se forma lentement, plus sûr, plus pondéré. Leurs désirs eurent le temps de mûrir; leur convoitise devint moins hargneuse.

GEORGES PEREC, *Les Choses*

[Abelson & Sussman 85] Harold Abelson, Gerald Sussman, avec Julie Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge MA, 1985.

[Arcangeli & Pomian 88] Jean-Paul Arcangeli, Christian Pomian, *Compilation de Plasma : les filtres*, Bigre+Globule 59, Avril 1988, pp 140–149.

[Arcangeli & Pomian 90] Jean-Paul Arcangeli, Christian Pomian, *Principles of Plasma Pattern and Alternative Structure Compilation*, Theoretical Computer Science, Vol 71, 1970, pp 177-191.

[Augustsson 85] Lennart Augustsson, *Compiling Pattern Matching*, Conference on Functional Programming and Machine Architecture, Nancy 1985, Lecture Note in Computer Science 201, Springer Verlag 1985.

[Barthes 77] Roland Barthes, *Fragments d'un discours amoureux*, Collection Tel Quel, Éditions du Seuil, 1977.

[Bawden & Burke & Hoffman 81] Alan Bawden, Glenn S. Burke, Carl W. Hoffman, *MacLisp Extensions*, MIT/LCS/TM-203, Juillet 1981.

[Beckman, Haraldson, Oskarsson, Sandewall 76] Lennart Beckman, Anders Haraldson, Östen Oskarsson et Erik Sandewall, *A Partial Evaluator, and Its Use as a Programming Tool*, Artifical Intelligence, Vol 7, 1976, pp 319–357.

[Ben Jelloun 87] Tahar Ben Jelloun, *La nuit sacrée*, Éditions du Seuil, Paris, 1987.

[Biotteau & Danvy 86] Gabrielle Biotteau, Olivier Danvy, *Matching Patterns Improves Pattern Matching*, Rapport LITP 86–21, Avril 1986.

[Bobrow *et al.* 86] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, Frank Zdybel, *Common Loops: Merging Lisp and Object-Oriented Programming*, OOPSLA 1986.

[Bobrow *et al.* 88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales et David A. Moon, *Common Lisp Object System Specification*, in [Steele 90].

[Borges 49] Jorge Borges, *Labyrinthes*, Gallimard, 1965.

[Chailloux 86] Jérôme Chailloux, *Le_Lisp Version 15.2, Le Manuel de Référence.* INRIA, Domaine de Voluceau, Rocquencourt, 78153 Le Chesnay Cedex, France, 1986.

[Christophe 04] Christophe, *La famille Fenouillard*, Armand Colin, 1904.

[Danvy 90] Olivier Danvy, *Semantics-Directed Compilation of Non-Linear Patterns*, Technical Report 303, Computer Science Dept, Indiana University, Janvier 1990.

[Delacour & Séniak 89] Vincent Delacour, Nitsan Séniak, *Picolo, Manuel de Référence*, in Écrits d'ICSLA, Rapport LIX.RR.89.02, 1989.

[Duprat & Hunot 88] Claude Duprat, Alain Hunot, *Introduction d'heuristique dans un* Pattern Matching *du deuxième ordre*, Congrès de Génie Logiciel 4, 1988.

[Durrieux & Saint-James] Jean-Louis Durrieux, Emmanuel Saint-James, *Une charpente de semi-unification*, Revue d'Informatique théorique et Applications, Vol 22, N° 2, 1988, pp 173-225.

[Emanuelson & Haraldson 80] Pär Emmanuelson, Anders Haraldson, *On Compiling Embedded Languages in Lisp*, Conference Record of the 1980 Lisp Conference, Stanford 1980, pp 208–215.

[Emanuelson 82] Pär Emanuelson, *From Abstract Model to Efficient Compilations of Patterns*, Lecture Notes on Computer Science, International Symposium on Programming, Turin, Springer-Verlag, 1982.

[Finet 84] Jean-Pierre Finet et Patrick Sallé, *L'inclusion de filtres : une approche de la création d'environnements symboliques en Plasma*, Lecture Notes in Computer Science 167, Springer-Verlag 1984.

[Fusaoka & Fujita 83] Akira Fusaoka, Hiroshi Fujita, *Lisp Programming Using Ellipsis Notation*, Journal of Information Processing, Vol 6, N° 2, 1983.

[Gengler 88] Marc Gengler, *Un algorithme optimisé de semi-unification du second ordre sur arborescences*, Bigre+Globule 59, Avril 1988, pp 238–247.

[Giannesini *et al.* 85] F. Giannesini, H. Kanoui, R. Pasero, M. Van Caneghem, *Prolog*, InterÉditions, Paris, 1985.

[Gimpel 73] J.F. Gimpel, *A Theory of Discrete Patterns and Their Implementation in SNOBOL4*, Communications of the ACM, Vol 16, N° 2, février 1973.

[Goossens 80] Daniel Goossens, *La méta-évaluation au service de la compréhension automatique de programmes*, Thèse de 3ème cycle, Rapport LITP 80-52, Université Paris VI.

[Greussay 77] Patrick Greussay, *Contribution à la définition interprétative et à l'implémentation des Lambda-langages*, Thèse d'état, Université Paris VI, Novembre 1977.

[Griswold 80] Ralph E. Griswold et David R. Hanson, *An Alternative to the Use of patterns in String Processing*, ACM Transactions on Programming Languages and Systems, Vol 2, N° 2, Avril 1980, pp 153–172.

[Heckmann 88] Reinhold Heckmann, *A Functional Language for the Specification of Complex Tree Transformations*, ESOP 88, Lecture Notes on Computer Science, Springer 1988.

[Hesse 62] Hermann Hesse, *Gertrude*, Calmann-Lévy, Paris, 1962.

[Hewitt & Smith 75] C. Hewitt, B. Smith, *Towards a Programming Apprentice*, IEEE Transactions on Software Engineering, SE-1, N° 1, Mars 1975, pp 26-45.

[Hewitt 78] Carl Hewitt, *Viewing Control Structures as Patterns of Messages Passing*, Artificial Intelligence, Vol 8, 1977, pp 323–364.

[Hoffmann & O'Donnell 79] Christoph M. Hoffmann, Michael J. O'Donnell, *An Interpreter Generator using Tree Pattern Matching*, Proceedings of the 6th Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, 1979, pp 169–179.

[Hoffmann & O'Donnell 82] Christoph M. Hoffmann, Michael J. O'Donnell, *Pattern Matching in Trees*, Journal of the ACM, Vol 29, N° 1, 1982, pp 68–95.

[Hudak & Wadler 90] Paul Hudak et Philip Wadler (eds), *Report on the Programming Language Haskell*, YALEU/DCS/RR-777, 1 Avil 1990.

[Huet 76] Gérard Huet, *Résolution d'équations dans des langages d'ordre 1, 2, ..., ω*, Thèse d'état, Université Paris VII, 1976.

[Kessler 88] Robert R. Kessler, *Lisp, Objects, and Symbolic Programming*, Scott, Foresman and Company, 1988.

[Kiremitdjian & Roy 85] Georges Kiremitdjian, Jean-Pierre Roy, *Lire Lisp, le langage de l'intelligence artificielle*, CEDIC-Nathan, Paris, 1985.

[Laville 88] Alain Laville, *Évaluation paresseuse des filtrages avec priorité, application au langage ML*, Thèse de Doctorat, Université Paris VII, Février 1988.

[Masinter 78] Larry Masinter, *The Pattern Match Compiler*, in [Teitelman 78].

[Morris & Goscinny 73] Morris et Goscinny, *L'héritage de Ran-Tan-Plan*, Dargaud 1973.

[Neidl 83] Eugen Neidl, *Étude des relations avec l'interprète dans la compilation de Lisp*, Thèse de troisième cycle, Université Paris VI, 1984.

[Perec 65] Georges Perec, *Les Choses*, Julliard, Paris, 1965.

[Peyo 82] Peyo, *Schtroumpf vert et vert schtroumpf*, Dargaud 1982.

[Peyton Jones 86] Simon L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, Mai 1986.

[Proust 17] Marcel Proust, *À la Recherche du Temps Perdu*, Gallimard, Paris, 1954.

[Queinnec 82] Christian Queinnec, *Langage d'un autre type: Lisp*, Eyrolles, Paris, 1982.

[Queinnec 84] Christian Queinnec, *Lisp Mode d'Emploi*, Eyrolles, Paris, 1984.

[Queinnec 90] Christian Queinnec, *Compilation of Non-Linear, Second-Order Patterns on S-expressions*, PLILP, Linkoping, Août 1990.

[Ramesh & Ramakrishnan 90] R. Ramesh, I. V. Ramakrishnan, *Parallel Tree Pattern Matching*, Journal of Symbolic Computation, N° 9, 1990, pp 485–501.

[Rees & Clinger 86] Jonathan A. Rees, William Clinger, *Revised³ Report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, 21, 12, Décembre 86, pp 37 – 79.

[Schnoebelen 88] Philippe Schnoebelen, *Refined Compilation of Pattern-Matching for Functional Languages*, Science of Computer Programming, Vol 11, 1988, pp 133–159.

[Slagle & Gini 87] James R. Slagle, Maria L. Gini, *Pattern Matching*, pp 716–720 *in Encyclopedia of Artificial Intelligence*, Stuart C. Shapiro (ed.), Wiley Interscience, 1987.

[Steele 90] Guy L. Steele Jr., *Common Lisp, the Language*, seconde édition, Digital Press, Burlington MA, 1990.

[Strachey & Wadsworth 68] Christopher Strachey, Christopher Wadsworth, *Continuations – a Mathematical Semantics for Handling Full Jumps*, Technical Monograph PRG-11, Programming Research Group, University of Oxford, 1974.

[Teitelman 78] Warren Teitelman, *InterLisp Reference Manual*, 1978.

[Tesler, Enea, Smith 73] Lawrence G. Tesler, Horace J. Enea, David C. Smith, *The Lisp70 Pattern Matching System*, troisième IJCAI, Août 1973.

[Thom 72] René Thom, *Stabilité structurelle et morphogenèse*, InterÉditions, 1984.

[Turner 85] D.A. Turner, *Miranda™: a Non Strict Functional Language with Polymorphic Types*, in Functional Programming and Computer Architecture, Nancy, LNCS 201, Springer, 1985.

[Valéry 46] Paul Valéry, *Monsieur Teste*, Gallimard, Paris, 1946.

[Vargas Llosa 79] Mario Vargas Llosa, *La tante Julia et le scribouillard*, Édition Folio, Gallimard 1979.

[Weis 89] Pierre Weis, *The CAML Reference Manual*, INRIA 1989.

[Wertz 84] Harald Wertz, *Étude, réalisation et évaluation d'un environnement de programmation utilisant des représentations multiples pour le développement continu de logiciels très évolués*, Thèse d'état, Université Paris VIII, Rapport LITP 84-16 avril 1984.

[Wertz 85] Harald Wertz, *Lisp, une introduction à la programmation*, Masson, Paris, 1985.

[Winston & Horn 81] Patrick H. Winston, Berthold K. Horn, *Lisp*, première édition, Addison Wesley, 1981.

[Winston & Horn 84] Patrick H. Winston, Berthold K. Horn, *Lisp*, seconde édition, Addison Wesley, 1984.

[Winston & Horn 88] Patrick H. Winston, Berthold K. Horn, *Lisp*, troisième édition, Addison Wesley, 1988.

9 Solutions des exercices

Nous avons rêvé le monde. Nous l'avons rêvé solide, mystérieux, visible, omniprésent dans l'espace et durable dans le temps; mais nous avons permis dans son architecture des fissures ténues et éternelles de la déraison qui nous disent qu'il est faux.

Jorge Borgès, *Labyrinthes*

Exercice 3.1 (page 24) :
La structure de `*atom` est semblable à `*even`.

```
(de-filtre3 *atom                                              *atom
   (atom e) )
```

Exercice 3.2 (page 24) :

```
(de-filtre3 *mcons                                             *mcons
   (if (consp (cdr f))
       ;Le filtre *MCONS doit avoir au moins un filtre argument
       (if (consp (cddr f))
           (if (consp e)
               (and (filtrer3 (car e) (cadr f))
                    (funcall (get 'filtrer3 '*mcons)
                         (cdr e)
                         ;(cons '*mcons (cddr f)) est correct
                         ;mais coûteux, ce qui suit est équivalent :
                         (cdr f) ) )
               ;L'expression n'est pas un doublet
               nil )
           (filtrer3 e (cadr f)) )
       (error "*MCONS sans argument") ) )
```

Exercice 3.3 (page 24) :

```
  (de-filtre4 *vector                                          *vector
   (if (and (vectorp e)
            (eql (length e) (length (cdr f))) )
       (labels ((scan (i env)
                  (if env
```

```
                  (if (eql i (length e))
                      env
                      (scan (1+ i)
                            (filtrer4 (elt e i)
                                      (elt (cdr f) i)
                                      env ) ) ) )))
        (scan 0 env) ) ) )
```

Exercice 3.4 (page 24) :
Une solution apparaît avec le filtre rudimentaire `**struct` page 63.

Exercice 3.5 (page 24) :
L'adaptation montre le cheminement de l'environnement. On ne peut plus utiliser `some` ou `every` mais, par exemple, `reduce`.

```
(de-filtre4 *or                                                    *or
   (reduce #'(lambda (r f)
               (or r (filtrer4-element e f env)) )
           (cdr f)
           :initial-value nil ) )

(de-filtre4 *and                                                   *or
   (reduce #'(lambda (r f)
               (and r (filtrer4-element e f env)) )
           (cdr f)
           :initial-value t ) )
```

Exercice 3.6 (page 24) :
Voir par la suite la programmation de `filtrer5` 4.5.

Exercice 4.7 (page 30) :
Le filtre de segment `(*maybe` *filtre* ...`)` est équivalent à `(*oorr (*progn` *filtre* ...`) (*progn))`.

Exercice 4.8 (page 30) :
`(??-)` filtre toutes les listes et donc la liste vide `()` ou les listes non vides qui répondent à `(*cons ?- ?-)`. A l'inverse `(?-)` ne filtre que les seules listes possédant un unique terme.

Exercice 4.9 (page 30) :
`filtrer44` commence par filtrer `(f o o)` au moyen de `(??- ?x ??-)` ce qui conduit à un succès avec `x` = `f`. Cet environnement est repris pour le filtrage de `(b o r)` par `(??- ?x ??-)` qui échoue puisque `f` n'apparaît pas dans `(b o r)`. Comme on a alors perdu la possibilité d'essayer d'autres liaisons pour `x`, le filtrage échoue lamentablement.

Exercice 4.10 (page 43) :
`(?x (*not ?x))` filtre toute liste de deux termes dont le second est différent du premier tandis que `((*not ?x) ?x)` refuse tout ! En effet, si `?x` accepte tout, `(*not ?x)` refuse tout et clôt ainsi le filtrage.

Exercice 4.11 (page 45) :
`(*list (*times (*times (*sexp))))`

Exercice 4.12 (page 54) :
Non corrigé

Exercice 4.13 (page 54) :
Non corrigé.

Exercice 4.14 (page 54) :
Non corrigé, car l'oracle sera abandonné au chapitre 5.

Exercice 4.15 (page 54) :
Sera corrigé dans une prochaine édition.

Exercice 5.16 (page 59) :

```
(defmacro match-case-lambda (&rest rules)
   (let ((arg (gensym))
         (label (gensym)) )
      `#'(lambda (,arg)
           (block ,label
            ,@(mapcar
               #'(lambda (rule)
                   `(funcall (match-lambda ,(car rule)
                               (return-from ,label
                                 ,(apply #'faire (cdr rule))))
                             ,arg ) )
                rules )
            nil ) ) ) )
```

Exercice 5.17 (page 59) :

```
(defmacro match-lambda (filtre . faire)
  (let ((label (gensym))
        (variable (gensym)) )
    `(lambda (,variable)
        (block ,label
          (funcall (match-all-lambda ,filtre
                      (return-from ,label (progn . ,faire)) )
                   ,variable ) ) ) ) )
```

Exercice 5.18 (page 88) :

****labelled-times**

```
(de-filtre6 (**labelled-times nom filtre1 filtre2)
 rudimentaire6 :compilation
    (if (extraire6 filtre1)
      (filtre6-erreur "MATCH : Affectations dans ~S."
                      (extraire6 filtre1) )
      (let ((try (gensym))
            (label (gensym))
            (success (gensym))
            (ge (gensym))
            (gr (gensym)) )
        `(block ,label
           (flet ((,success (,ge)
                    ,(compiler6 ge filtre2 env succ
                      ou-sinon6nil segvarenv timesvarenv ) ))
             (prog ((,gr ,e))
               (,success ,gr)
               ,try
               ,(compiler6 gr filtre1 env
                 (et-puis6 (e1 env1 alt1)
                  (filtre6-erreur
```

```
                  "MATCH : Repetition en cours !" ) )
               (ou-sinon6 (sechapper label
                                     (reprendre6 alt) ))
               (initial6env)
               (ajouter6boucle (initial6env) nom
                 (et-puis6 (e1 env1 alt1)
                    (faire `(,success ,e1)
                           (affecter gr e1) ) ) ) )
             (go ,try) ) ) ) ) ) )
```

Exercice 5.19 (page 91) :
Laissé à votre sagacité naturelle.

Exercice 5.20 (page 101) :
En voici un qui n'est certainement pas celui auquel vous avez pensé (et pour lequel je ne donnerai d'ailleurs pas de solution) :

```
(de-filtre6 (*trace . filtres) macro-element6                                    *trace
  :expansion
    `(*and (*success
             (progn (setf *trace-compteur* 0)
                    (format t "~& Je débute la trace ...")
                    t ) )
           (*list ,@(mapcar #'(lambda (f)
                    `(*and ,f
                           (*success
                             (progn
                              (format t " ~D"
                               (incf *trace-compteur*) )
                             t ) )) )
                          filtres ) )
           (*success (progn (format t ".")
                          t )) ) )
```

Ce filtre imprime des chiffres lorsque ses différents termes sont filtrés avec succès.

Exercice 5.21 (page 120) :
Cet exercice est un véritable projet. Lisez d'abord [Queinnec 90] afin de partir sur des bases plus simples que celles ici exposées.

Index des fonctions

J'écris. J'écris que j'écris. Mentalement je me vois écrire que j'écris et je peux aussi me voir voir qui écris. Je me rappelle écrivant déjà et aussi me voyant qui écrivais. Et je me vois me rappelant que je me vois écrire et je me rappelle me voyant me rappeler que j'écrivais et j'écris en me voyant écrire que je me rappelle m'être vu écrire que je me voyais écrire que je me rappelais m'être vu écrire que j'écrivais et que j'écrivais que j'écris que j'écrivais. Je peux aussi m'imaginer écrivant que j'avais déjà écrit que je m'imaginais écrivant que je me vois écrire que j'écris.

Salvador Elizondo, *Le Graphographe*

Index