Master Mention Informatique
Parcours AIGLE

HMIN 104 Compilation et Interprétation

Petit imprécis de LISP
Version COMMON LISP

Roland Ducournau
ducournau@lirmm.fr
http://www.lirmm.fr/~ducour
—
Département d'Informatique
—
Faculté des Sciences
Université de Montpellier

7 octobre 2015

1

Le langage Lisp, dialecte COMMON LISP

Un polycopié lu vaut mieux qu'un étudiant lambda collé.

1.1 Introduction

Ce polycopié est un cours LISP en chantier (déjà bien avancé, mais éternel). Après cette brève introduction, il est constitué :

- d'un bref manuel du langage et du système LISP (chapitre 2) ;
- d'un cours de programmation en LISP (chapitre 3) ;
- d'un récapitulatif des fonctions principales qu'un programmeur LISP doit connaître, et qu'un examen peut présumer connues (chapitre 4).
- d'une description de l'environnement d'utilisation de COMMON LISP qui est utilisé dans les TP : cette partie est à lire impérativement (chapitre 5).
- d'un ensemble d'exercices de niveau débutant, c'est-à-dire pouvant constituer un examen après un cours de 25 à 30 heures (chapitre 6) ;
- d'un glossaire des termes usuels de LISP et de la programmation fonctionnelle (chapitre 7) ;
- il est enfin accompagné d'un second polycopié dédié à la compilation et interprétation des langages [DL13].

Ce polycopié s'adresse particulièrement au dialecte COMMON LISP [Ste90] mais il provient d'une révision à partir d'un original (Chapitres 4 et 6) destiné à LELISP [Cha93], lui-même révisé pour ILOGTALK [ILO95] [1].

COMMON LISP fait partie des distributions Linux usuelles sous le nom de CLISP ou est téléchargeable sur le site Web `http://clisp.cons.org/`. C'est l'implémentation GNU de COMMON LISP. Il est aussi porté sur MAC et sur CYGWIN, mais pas directement sur Windows. Pour Windows, il doit exister des implémentations alternatives, qui ne sont pas forcément gratuites...

Il ya des alternatives à CLISP :

- SBCL (`http://www.sbcl.org/`), dérivé de CMU CL (`http://www.cons.org/cmucl/`) ;
- Clozure CL `http://trac.clozure.com/ccl` muni de l'IDE `http://common-lisp.net/project/lispbox/` ;
- pour un survol récent : `http://common-lisp.net/~dlw/LispSurvey.html`.

COMMON LISP possède à la fois un noyau objet très primitif (macro `defstruct`) et une sur-couche objet très sophistiquée, appelée CLOS ([Ste90]), qui ne sont pas décrits ici mais présenté dans un polycopié des modules de Master sur la programmation objets [Duc13].

1.2 Objectif

Ce polycopié a un triple objectif :

1. Des incohérences ou erreurs sont donc possibles : merci de les signaler à `ducournau@lirmm.fr`.

1. c'est un manuel du langage et un cours de programmation en LISP : le lecteur est censé maîtriser le langage à l'issue du cours ;
2. ce langage est utilisé dans le cours de compilation comme outil, pour programmer des mécanismes de compilation et d'interprétation ; il est aussi utilisé comme langage source des mécanismes de compilation et d'interprétation : l'objectif du cours est d'écrire en LISP un compilateur et un interprète du langage LISP ; la problématique de la compilation et de l'interprétation des langages est abordée dans un autre polycopié, mais les éléments du langage qui sont directement en rapport avec cet objectif seront soulignés au fur et à mesure ;
3. enfin, ce manuel est aussi un cours de programmation plus abstrait : la plupart des concepts, remarques et conseils sur la programmation peuvent s'abstraire du cas particulier de LISP pour s'appliquer à n'importe quel langage.

1.3 Bibliographie

1.3.1 Ouvrages de référence

[ASS89] est une bonne référence pédagogique, bien que basée sur encore un autre dialecte, SCHEME : la bibliothèque universitaire en possède de nombreux exemplaires.

[WH84] est un livre classique, très pédagogique, que je conseille.

[Que94] et [SJ93] sont des livres de plus haut niveau, ne pouvant intéresser que les étudiants qui maîtrisent déjà LISP et qui auraient apprécié ce cours.

[Nor92] donne de nombreux exemples d'utilisation de LISP en Intelligence Artificielle.

De très nombreux autres ouvrages sur LISP existent : il n'y a pas de raison qu'ils ne soient pas bons !

1.3.2 Manuels en ligne

[Ste90] est le manuel de référence complet et constitue la bible de COMMON LISP : il est indispensable pour qui veut programmer sérieusement en COMMON LISP mais présente un immense intérêt pour tous les programmeurs LISP pour le recul avec lequel il présente les spécifications du langage. Il est disponible sur les machines du département à l'URL `file:/usr/doc/cltl/clm/clm.html` et sera appelé dans la suite "**le manuel**". Le chapitre 4 du présent document constitue un pense-bête qui ne remplace pas la consultation de la description de ces fonctions dans le manuel en ligne.

Une documentation sur CLISP, qui regroupe [CLI97a, CLI97b] est disponible sur le site Web `http://clisp.cons.org/`.

La page du module (ou des anciens modules de compilation du Master), accessible à partir de ma page web — `http://www.lirmm.fr/~ducour/`, puis suivre *Enseignement* — pointe sur une page sur LISP qui regroupe un grand nombre de ressources en ligne.

Les moteurs de recherche du commerce (pas de nom !) sont enfin une dernière source.

2

Langage et système LISP

Un copier-coller est une λ-abstraction qui s'ignore.

Ce survol rapide des points essentiels n'a pas l'intention d'être complet. Le lecteur qui ne connaît pas LISP ni SCHEME, peut compléter avec l'un des ouvrages de la bibliographie, en particulier le manuel en ligne de COMMON LISP [Ste90].

Ce chapitre présente le langage et le système LISP, alors que le chapitre suivant s'intéresse à la façon de s'en servir.

Notations et méta-notations Dans ce qui suit, les guillemets français (« ») sont utilisés dans le méta-langage pour désigner des éléments du langage formel LISP. Les simples ou doubles *quotes* à l'anglaise (‘ ’ ou “ ”) sont utilisés dans le méta-méta-langage pour désigner les termes du méta-langage.

Remarque 2.1. Sur le sens du mot « méta » et de ses différentes déclinaisons, voir le polycopié de Compilation [DL13], Section 4.1. Le polycopié de programmation par objets [Duc13] donnera encore un autre éclairage sur ce mot.

2.1 Syntaxe

La syntaxe du langage LISP est la plus simple qui soit. Elle repose sur une notation dite *polonaise préfixée complètement parenthésée*.

Au lieu d'être *infixe*, comme « `a+b` » la notation est *préfixée*, en plaçant l'opérateur en premier, ce qui donne « `+ a b` ». Elle est enfin *complètement parenthésée* « `(+ a b)` » et les espaces entre les éléments de l'expression sont obligatoires, « `(+ab)` » ayant un tout autre sens. Pour les appels de fonctions ou procédures, la parenthèse ouvrante est placée avant la fonction, comme dans « `(foo x)` », et non pas après comme dans « `foo(x)` ».

Cette syntaxe fait que la structure des programmes consiste en une imbrication d'expressions parenthésées qui font traduire « LISP » par *Lot of Idiotic* (ou *Insane*) *Stupid Parentheses*. Il est donc essentiel d'utiliser un éditeur de texte qui sache gérer proprement ces parenthèses, donc avec un mode LISP : peu de solutions, `emacs` ou `vim` (ou `gvim`), voire ECLIPSE, dans chaque cas avec le mode LISP spécialisé ! Éviter comme la peste les éditeurs à la Windows, de Windows, KDE ou GNOME !

2.1.1 Expressions LISP

Contrairement à de nombreux langages, LISP ne fait pas la différence entre *expression* et *instruction* : la syntaxe de LISP est uniformément basée sur la notion d'*expression* [1].

En première approximation, la syntaxe se définit formellement comme suit :

1. On trouve dans la littérature 2 termes synonymes pour ‘expression’ : *S-expression* et *form* (ou *forme* en français).

<expr>	:=	<atome> | <cons>
<atome>	:=	<symbol> | <constante>
<constante>	:=	<number> | <string> | « ( ) » | etc.
<liste>	:=	« ( ) » | <cons>
<cons>	:=	« (» <expr>+ «) » | « (» <expr>+ « . » <expr> «) »

Cette syntaxe est *abstraite* : elle ne tient pas compte de la syntaxe *concrète* des diverses constructions du langage, qui sera introduite au fur et à mesure.

Une expression est soit un *atome*, soit une *liste non vide*. Un *atome* est lui-même un *symbole* ou un *littéral* (aussi appelé *constante*.

Remarque 2.2. La syntaxe qui précède, ainsi que toutes celles qui suivront, suit une autre syntaxe [2], dite *Backus-Naur Form* (BNF), dont les conventions sont les suivantes. Une syntaxe s'exprime par un ensemble de règles. Chaque règle est de la forme `<gauche> := droite`, où `<gauche>` est un syntagme non terminal dont *droite* décrit la décomposition. Chaque syntagme non terminal est représenté par un symbole entre « < > ». Dans *droite*, le symbole « | » décrit des alternatives, « + » représente la répétition d'au moins une occurrence du syntagme qui précède et « * » représente une répétition quelconque. Enfin les '« »' insèrent des littéraux dans la syntaxe.

2.1.2 Commentaires

La seule alternative aux expressions est constituée par les commentaires, qui sont de syntaxe libre à part leurs délimiteurs.

Les commentaires mono-lignes sont introduits par un « ; », jusqu'à la fin de la ligne. Les commentaires multi-lignes sont délimités par « #| » et « |# ».

2.1.3 Atomes

Un atome peut être un *symbole* (`symbol`) ou n'importe quelle *constante* (ou *littéral*) du langage : nombre (`integer`, `float`, ...), chaînes de caractères (`string`), etc. La syntaxe de ces constantes dans les usages usuels est en général la syntaxe usuelle. Pour les usages plus sophistiqués (diverses représentations des nombres, caractères d'échappement, etc.) voir le manuel.

Séparateurs

Les seuls séparateurs du langage sont : les parenthèses, l'espace et le passage à la ligne.

NB Les caractères utilisés dans la plupart des langages comme opérateurs arithmétiques (+, –, *, /, etc.) ne sont pas des séparateurs, pas plus que le « . » et la plupart des ponctuations.

Caractères réservés

Quelques caractères sont réservés à un usage particulier : « # », « , », « ' » et « ` », « " », « : », etc. : leur usage inconsidéré provoque une erreur. Ils seront décrits au fur et à mesure du besoin.

Symboles

Les *symboles* jouent le rôle des *identificateurs* habituels dans les langages de programmation, mais leur syntaxe est en LISP beaucoup plus générale (seuls sont exclus quelques caractères spéciaux). De plus il s'agit de valeurs du langage, de type `symbol` (voir la fonction `get` et la notion de propriété des symboles).

Tous les caractères autres que ceux cités avec un rôle particulier peuvent apparaître dans un symbole et tout ce qui n'est pas un nombre est alors un symbole. Pour les caractères alphabétiques, la *casse* (*minuscule* ou *majuscule*) est sans effet : le langage traduit tout en majuscules et il est d'usage de programmer en minuscules.

2. Une méta-syntaxe donc...

Exemples d'atomes

```
( )        ; liste vide
3.14151    ; flottant
3          ; entier
3/4        ; rationnel
"3"        ; chaîne
azerty     ; symbole AZERTY
aZeRtY     ; symbole AZERTY
AZERTY     ; symbole AZERTY
"aZeRtY"   ; chaîne
3.14AZER   ; symbole
A.B        ; symbole
A+B        ; symbole
A,B        ; erreur
"A,B"      ; chaîne
A!         ; symbole
```

2.1.4 Listes

Une *liste* est une séquence d'expressions, séparées par des espaces ou passages à la ligne, et délimités par des parenthèses. La liste vide est un aussi un atome, noté `( )` ou `nil`.

Exemples de listes

```
( )                    ; liste vide
(A B)                  ; liste à 2 éléments
(+ 3 4)                ;
(foo x y)              ;
(A B "3.14" C)         ; liste à 4 éléments
(A B "3.14 C)"         ; erreur (mauvais imbrication)
(A (B (C D) E) F)      ; liste imbriquée
(A (B                  ;
     (C D) E)          ;
         F)            ; liste sur plusieurs lignes
"(A (B (C D) E) F)"    ; chaîne de caractères !
("A (B (C D) E) F")    ; liste d'un élément !
(A.B)                  ; liste à 1 éléments
(A . B)                ; paire pointée (voir plus loin)
(A,B)                  ; erreur
```

Éléments de style

Comme dans tout langage, naturel ou artificiel, la syntaxe peut être correcte mais le texte illisible par un humain car les conventions habituelles ne sont pas respectées.

Les conventions sur les listes et la place des parenthèses sont les suivantes :

```
(A b C)      ; OUI   pas d'espace à l'intérieur des parenthèses
( A b C )    ; NON
(A (b C) D)  ; OUI   des espaces à l'extérieur des parenthèses
(A(b C)D)    ; NON
(            ; NON   pas de passage à la ligne après une ouvrante
     B C)    ;
(B C         ; NON   pas de passage à la ligne avant une fermante
     )       ;       surtout la première
```

Les consignes sur l'*indentation* seront examinées plus loin. Dans tous les cas, oublier le style des accolades (« { » et « } ») de C++ et JAVA !

Les conventions sur les identificateurs (symboles) sont les suivantes :

— pas de distinction majuscule/minuscule ;
— la plupart des caractères spéciaux (-,+,*, !, ?,~ etc.) ne le sont pas en LISP, et peuvent donc être utilisés dans les identificateurs ;
— en particulier, les mots composés sont séparés par des « – » (tirets) et non par des « _ » (soulignés). Cette convention est très importante car COMMON LISP est une bibliothèque de plusieurs centaines ou milliers de fonctions dont les noms sont souvent composés.

2.2 Sémantique : évaluation

La *sémantique* d'un langage spécifie le fonctionnement des programmes.

2.2.1 Expressions et valeurs

En LISP, elle est basée sur l'*évaluation* des expressions, qui doit toujours *retourner* une *valeur*, sauf terminaison exceptionnelle. Mais toutes les expressions ne sont pas évaluables et la sémantique du langage est déterminée par l'évaluation des *expressions évaluables*.

En revanche, toute expression LISP, y compris les expressions évaluables, est une donnée correcte pour un programme LISP : toute expression est une valeur. Cependant, les termes 'valeur' et 'expression' ne sont pas exactement synonymes : il existe des valeurs qui ne sont pas imprimables et relisibles. Donc, une expression est une valeur restituable par `print` et `read`.

La syntaxe d'une expression évaluable est la suivante :

<expr-eval>	:=	<atome> | <cons-eval> | <backquote>
<atome>	:=	<symbol> | <constante>
<constante>	:=	<number> | <string> | « ( ) » | etc.
<cons-eval>	:=	« (» <fonction> <expr-eval>* «) » | <forme-speciale> | <forme-macro>
<fonction>	:=	<symbol> | « **(lambda** » <liste-param> <expr-eval>+ « **)** »
<liste-param>	:=	« (» <symbol>* «) »

La syntaxe des `<forme-speciale>`, `<forme-macro>` et `<backquote>` sera examinée plus loin.

2.2.2 Évaluation des atomes

Constantes L'évaluation d'une *constante* (aussi appelée *littéral*) du langage retourne la constante.

Symboles L'évaluation d'un *symbole* considère le symbole comme une *variable* et retourne la valeur liée à ce symbole — on parle de *liaison* — dans l'*environnement* courant. Une *exception* est levée lorsque le symbole n'est pas lié dans l'environnement courant.

Il est très important de noter qu'un symbole **n'est pas** une variable mais **joue le rôle** d'une variable.

2.2.3 Évaluation des listes non vides : les fonctions

L'évaluation d'une liste non vide interprète cette liste comme une expression en *notation (polonaise) préfixée, totalement parenthésée* et le premier élément de la liste est considéré comme une *fonction*. Une exception est levée dans le cas contraire.

Contrairement aux langages de programmation "habituels", LISP ne fait pas de différence entre opérateurs, mots-clés, procédures et fonctions : **tout est fonction**. En revanche, il y a plusieurs catégories de fonctions.

Les "vraies" fonctions

C'est l'essentiel des fonctions, celles qui sont prédéfinies pour la manipulation des différents types de données et celles que le programmeur définit. Le principe de l'évaluation de ces expressions consiste à *appliquer* la fonction aux résultats de l'évaluation récursive de ses arguments, qui sont les autres éléments de la liste. Les vraies fonctions peuvent être globales (introduites par `defun`) ou locales (introduites par `labels`), ou enfin anonymes (λ-fonctions).

Fonctions globales Les fonctions globales sont connues dans la globalité de l'environnement d'évaluation. Elles sont *prédéfinies* ou bien introduites par `defun`.

Les λ-fonctions (lambda-fonctions) Ce sont de vraies fonctions, mais anonymes. Elles servent 1) à introduire des *variables locales* — mais on préférera la forme syntaxique `let` — ou 2) à faire des valeurs fonctionnelles (*fermetures*).

Fonctions locales Ce sont des fonctions qui ne sont liées à leur identificateur que dans la *portée lexicale* de la construction qui les introduit, de la même manière que les variables mais dans un espace de nom différent. Voir `labels` et `flet` (Section 3.2.1).

Les formes syntaxiques

Il est impossible de ne programmer qu'avec des vraies fonctions : une construction aussi fondamentale que la conditionnelle serait impossible. Le langage LISP dispose donc de constructions syntaxiques, dites *formes syntaxiques* ou *formes spéciales*, qui sont destinées à réaliser ce qui ne peut pas être fait au moyen de vraies fonctions.

Ces formes syntaxiques sont toutes *prédéfinies* et leur principe d'évaluation est propre à chacune. Elles n'évaluent pas toujours tous leurs arguments. Les principales sont `if`, `quote`, `defun`, `let`, `cond`, `labels` et `flet`, etc. Voir leurs définitions dans la suite de ce chapitre et en Section 4.1.1.

Les macros

Les macros sont des fonctions qui assurent des *transformations source à source*. Ce sont des fonctions à effet syntaxique, qui n'évaluent pas leurs arguments et retournent une nouvelle expression évaluable qui remplace l'expression d'origine avant d'être à son tour évaluée. Voir Section 3.5.

2.3 Application d'une vraie fonction

2.3.1 Définition d'une fonction

On définit une fonction globale par la forme syntaxique `defun` qui n'évalue aucun de ses arguments. La syntaxe de `defun` est la suivante :

```
(defun <nom> <liste-param> <expr-eval>)
```

L'expression `<expr-eval>` constitue le *corps* de la fonction. La liste de paramètres d'une fonction est constituée, dans l'ordre,

1. d'un certain nombre de paramètres *obligatoires*,
2. suivis éventuellement de paramètres *optionnels*, préfixés par `&optional`,
3. ou bien de paramètres passé par *mots-clés*, préfixés par `&key`
4. et finalement d'un paramètre lié à la liste des arguments restants, préfixé par `&rest`.

La plupart des fonctions se contentent du cas (1).

λ-fonction Rappelons que λ (prononcer et écrire en LISP `lambda`) est l'équivalent du 'L' dans l'alphabet grec [3].

La syntaxe d'une λ-fonction est la suivante :

```
(lambda <liste-param> <expr-eval>)
```

Exemples On définira par exemple la fonction `square` comme suit :

```
(defun square (x) (* x x))
```

et on s'en servira ainsi :

```
(square 9) ; -> 81
```

Enfin on fera la même chose avec une *λ-fonction*, ce qui donne une *λ-expression*.

```
((lambda (x) (* x x)) (+ 4 5)) ; -> 81
```

Noter les 2 « (» (ouvrantes) qui débutent l'expression : la première ouvre la *λ-expression* et la seconde la *λ-fonction* [4].

2.3.2 Application et environnement

L'application d'une vraie fonction consiste à évaluer le *corps* de la fonction dans l'*environnement* résultant de l'appariement des *paramètres* de la fonction avec les *valeurs* résultant de l'*évaluation* des *arguments* correspondants, et à *retourner* la *valeur* obtenue. Chaque paire paramètre-valeur est appelée une *liaison*.

Si le nombre d'arguments fournis n'est pas compatible avec la liste de paramètres, une exception est levée.

Exemple Soit la fonction

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

et l'expression à évaluer

```
(foo 3 4 (+ 5 6))
```

De façon plus procédurale cela donne la séquence suivante :

1. on évalue récursivement les arguments de l'appel, ce qui donne successivement les valeurs 3, 4 et 11 ;
2. on apparie les paramètres avec les valeurs, ce qui donne un "environnement" c'est-à-dire une structure invisible où chaque symbole est associé à sa valeur : $\{A \to 3, B \to 4, C \to 11\}$;
3. on évalue le corps de `foo`, c'est-à-dire `(- (* b b) (* 4 a c))` dans cet environnement ;
4. on évalue donc récursivement les arguments de $-$, dont les valeurs sont 16 et 132 ;
5. et on retourne -116.

Remarque 2.3. On essaie d'utiliser ici les termes de 'paramètre' et 'd'argument' avec le plus de rigueur possible. Un *paramètre* est un paramètre *formel*, c'est-à-dire le nom utilisé à l'intérieur de la fonction : c'est le point de vue de l'*appelée*. Un *argument* représente l'expression (et sa valeur) qui est passée à la fonction pour un appel particulier : c'est le point de vue de l'*appelant*.

3. En français dérivé d'un argot scientifique, 'lambda' signifie 'quelconque'. Comble de l'argot : un pékin lambda, c'est un quidam.

4. Faites ce que je dis, pas ce que je fais (ou plutôt ce que j'écris, pas ce que je dis) : à l'oral, par suite d'une vieille habitude, je dirai souvent l'un pour l'autre.

Environnement d'appel et de définition, environnements imbriqués

L'environnement d'application d'une fonction — celui qui résulte de l'appariement paramètres-valeurs, cf. ci-dessus — contient toujours l'environnement dans lequel la fonction a été définie et non pas l'environnement d'appel de la fonction. En général, pour les fonctions globales, cet environnement de définition est vide : si ce n'est pas le cas, on parlera de *fermeture* (voir ci-dessous). Dans le seul cas des λ-expressions qui sont utilisées pour introduire des variables locales (comme dans l'exemple ci-dessus), les deux environnements sont confondus.

Soit les deux fonctions `f` et `g` :

```
(defun g (v) (* 5 (f (+ v 2))))
(defun f (x) (+ v x))
```

Si ces deux définitions sont faites au *toplevel* — c'est-à-dire ne sont pas incluses dans une autre expression LISP—, leur *environnement de définition* est vide et la définition de `f` est erronée car tout appel de `f` produira une erreur, même lorsque `f` est appelée par `g`. Soit l'appel `(g 8)` : le corps de `g` est évalué dans l'environnement `{V` $\rightarrow$ `8}` : c'est l'*environnement d'appel* de `f` qui va être appelé avec un argument qui vaut 10. Mais le corps de `f` va être évalué dans l'environnement `{X` $\rightarrow$ `10}` qui n'inclut pas l'environnement d'appel, donc `v` est un symbole qui ne joue aucun rôle de variable dans le corps de `f`.

En revanche, si l'on remplace l'appel de `f` par la λ-fonction correspondante, on obtient :

```
(defun g (v) (* 5 ((lambda (x) (+ v x)) (+ v 2))))
```

L'environnement d'appel de la λ-fonction est toujours `{V` $\rightarrow$ `8}`, mais c'est aussi son environnement de définition et le corps de la λ-fonction sera évalué dans l'environnement `{V` $\rightarrow$ `8,X` $\rightarrow$ `10}`. C'est le principe habituel de *portée* des identificateurs dans des blocs imbriqués et, si le même symbole est *introduit* par plus d'une λ-fonction, chaque occurrence désigne le symbole introduit par la λ-fonction englobante la plus proche.

On désigne habituellement ce principe par le terme 'lexical' : portée, liaison et environnement sont ici *lexicaux*. Le principe de la *lexicalité* fait que les noms sont sans importance : étant donné une λ-fonction (ou toute autre fonction) `(lambda (x) ...)` qui introduit le paramètre `x`, on peut substituer à `x` n'importe quel autre symbole non constant, pourvu qu'il soit absent du corps de la λ-fonction.

'Lexical' s'oppose à 'dynamique' qui sera examiné ailleurs.

Passage de paramètres

La syntaxe complète de la liste de paramètres est la suivante :

<liste-param>	:=	« **(** » <param-oblig>* [« **&optional** » <param-opt>+][« **&rest** » <param-oblig>] « **)** » | « **(** » <param-oblig>* [« **&key** » <param-opt>+] « **)** »
<param-opt>	:=	<param-oblig> | « (» <param-oblig> <expr-eval> [<param-oblig>] «) »
<param-oblig>	:=	<symbol-var>
<symbol>	:=	<symbol-var> | <symbol-const>
<symbol-const>	:=	« **NIL** » | « **T** » | <keyword>
<keyword>	:=	« **:** »<symbol-var>

Dans les paramètres optionnels, l'expression `<expr-eval>` représente la valeur par défaut associée au paramètre si aucun argument n'est fourni (par défaut, cette valeur est `nil`) et le deuxième `<param-oblig>` est un paramètre facultatif qui est lié à `t` ou `nil` suivant que l'argument a été explicitement passé ou pas : il est ainsi possible, dans la fonction de distinguer le cas où la valeur par défaut a été passée comme un argument explicite. Tous les `<param-oblig>` de chaque `<liste-param>` doivent être 2 à 2 distincts.

La correspondance entre les paramètres et les valeurs des arguments se fait comme suit :

— les paramètres obligatoires et optionnels sont *positionnels* : c'est leur position qui détermine la correspondance et la seule différence entre les 2 est que le paramètre optionnel peut suppléer à un argument manquant par une valeur par défaut ;

— le passage des paramètres par *mot-clé* remplace le critère de la position par le nom du paramètre, sous la forme d'un '*keyword*', c'est-à-dire d'un symbole (le nom du paramètre) préfixé par « : ».

Exemples

`Paramètres`	`Arguments`	Environnement	
`(x y z)`	`(1 2 3)`	$\{x \to 1, y \to 2, x \to 3\}$	*param. obligatoires*
`(x y z)`	`(1 2)`		*pas assez d'arguments*
`(x y z)`	`(1 2 3 4)`		*trop d'arguments*
`(x y &optional z)`	`(1 2 3)`	$\{x \to 1, y \to 2, z \to 3\}$	*param. optionnels*
`(x y &optional z)`	`(1 2)`	$\{x \to 1, y \to 2, z \to ()\}$	
`(x y &optional (z 4))`	`(1 2)`	$\{x \to 1, y \to 2, z \to 4\}$	
`(x y &optional (z 4 zp))`	`(1 2)`	$\{x \to 1, y \to 2, z \to 4, zp \to ()\}$	
`(x y &optional (z 4 zp))`	`(1 2 4)`	$\{x \to 1, y \to 2, z \to 4, zp \to t\}$	
`(x y &rest z)`	`(1 2)`	$\{x \to 1, y \to 2, z \to ()\}$	*param.* `&rest`
`(x y &rest z)`	`(1 2 3)`	$\{x \to 1, y \to 2, z \to (3)\}$	
`(x y &rest z)`	`(1 2 3 4)`	$\{x \to 1, y \to 2, z \to (34)\}$	
`(x &key y z)`	`(1 :z 2 :y 3)`	$\{x \to 1, y \to 3, z \to 2)\}$	*mots-clés*
`(x &key y z)`	`(1 :z 2)`	$\{x \to 1, y \to (), z \to 2)\}$	
`(x &key y z)`	`(1)`	$\{x \to 1, y \to (), z \to ())\}$	
`(x &key y z)`	`(1 :w 4)`		*mot-clé illicite*

On peut faire la combinatoire des déclarations de valeurs par défaut de paramètres `&key` comme pour `&optional`. On peut aussi utiliser ensemble `&optional` puis `&rest`. Il est en revanche dangereux de combiner `&key` avec les deux autres.

2.3.3 Fermetures

C'est une notion plus avancée qui sera traitée Section 3.4.3.

2.4 Structures de contrôle et formes syntaxiques

2.4.1 Formes syntaxiques usuelles

La programmation LISP usuelle utilise un ensemble limité de formes syntaxiques dont la syntaxe est spécifique à chacune.

<forme-speciale>	:=	<quote> | <if> | <cond> | <case> | <labels> | <let> | <defun> | ...

— `quote` permet d'éviter d'évaluer une expression, donc la considère comme une donnée et non pas un programme ; c'est sans effet sur les constantes mais nécessaire lorsque l'on souhaite traiter un symbole ou une liste comme une donnée ; `quote` s'abrévie en « `'` » (voir Section 2.8.1) ;

<quote>	:=	« (**quote** » <expr> «) » | « ' » <expr>

— `if` est la forme conditionnelle de base ; la syntaxe est la suivante :

<if>	:=	« (**if** » <condition> <alors> <sinon> «) »
<condition>	:=	<expr-eval>
<sinon>	:=	<expr-eval>
<alors>	:=	<expr-eval>

La `<condition>` est d'abord évaluée : si sa valeur est *vraie*, c'est-à-dire différente de `nil`, le deuxième argument `<alors>` est ensuite évalué, sinon c'est le troisième, `<sinon>`, qui est évalué ; `if` retourne la valeur retournée par l'évaluation de `<alors>` ou `<sinon>` ;

— `cond` est une alternative à `if`, qui permet d'enchaîner une séquence de tests ; la syntaxe est la suivante :

<cond>	:=	« **(cond** » <clause-cond>+ «) »
<clause-cond>	:=	« (» <condition> <expr-eval>+ «) » | « (» <condition> «) »

Une expression `cond` est une liste non vide de *clauses*. Chaque clause est elle-même formée d'une condition seule ou d'une condition suivie par une liste d'expressions. Les conditions sont évaluées successivement jusqu'à la première qui retourne non `nil` : si la clause se réduit à la condition (cas 1), la valeur de la condition est retournée, sinon les expressions sont évaluées en séquence (`progn` implicite) et la valeur de la dernière est retournée. Si aucune condition n'est satisfaite, l'expression `cond` retourne `nil`.

— `case` permet de comparer une valeur à des constantes successives :

<case>	:=	« **(case** » <expr-eval> <clause-case>+ «) »
<clause-case>	:=	« (» <constante> <expr-eval>* «) » |
		« ((» <constante>+ «) » <expr-eval>* «) »

Dans chaque clause, la condition est ici remplacée par une constante ou une liste de constante : `<expr-eval>` est d'abord évalué puis sa valeur est confrontée aux constantes de chaque clause, jusqu'à ce que l'une d'elles soit égale (`eql`) à la valeur.

— `defun` a été décrit ci-dessus,

— les autres formes syntaxiques seront décrites au fur et à mesure.

Chacune de ces fonctions est décrite plus précisément dans le chapitre 4.

Remarque 2.4. Comme dans beaucoup de langages, il n'y a pas de type booléen en LISP et la convention est que `nil` joue le rôle de 'faux' et toute autre valeur le rôle de 'vrai'. Le symbole constant `t` tient souvent le rôle de 'vrai' (`t` pour *true*).

Exemples

```
(+ 2 3)            → 5
(quote (+ 2 3))    → (+ 2 3)
'(+ 2 3)           → (+ 2 3)
(if (< 3 4) 5 6)   → 5
```

2.4.2 Récursion

La structure de contrôle la plus communément utilisée en LISP est la récursion. On définira ainsi la fonction factorielle :

```
(defun fact (n)
    (if (= n 0)
        1
      (* n (fact (- n 1)))))
```

2.4.3 Itération

Une structure alternative, moins familière aux lispiens est l'itération. On imprimera tous les éléments d'une liste comme suit :

```
(loop x in '(1 2 3 4 5)
      do
      (print x))
```

Voir le chapitre sur la macro `loop` dans le manuel COMMON LISP.

2.4.4 Variables locales

L'une des morales à tirer du langage LISP est qu'il n'y a pas de *variable* sans *fonction*. Toute variable est *introduite* par la définition d'une fonction dont elle est un paramètre.

La façon primitive d'introduire une variable locale est donc la λ-expression (ou `lambda`). La syntaxe n'est pourtant pas très agréable.

```
(defun foo (x y z)
     ((lambda (a b c d)
              <corps de lambda>)
      <expr-a>
      <expr-b>
      <expr-c>
      <expr-d>))
```

Dans cet exemple, la fonction `foo` a besoin de 4 variables locales initialisées par les valeurs des expressions `<expr-a>` à `<expr-d>`. S'il y a beaucoup de variables ou si `<corps de lambda>` fait plusieurs lignes, voire une page, la correspondance entre les variables et les valeurs d'initialisation sera difficile à établir pour le programmeur (humain).

Aussi une forme syntaxique plus appropriée consiste à regrouper par paires variables et expressions d'initialisation :

```
(defun foo (x y z)
     (let ((a <expr-a>)
           (b <expr-b>)
           (c <expr-c>)
           (d <expr-d>))
        <corps de lambda>))
```

Les deux formes sont strictement équivalentes. La syntaxe du `let` est la suivante :

<let>	:=	« **(let (** » <var-let>+ «) » <expr-eval> «) »
<var-let>	:=	<param-oblig> | « (» <param-oblig> [<expr-eval>] «) »

Si la variable n'est pas accompagnée par une `<expr-eval>`, elle est initialisée à `nil`.

Remarque 2.5. Quand faut-il introduire une variable locale ? La variable `a` de l'exemple précédent est nécessaire si

1. elle apparaît plusieurs fois dans `<corps de lambda>` comme sous-expression évaluable ;
2. elle est initialisée par une expression `<expr-a>` non triviale : c'est-à-dire un appel de fonction qui soit n'est pas en temps constant, soit est d'un temps constant suffisamment élevé.

Si la première condition n'est pas réalisée, il suffit de remplacer `a` par `<expr-a>` dans `<corps de lambda>` pour simplifier le code sans affecter son efficacité et sa lisibilité (voir aussi Remarque 3.4). Si la première condition est réalisée, pour la deuxième condition, si `<expr-a>` est un appel récursif de `foo` la variable locale est indispensable. Si `<expr-a>` est une constante ou une expression *quotée*, la variable locale est inutile. Finalement, si `<expr-a>` est une expression comme `(car x)`, les deux sont possibles.

`Let` séquentiel et `let` parallèle Cette forme `let` est dite *parallèle* car les expressions d'initialisation sont d'abord toutes évaluées, puis ensuite 'affectées' (le terme est parfaitement impropre ce n'est pas une affectation) en bloc aux variables correspondantes. En pratique, cela signifie que `<expr-b>` ne peut pas se servir de la valeur de `a`, qui n'est pas encore connu.

Une forme alternative existe, `let*` dit '`let` *séquentiel*', de même syntaxe que `let`, qui correspond à une imbrication de λ-expressions :

```
(defun foo (x y z)
     ((lambda (a)
              ((lambda (b)
                       ((lambda (c)
                                ((lambda (d)
                                         <corps de lambda>)
                                 <expr-d>))
                        <expr-c>))
               <expr-b>))
      <expr-a>))
```

Noter que l'ordre des évaluations reste de `<expr-a>` à `<expr-d>`, mais que l'ordre d'apparition dans le texte est inverse. C'est un argument supplémentaire en faveur de la syntaxe `let` / `let*`.

2.5 Structures de données et bibliothèque de fonctions

Le système LISP constitue aussi une bibliothèque de fonctions permettant de manipuler de nombreuses structures de données.

2.5.1 Typage dynamique et prédicats

LISP est typé *dynamiquement*, c'est-à-dire que les valeurs sont partitionnées en types mais le code n'est pas annoté pour indiquer le type des expressions (variables, fonctions, etc.). Aussi la vérification de type ne peut pas être effectuée simplement par le compilateur et le programmeur a la charge de faire des vérifications explicites là où il pense que c'est utile. De son côté, le langage fournit pour chaque type un *prédicat*, c'est-à-dire une fonction à un paramètre à valeur "booléenne", qui vérifie que la valeur passée en paramètre est bien du bon type.

A part quelques rares contre-exemples (`atom`, `null`), le nom des prédicats de type est formé par le nom du type suffixé par « `p` »[5] : `integer` donne `integerp`, `string` donne `stringp`, etc. Si le nom de type contient déjà un « – », le suffixe est « `-p` ».

Le principe de la vérification de type est le suivant. A l'endroit voulu, par exemple à l'entrée d'une fonction, on insère un test explicite sur le type, et on traite les échecs de façon spéciale, par exemple en faisant appel à la fonction `error` qui lève une exception. Pour la fonction `fact`, on aura ainsi :

```
(defun fact (n)
    (if (integerp n)
        (if ... )
      (error "~s n'est pas un entier" n)))
```

Alternativement, on peut utiliser la macro `assert`, qui évite d'avoir à traiter explicitement les échecs. On peut aussi inclure d'autres conditions que le test de type, par exemple :

```
(defun fact (n)
    (assert (and (integerp n) (>= n 0)))
    (if ... ))
```

Enfin, on peut aussi utiliser les tests de type comme des conditions booléennes normales dans les structures de contrôle conditionnelles, par exemple :

```
(cond ((integerp x)
       ...)        ; cas des entiers
      ((floatp x)
       ...)        ; cas des flottants
      ...)
```

5. En SCHEME, la convention utilise le suffixe « ? ».

2.5.2 Les listes

LISP se traduit aussi par *LISt Processing* : le langage a été inventé pour manipuler des listes, essentiellement de symboles — par exemple du code LISP. C'est la grande originalité de LISP d'être un langage conçu pour manipuler ses propres programmes et c'est la raison de son utilisation quasi universelle dans les cours de compilation.

Les fonctions `eval` et `quote` sont les deux fonctions magiques qui permettent de traverser le miroir : `eval` permet de considérer une donnée comme un programme alors que `quote` permet de considérer un programme comme une donnée.

Primitives

Trois fonctions primitives — deux d'analyse et une de synthèse — sont suffisantes pour implémenter les listes.

— `car` retourne le premier élément d'une liste non vide ;
— `cdr` retourne le reste d'une liste non vide ;
— `cons` construit une liste à partir d'un élément et d'une liste.

Par abus de notation, `car` et `cdr` sont étendues à `nil` (ou `( )`) et retournent `nil`.

Remarque 2.6. Le couple `car`/`cdr` est aussi dénommé `first`/`rest` ou `head`/`tail`. Ces appellations sont plus parlantes mais `car` et `cdr` (prononcer "queudeur") sont une appellation historique et sont passés à la postérité. Par ailleurs, une famille de fonctions `c*r` est aussi définie, où `*` est une chaîne de 1 à 4 occurrences de « `a` » ou « `d` » : chaque fonction correspond à l'imbrication de `car` et de `cdr`, dans le même ordre que celui des `a` et `d`. Les fonctions `cadr`, `caddr` et `cadddr` sont ainsi respectivement équivalentes à `(car (cdr -))`, `(car (cdr (cdr -)))` et `(car (cdr (cdr (cdr -))))`. Elles sont aussi dénommées `second`, `third` et `fourth`.

Exemple

```
(car '(+ 2 3))    → +
(cdr '(+ 2 3))    → (2 3)
(cons '+ '(2 3)) → (+ 2 3)
```

Doublets

On déduit de l'interface fonctionnelle des listes le modèle des *doublets*.

— Une liste est constituée de doublets (aussi appelés *cellules* ou *cons*) qui contiennent 2 champs ;
— l'appel de la fonction `cons` crée un nouveau doublet dont les champs sont remplis par les 2 arguments ;
— la fonction `car` retourne le contenu du premier champ ;
— la fonction `cdr` retourne le contenu du second champ.

La liste `(1 2 3 4)` et la définition de factorielle se représentent donc ainsi comme dans la figure 2.1a. Par définition, tout ce qui n'est pas un doublet est un atome et réciproquement. En particulier, la liste vide n'est pas un doublet !

Prédicats

Les 3 primitives précédentes s'accompagnent de prédicats pour tester si une valeur est une liste (ou un doublet) ou pas :

— `atom` retourne *vrai* (le symbole `t`) ssi son argument est un atome ;
— `consp` retourne *vrai* ssi son argument est une liste non vide (un doublet) ;
— `listp` retourne *vrai* ssi son argument est une liste ;
— `null` retourne *vrai* ssi son argument est la liste vide ;

Ces 4 prédicats entretiennent des relations très fortes :

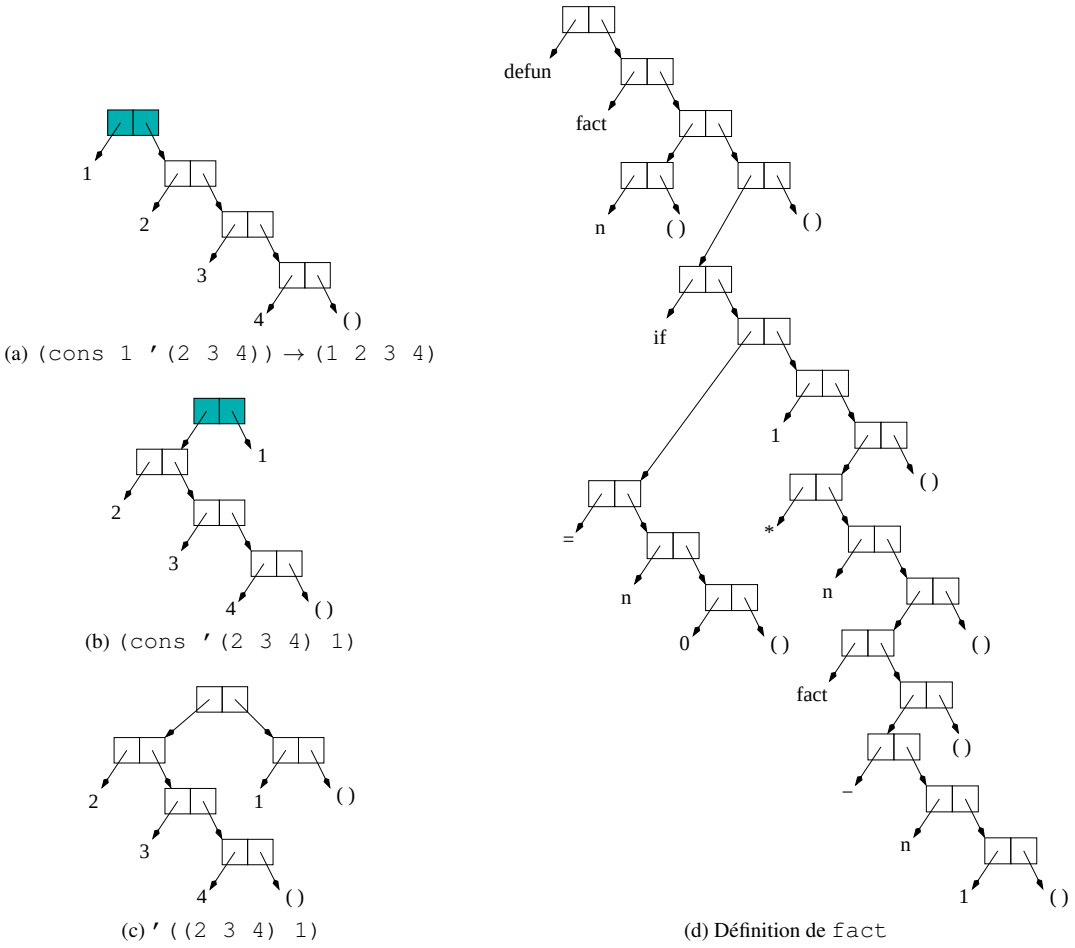


FIGURE 2.1 – Représentation par doublets : les cellules grisées sont produites par l'appel à `cons` correspondant.

— `atom` et `consp` sont des complémentaires logiques : leur disjonction est toujours vraie et leur conjonction toujours fausse ;
— `listp` et `consp` ne différent que pour `nil` et `listp` est la disjonction de `consp` et `null` ;
— la disjonction de `listp` et `atom` est toujours vraie et leur conjonction est identique à `null`.

Notation pointée

On construit une liste élément par élément en 'rajoutant' le nouvel élément en tête : `(cons 1 '(2 3 4))` retourne `(1 2 3 4)` mais le terme de 'rajout' est un abus de langage : cela retourne en réalité une nouvelle cellule (donc une nouvelle liste) qui pointe sur l'ancienne.

Que se passe-t-il si l'on inverse les 2 arguments de `cons`, par exemple en faisant `(cons '(2 3 4) 1)` ? On obtient la représentation de la figure 2.1b mais comment l'écrit-on dans la syntaxe de LISP ? La réponse n'est pas `((2 3 4) 1)` qui se représente par la figure 2.1c. Une syntaxe complémentaire est nécessaire, qui est constituée par la notation dite *pointée* : `((2 3 4) . 1)`.

Remarque 2.7. le « . » n'est pas un séparateur, il faut des espaces autour ! ! !

Il résulte de la notation pointée que la même expression LISP peut s'écrire de différentes façons, par exemple `(1 2)`, `(1 . (2))`, `(1 2 . ( ))` ou `(1 . (2 . ( )))`, de la notation pointée *minimale* à la *notation totalement pointée* en passant par tous les intermédiaires.

Le principe est le suivant : dans une liste on peut supprimer un point suivi d'une ouvrante (et l'ouvrante correspondante), ou au contraire les insérer entre deux éléments d'une liste. Avec la notation pointée, la transformation suivante est donc licite dans les deux sens :

`( <expr-lisp>+ . ( <expr-lisp>+ )) ↔ ( <expr-lisp>+ <expr-lisp>+ )`

Une notation pointée est donc minimale lorsqu'elle ne contient pas de point suivi d'une ouvrante.

La syntaxe des listes se précise donc ainsi, en distinguant les expressions *propres* (sans aucun « . »), *pointées minimales*, *totalement pointées* et *quelconques* — les 3 dernières désignent des notations différentes pour le même ensemble de valeurs, alors que les expressions propres ne sont qu'un sous-ensemble des expressions. On distingue de la même manière les listes *propres* et *quelconques* :

<atome>	:=	<symbol> | <constante>
<constante>	:=	<number> | <string> | « ( ) » | etc.
<expr-propre>	:=	<atome> | « (» <expr-propre>+ «) »
<expr-tot-pointée>	:=	<atome> | « (» <expr-tot-pointée> « . » <expr-tot-pointée> «) »
<expr-min-pointée>	:=	<atome> | « (» <expr-min-pointée>+ «) » |
		« (» <expr-min-pointée>+ « . » <atom-non-nil> «) »
<liste-qcq>	:=	« ( ) » | <cons-qcq>
<liste-propre>	:=	« (» <expr-qcq>* «) »
<expr-qcq>	:=	<atome> | <cons-qcq>
<cons-qcq>	:=	« (» <expr-qcq>+ « . » <expr-qcq> «) » | « (» <expr-qcq>+ «) »

En lecture, la fonction `read` admet toutes les variantes. En écriture, la fonction `print` utilise la notation pointée minimale.

Il ressort de cette syntaxe que certaines configurations de « (», «) » et « . » sont illégales, par exemple :

```
( <expr>+ . <expr>* . <expr>)    ; 2 « . » dans la même paire de « ( ) »
                ( <expr>+ . )    ; pas d'expression après le « . »
                ( . <expr>+ )    ; pas d'expression avant le « . »
   ( <expr>+ . <expr>+ <expr>)   ; plus d'une expression après le « . »
```

Liste plates, propres ou imbriquées, arbres binaires ou n-aires

Les doublets et la notation pointée permettent donc de représenter n'importe quel *arbre binaire* de valeurs LISP dont les *feuilles* sont des atomes. Etant donné un tel arbre, on peut le voir sous différents points de vue :

— comme un *arbre binaire* de cellules.
— en tant que *liste plate* : c'est le point de vue le plus courant, où l'on ne s'intéresse pas à ce qu'il y a dans le `car`, et on récurse de `cdr` en `cdr`.
— le plus souvent, on pense même à une *liste propre*, c'est-à-dire à une liste plate dont le dernier `cdr` est `nil`. Cependant, comme le `cdr` n'est pas forcément une liste, il est impératif de le vérifier, et le test d'arrêt le plus robuste repose sur `atom`/`consp`, et non pas sur `null`.
— en tant que *liste imbriquée*, l'arbre est une liste plate d'éléments qui peuvent être eux-mêmes des listes, et les traitements récurseront sur ces éléments. En général, les listes imbriquées sont propres. Une liste imbriquée constitue un *arbre n-aire*, où chaque liste plate a pour fils ses éléments.
— enfin, l'arbre peut représenter une expression LISP évaluable : c'est alors une liste imbriquée et propre, comme la définition de `fact` (Fig. 2.1d) : tous les `cdr` sont des listes. L'arbre n-aire des expressions évaluables n'est cependant qu'un sous-arbre de l'arbre n-aire des listes imbriquées car certains noeuds et feuilles de ce dernier ne sont pas évaluables.

2.5.3 Arithmétique étendue

COMMON LISP comporte deux arithmétiques distinctes :
— l'arithmétique flottante habituelle ;
— une arithmétique entière étendue aux rationnels et aux grands nombres (`bignum` s), qui autorise une arithmétique en précision illimitée.

Comme le typage est dynamique, les mêmes fonctions (opérateurs) s'appliquent à toutes ces arithmétiques.

2.5.4 Fonctions

LISP est un langage dit *fonctionnel*, où la seule abstraction procédurale est la *fonction* et où les *fonctions* sont des valeurs dites "de première classe" : il existe donc un type `function`, le prédicat `functionp` associé et des fonctions d'ordre supérieur qui permettent d'appliquer des fonctions.

Il existe enfin une forme syntaxique, `function`, qui permet d'obtenir la valeur fonctionnelle associée à un élément syntaxique, symbole ou λ-expression, et une fonction `apply` pour appliquer une valeur fonctionnelle à des valeurs d'arguments. Voir la section 3.4.

2.5.5 Structures de données diverses

COMMON LISP présente une grande variété de types de données, certains très communs (`array`, `char`, `string`), d'autres plus originaux ou de plus haut niveau, comme `hashtable`. Se reporter au manuel.

Symboles Le type `symbol` est très important : c'est une réification des noms qui permet d'avoir la même implémentation du nom dans toutes ses occurrences, au contraire des `string` qui font que chaque occurrence de la chaîne a sa propre implémentation.

Comme les listes, les symboles appartiennent à la fois à l'espace des données (il faut les `quote` r) et à l'espace des programmes, c'est-à-dire comme nom de variable ou de fonction.

Parmi les symboles, certains sont *constants* : `nil` et `t` qui sont utilisés comme booléens ainsi que les `keyword` s dont le nom est préfixé par « : ». Ces symboles constants ne peuvent pas être utilisés dans un rôle de variable (paramètre de fonction).

Séquences Le type `sequence` est une généralisation des tableaux (`string`, `array`) et des listes plates et propres. Un certain nombre de fonctions très puissantes s'appliquent indifféremment à toutes ces séquences.

2.5.6 Test d'égalité

Dans tous les langages de programmation le test d'égalité pose un problème délicat. Le typage statique résout une partie du problème — mais ça reste toujours un grand problème dans les langages à objets (dit

des *méthodes binaires*) — et la question de l'égalité du *contenant* ou du *contenu* (*physique* ou *logique*) est un problème universel.

LISP propose donc de nombreuses fonctions d'égalité qu'il faut savoir utiliser. On distingue :

— l'égalité physique de représentation (`eq`), qui compare la représentation physique des 2 valeurs, codées sur 32 ou 64 bits ;
— l'égalité numérique (« = ») ;
— une égalité par grand type de données (par exemple, `string-equal` pour les chaînes) ;
— l'égalité physique ou l'égalité d'entiers (`eql`) : c'est la fonction la plus courante ;
— l'égalité logique des listes (`equal`).

Exemples

`(= 123 123)`	→ `t`	
`(= 123 246/2)`	→ `t`	
`(= 123 (/ 246 2))`	→ `t`	
`(= 123 123.0)`	→ `t`	
`(= 123 "123")`	→ `error`	"123" n'est pas un nombre
`(eq 'x 'x)`	→ `t`	un symbole est toujours `eq` à lui-même
`(eq "123" "123")`	→ `nil`	faux pour une chaîne
`(eq "x" 'x)`	→ `nil`	
`(eq nil ())`	→ `t`	
`(eq () ())`	→ `t`	
`(eq '(1 . 2) '(1 . 2))`	→ `nil`	
`(eq (cons 1 2) (cons 1 2))`	→ `nil`	
`(eq '(1 . 2) (cons 1 2))`	→ `nil`	
`(eq 12 12)`	→ `t`	oui pour les petits entiers
`(eq (fact 1000) (fact 1000))`	→ `nil`	pas pour les `bignum` s
`(eql (fact 1000) (fact 1000))`	→ `t`	oui pour les bignums aussi
`(equal '(1 . 2) '(1 . 2))`	→ `t`	2 cellules allouées par `read`
`(equal (cons 1 2) (cons 1 2))`	→ `t`	2 cellules allouées par `eval`
`(equal '(1 . 2) (cons 1 2))`	→ `t`	2 cellules allouées par `read` et `eval`
`((lambda (x) (eq x x)) <expr>)`	→ `t`	une valeur est toujours `eq` à elle-même
`((lambda (x y) (eq x (car (cons x y))))` `<expr1> <expr2>)`	→ `t`	
`((lambda (x y) (eq y (cdr (cons x y))))` `<expr1> <expr2>)`	→ `t`	
`(string-equal "123" "123")`	→ `t`	
`(string-equal 123 "123")`	→ `error`	123 n'est pas un chaîne
`(string-equal 'x "X")`	→ `t`	accepte la conversion des symboles
`(string-equal 'x "x")`	→ `t`	et ne tient pas compte de la casse

Comme on peut le voir avec l'exemple de `string-equal` il est important de regarder les spécifications précises des fonctions ! Toutes ces fonctions d'égalité implémentent bien des relations d'équivalence (transitives, réflexives et symétriques).

EXERCICE 2.1. Vérifier cette dernière affirmation, qui n'est pourtant pas évidente au vu des exemples présentés. □

EXERCICE 2.2. Définir la fonction `fact-first-bignum` qui retourne le plus petit entier `n` tel que `(fact n)` soit un `bignum`. □

EXERCICE 2.3. En utilisant la fonction `ash` (voir le manuel), déterminer le nombre de bits utilisés par l'implémentation des entiers primitifs. □

2.5.7 Gestion mémoire

La manipulation des listes entraîne une allocation de mémoire avec un grain très fin (le doublet) et une fréquence élevée, ainsi qu'un partage fréquent de ces doublets entre liste différentes. La gestion manuelle

de cette mémoire serait insupportable pour le programmeur et elle est assurée en LISP par un mécanisme automatique, le *Garbage Collector* (GC) appelé aussi plus poétiquement *Glaneur de Cellules* et plus couramment, bien que très improprement, *Ramasse-miettes*.

La macro `time` donne une estimation exacte de la quantité de mémoire nécessaire à l'évaluation de son argument.

2.6 Exécution en séquence et effets de bord

LISP est un langage dit *fonctionnel*, et il y a au moins deux sens à ce terme. On a vu le premier : la *fonction* est la seule abstraction procédurale et les fonctions sont des valeurs *de première classe*.

2.6.1 Programmation fonctionnelle vs. effets de bord

Un second sens est plus stylistique : LISP est un langage qui permet un style de programmation *fonctionnel*, proche des fonctions mathématiques et caractérisé par l'absence d'*effets de bord*[6]. Les fonctions effectuent des calculs ou construisent des objets sans modifier leur environnement. L'ordre des calculs est ainsi indifférent et le corps de chaque fonction est réduit à une seule expression.

Dans la pratique, les programmes sont obligés de faire des effets de bord : il faut bien par exemple faire des *entrées-sorties*. D'un autre côté, des calculs purement fonctionnels sont séduisants intellectuellement mais pas toujours très efficaces, et les programmeurs ont souvent recours, même en LISP, à des *affectations*.

2.6.2 Exécution en séquence

A partir du moment où il y a des effets de bord, donc des modifications de l'environnement, l'ordre des évaluations devient crucial et il faut envisager des exécutions en séquence.

En LISP, une liste (non vide) d'expressions évaluables n'est en général pas une expression évaluable ! Il faudrait pour cela que la première expression soit à la fois une fonction et une variable et de toute façon ce ne serait pas une expression évaluant en séquence ses éléments puisque en LISP, contrairement à SCHEME, le premier élément n'est pas évalué.

Donc plusieurs fonctions sont prévues par le langage pour faire ces évaluation en séquences, dont les principales sont :

— `progn`, qui prend un nombre quelconque d'arguments, les évalue séquentiellement, et retourne la valeur du dernier ;
— `prog1`, qui prend un nombre quelconque d'arguments, les évalue séquentiellement, et retourne la valeur du premier ;
— `prog2` est une curiosité qui retourne la valeur du second argument.

Le corps de toutes les fonctions est un "`progn` implicite", donc la véritable syntaxe de `defun` :

```
(defun <nom> <liste-param> <expr-eval>*)
```

Idem pour `lambda` et `let`. Les clauses de `cond` et `case` sont aussi des `progn` implicites. De ce fait, l'emploi explicite de `progn` est rare : on le rencontre en général uniquement dans une des branches d'un `if`.

EXERCICE 2.4. Si `prog1` et `prog2` n'existaient pas, comment pourrait-on simplement les simuler ? En déduire une définition de macro équivalente à ces deux fonctions (cf. Section 3.5). □

2.6.3 Affectation généralisée

L'affectation s'effectue avec la forme syntaxique `setf` dont la syntaxe

<setf>	:=	« **(setf** » {<place> <valeur>}* «) »
<place>	:=	<symbol-var> | « (» <fun-setfable> <expr-eval>* «) »
<valeur>	:=	<expr-eval>

6. En anglais *side effects* : les militaires parlent maintenant d'*effets collatéraux*.

fait alterner des `<place>` qui ne sont pas évalués mais doivent désigner des emplacements où il est licite d'écrire, et des `<valeur>` qui sont des expressions évaluables, qui sont évaluées et dont la valeur est affectée à l'emplacement désigné par `<place>`. La dernière valeur affectée est retournée par `setf`.

Il y a deux grandes sortes de `<place>`. Dans les deux cas, une `<place>` est *licite* s'il serait possible de l'évaluer, c'est-à-dire si on peut remplacer `(setf <place> <valeur>)` par `<place>` sans provoquer d'erreur directe.

Affectations sur les variables On peut affecter une valeur à une variable, c'est-à-dire à un symbole qui joue le rôle de variable. Cela relève souvent d'un 'mauvais style', mais c'est possible. Encore faut-il le faire de façon licite, c'est-à-dire que la variable en soit une, donc que le symbole soit lié dans l'environnement courant.

Remarque 2.8. L'affectation est la chose la plus difficile à comprendre donc à éviter le plus possible. L'affectation d'une variable consiste à modifier l'environnement courant : c'est un effet de bord qui n'est que local, quand l'environnement courant est "refermé", l'effet de l'affectation disparaît (sauf si l'environnement a été capturé par une *fermeture* : mais c'est encore plus difficile à comprendre (cf. Section 3.4.3)). Ainsi, dans `(let ((x ...)) ... (setf x <expr-eval>))`, l'affectation termine le `progn` implicite du corps du `let` : elle ne sert donc à rien et `(let ((x ...)) ... <expr-eval>)` aurait exactement le même effet. La remarque vaut aussi pour `defun` ou `lambda`.

Remarque 2.9. La forme spéciale `setq` est une version obsolète (mais qui marche toujours) de `setf` pour les variables.

Affectations sur des champs d'objets construits Toutes les structures de données construites (listes, chaînes, tableaux, structures et objets) sont implémentées par un certain nombre de champs qui sont en général accessibles en lecture et écriture. Si `<place>` désigne un tel champ, `(setf <place> <valeur>)` est licite.

Par exemple, sur les cellules, il est possible d'affecter les `car` et `cdr` — c'est le principe de la *chirurgie de listes* et des *fonctions destructrices* (Section 3.3.3). On pourra donc écrire :

```
(setf (car <expr-eval>) <valeur>)
```

et pour que ce soit licite, il faut que l'évaluation de `<expr-eval>` retourne une cellule (et pas simplement une liste : `car` et `cdr` sont étendus à la liste vide mais c'est un abus de notation). Noter que `<place>`, c'est-à-dire `(car <expr-eval>)`, n'est pas évalué mais que `<expr-eval>` l'est.

Remarque 2.10. La forme `setf` s'applique à un certain nombre de fonctions prédéfinies, dite *`setf`-able*. De façon générale, si le premier élément de `<place>` est une *macro*, la macro est expansée : le résultat final de l'expansion doit être une fonction `setf`-able. Il est aussi possible de définir son comportement pour des fonctions définies par l'utilisateur, mais cela reste assez délicat.

2.7 Erreurs de parenthésage

Il ressort de la syntaxe des expressions évaluables (Section 2.2.1) et de la sémantique de l'évaluation que, dans une expression évaluable :

1. un symbole qui joue le rôle de **variable n'est jamais précédé d'une ouvrante** ;
2. un symbole qui joue le rôle de **fonction est toujours précédé d'une ouvrante** ;
3. **une expression évaluable ne commence jamais par deux ouvrantes**, sauf si c'est une λ-expression [7].

A ces trois règles correspondent 3 types d'erreur que tout programmeur LISP débutant doit apprendre à reconnaître. Soit la fonction

```
(defun foo (x) ....)
```

Chacune de ces erreurs de parenthésage produira l'un des messages d'erreur suivants :

7. La syntaxe de `let` ou `labels` n'est pas un contre-exemple : si deux ouvrantes se succèdent dans une sous-expression, cette sous-expression n'est en fait pas évaluable (cf. Sections 2.4.4 et 3.2.1).

1. *fonction indéfinie* `X` : une parenthèse ouvrante erronée a été mise devant `x`, par exemple dans `(foo (x))` au lieu de `(foo x)` ;
2. *variable indéfinie* `FOO` : une parenthèse ouvrante a été omise devant `foo` (et souvent placée après) ; le programmeur débutant écrit `foo(x)` au lieu de `(foo x)` :
3. `(FOO 3)` *n'est pas une fonction* ou bien `(FOO 3)` *n'est pas une expression* `LAMBDA` ; dans ce dernier cas, le message d'erreur varie suivant qu'il s'agit d'une évaluation contenant l'expression erronée ou d'une définition de fonction (`defun`) : le programmeur a écrit `((foo 3))` au lieu de `(foo 3)` ;

Les erreurs se combinent : le programmeur écrit `foo ((cdr x))` au lieu de `(foo (cdr x))`. Si c'est dans un `defun`, le premier message d'erreur sera la deuxième version du 3. Au *toplevel*, le première erreur sera la 2. Si la correction n'est pas complète, par exemple ajout et non déplacement de parenthèse, le message suivant sera la première version de 3. Dans le cas de l'erreur 2, si le programmeur corrige `foo(x)` en rajoutant un ouvrante, il tombera ensuite sur l'erreur 1 !

Une quatrième erreur résulte de l'absence de « `'` » (ou `quote`) et entraîne une tentative d'évaluation d'une expression évaluable : l'un des 3 messages précédents peut en résulter. Par exemple, la tentative d'évaluation de `(1 2 3 4)` se traduira par le message *fonction indéfinie* `1`.

Remonter de ces messages d'erreur à l'erreur elle-même doit être automatique, surtout si le programmeur s'appuie sur les fonctionnalités de *debug* (Section 5.5).

2.8 Le système LISP

LISP n'est pas simplement un langage, c'est un *système* qui inclut divers éléments :
— le langage lui-même,
— un *interprète*,
— une bibliothèques de structures de données et de fonctions,
— un *compilateur*,
— une extension objet CLOS,
— diverses interfaces avec le monde extérieur (langage C, interfaces graphiques, SGBD et SQL, etc.)
En dehors du langage, de la bibliothèque de fonctions et de CLOS qui sont tous normalisés ANSI, tout ce qui concerne le système LISP dépend de l'implémentation précise. Pour plus de détails sur le système utilisé en cours, voir le chapitre 5.

2.8.1 L'interprète et la boucle de *top-level*

LISP est un langage *interprété*. Cela signifie qu'il n'est pas nécessaire de compiler les programmes avant leur exécution et que les fonctions peuvent être testées au fur et à mesure de leur définition. Le cœur de l'interprète est constitué par la boucle dite de *toplevel* ou boucle `read-eval-print`, qui peut être schématisée par l'expression suivant :

```
(loop while t
      do
      (print (eval (read))))
```

C'est une boucle infinie (`loop while t`), qui lit une expression LISP sur l'entrée standard (fonction `read`), évalue l'expression (fonction `eval`) et imprime (fonction `print`) sur la sortie standard ce que l'évaluation retourne.

Boucle de debug

La boucle de *toplevel* se déroule infiniment jusqu'à ce qu'une situation anormale soit rencontrée, qui provoque le signalement d'une exception. Une nouvelle boucle s'ouvre alors, dans l'environnement de l'expression qui a provoqué l'exception, ce qui permet au programmeur de localiser l'erreur, de consulter l'état des variables, de se déplacer dans la pile, etc. Voir Section 5.5.

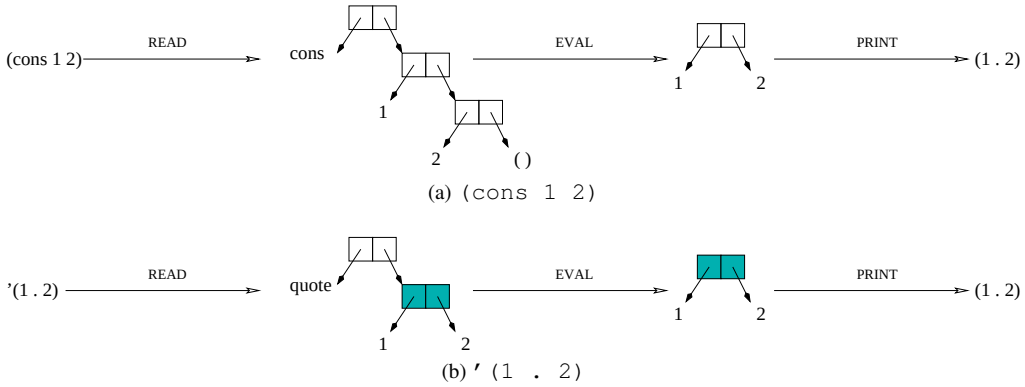


FIGURE 2.2 – Rôles de `read` et `eval` : les cellules blanches sont toutes 2 à 2 distinctes mais les deux cellules grisées représentent la même cellule.

Le chargement

L'interprète permet de taper directement des définitions de fonctions dans la boucle de *toplevel*, mais il est bien sûr nécessaire de disposer de fonctionnalités permettant de faire ces définitions dans un fichier, puis de les *charger* sous l'interprète. Cela se fait par la fonction `load`, qui est une variante sans `print` de la boucle de *toplevel*.

```
(loop while t
      do
      (eval (read)))
```

Bien entendu, la lecture se fait maintenant sur un canal ouvert en lecture sur le fichier considéré et la fin de fichier provoque l'interruption de la boucle.

La fonction `load` prend en argument le nom du fichier : `(load "monfichier.lisp")`.

Rôles de `read` et `eval`

Aussi bien `read` que `eval` ont une action d'allocation mémoire et il est indispensable de savoir distinguer le rôle de chacun.

Les expressions `(cons 1 2)` et `'(1 . 2)` sont évaluables et se ressemblent beaucoup : les deux retourneront `(1 . 2)`, qui sera imprimé sur la sortie courante. Dans les deux cas, il y a une cellule dont le `car` contient 1 et le `cdr` contient 2. Mais la cellule est créée par `eval` dans le premier cas, et par `read` dans le deuxième. La figure 2.2 illustre cela.

Aussi, l'évaluation de l'expression `(loop repeat 10 do (cons 1 2))` va allouer successivement 10 cellules, alors que `(loop repeat 10 do '(1 . 2))` n'en alloue aucune : la seule cellule est allouée à la lecture. Dans un vrai programme, la cellule `quot` ée est une constante, partagée par toutes les listes qui sont `cons` truites à partir d'elle. En général cela ne pose pas de problème. Cependant, l'utilisation de fonctions *destructrices* sur les listes peut avoir un effet dramatique lorsqu'elles s'appliquent sur une cellule `quot` ée.

Le rôle de `read` doit aussi être distingué dans tous les cas de constructions syntaxiques particulières, par exemple pour « `'` » et « `#'` ». C'est `read` qui génère la cellule dont le `car` est `quote` (Figure 2.2b) et `eval` ne voit pas le caractère « `'` ».

Remarque 2.11. La fonction `print` fait l'inverse de `read` : plutôt que d'imprimer en toute lettre `(quote x)`, elle reconnaît le symbole et restitue le caractère « `'` », pour donner « `'x` ».

Remarque 2.12. Diverses optimisations sont possibles à la lecture et peuvent constituer des paramétrages implicites ou explicites des chaque interprète. Une optimisation très efficace mais potentiellement dangereuse consiste à identifier les littéraux de type liste ou chaîne. Deux occurrences successives de la même chaîne (`"toto"`) ou de la même paire (`(1 . 2)`) seront alors `eq`, mais gare au programmeur qui veut faire de la chirurgie sur l'une des deux.

2.8.2 Le compilateur-chargeur

LISP est un langage interprété — pourvu d'un interprète et de la boucle `read-eval-print` — mais il dispose aussi d'un *compilateur* et *chargeur*, qui s'intègre dans le système de façon plus ou moins transparente suivant les systèmes.

Le rôle du compilateur est de générer du *code machine*, directement ou indirectement, à partir d'un fichier ou d'une fonction isolée. Le rôle du chargeur est de *lier* ce code machine au code du processus LISP. L'ensemble est suffisamment transparent pour que l'interprète se débrouille pour appeler, pour chaque fonction, le code interprété ou le code compilé suivant le cas, et pour que les codes compilé et interprété puissent s'appeler mutuellement.

Suivant les systèmes LISP, cette compilation est mise en œuvre de façon automatique ou doit être explicitement demandée par le programmeur. Dans ce dernier cas, il dispose de fonctions (voir Chapitre 5) pour compiler un fichier, qui génère du code dans une *machine virtuelle*. Appliquée à des fichiers compilés, la fonction `load` décrite ci-dessus provoque la compilation du code de la machine virtuelle en du code machine et sa liaison.

A quoi sert le compilateur et quand faut-il s'en servir ? De façon générale, il produit une accélération assez sensible de l'exécution et détecte des erreurs. Comme le typage est dynamique, il ne faut pas trop compter sur la compilation pour diagnostiquer beaucoup d'erreurs — même si elle peut être très informative — et il est impératif de tester aussi les fonctions dès leur définition. Par ailleurs, le code compilé est en général indébuguable, donc le diagnostic nécessite une exécution où le code qui signale l'exception est interprété.

Au total, la compilation doit être réservée à du code éprouvé, dans une étape d'exploitation des programmes. Dans les exercices qui suivent, on s'en servira aussi pour comparer les deux systèmes d'exécution, compilé ou interprété.

Remarque 2.13. En CLISP, la compilation des fonctions s'effectue uniquement à la demande du programmeur, soit par appel explicite des fonctions de compilation, soit par configuration de l'environnement pour que le code LISP soit compilé à la volée. Dans d'autres implémentations, le fonctionnement peut être différent.

2.9 Différence entre LISP et SCHEME

Cette section n'est destinée qu'aux étudiants qui connaissent déjà SCHEME *ou* LISP.

COMMON LISP et SCHEME sont deux dialectes de la grande famille des langages LISP. Les ressemblances sont très grandes mais les différences de détail trop nombreuses pour être listées ici.

Espaces de noms de fonctions et variables La différence principale est la suivante.

En LISP, les noms de variables et les noms de fonctions ne sont pas dans le même espace de noms. En évaluant `(foo x y)`, l'évaluateur va chercher la valeur de `x` et `y` dans l'*environnement lexical* courant, et la fonction associée à `foo` dans l'*environnement global des fonctions*. Ainsi, la définition suivante de factorielle est tout à fait licite, même si on ne saurait conseiller un tel style.

```
(defun fact (fact)
    (if (= fact 0)
        1
      (* fact (fact (- fact 1)))))
```

En revanche, en SCHEME, fonctions et variables se partagent le même espace de nom et la forme syntaxique `defun` est remplacée par `define` avec une syntaxe légèrement différente :

```
(define (fact fact)
    (if (= fact 0)
        1
      (* fact (fact (- fact 1)))))
```

La syntaxe est correcte mais, à la suite de cette définition, l'évaluation de `(fact 4)` se fait dans un environnement contenant $\{FACT \rightarrow$ `(lambda (fact) (if ...))` $\}$ qui provoque l'évaluation du corps

de la fonction `fact` dans un environnement contenant $\{FACT \rightarrow 4\}$. L'évaluation de l'expression `(fact (- (fact 1)))` provoque alors une erreur puisque `4` n'est pas une fonction.

Par ailleurs, `defun` est toujours global, alors que `define` est toujours local : il permet de rajouter une définition dans l'environnement courant. Donc, `define` joue le rôle de `defun` au *top-level* mais de `labels` et de `let` à l'intérieur d'une définition de fonction.

EXERCICE 2.5. Pour un non spécialiste de SCHEME (l'auteur de ces lignes par exemple), le rôle de `define` est ambigu. Il permet de rajouter une liaison à l'environnement courant, que ce soit pour une fonction ou une variable. Mais quel est l'effet de cet ajout sur les fermetures qui auraient préalablement capturé l'environnement ? Faire des essais et lire les documentations *ad hoc*. □

Fonctions d'ordre supérieur Le fait qu'il y ait un ou deux espaces de noms est important mais a un effet pratique mineur puisqu'il est fortement déconseillé d'utiliser les mêmes noms pour des variables et des fonctions. Un effet non négligeable apparaît néanmoins dans les fonctions d'ordre supérieur. En SCHEME, on pourrait écrire `fact` comme suit :

```
(define (fact foo n)
    (if (= n 0)
        1
      (* n (foo foo (- n 1)))))
```

à condition de l'utiliser en appelant `(fact fact 4)`, au lieu de `(fact 4)`.

En LISP, la situation se complique car il faut faire passer la fonction de l'espace des noms de fonctions à celui des variables, et réciproquement :

```
(defun fact (foo n)
    (if (= n 0)
        1
      (* n (apply foo foo (- n 1) ( )))))
```

L'appel récursif de `fact` désigné par la variable `foo` nécessite de dire explicitement que l'on veut *appliquer* une valeur de type `function` : c'est le rôle de la fonction `apply`. On utilisera ensuite la fonction ainsi définie en l'appelant par `(fact (function fact) 4)`, en utilisant la forme syntaxique `function` qui retourne la *valeur fonctionnelle* de son argument.

Le couple `function` et `apply` permet ainsi la traversée du miroir entre les 2 espaces de noms, d'une façon parallèle à celle de `quote` et `eval`. D'ailleurs, `function` s'abrévie par « `#'` » comme `quote` par « `'` ». Pour plus de détails, voir la section 3.4.

Macros Il n'y a pas non plus de macros en SCHEME, en tout cas sous la forme des macros globales de COMMON LISP. Il existe cependant des macros lexicales, introduites par `let-syntax` et `letrec-syntax`.

Conventions de noms Les conventions de nommage des fonctions diffèrent légèrement. En SCHEME les prédicats — et les fonctions booléennes — sont suffixés par « `?` » au lieu de « `p` » en LISP. De la même manière, les fonctions qui font des effets de bord sont suffixées par « `!` » — par exemple, `set!` remplace `setf`.

Bibliographie Une fois que ces différences sont bien identifiées, il est tout à fait possible d'utiliser pour l'apprentissage de LISP les livres dédiés à SCHEME, comme [ASS89] ou [Que94], ainsi que les notes de cours des années de Licence. La consultation du manuel de référence [Ste90] en ligne sera juste nécessaire pour vérifier le nom et la syntaxe exacte des fonctions.

Il restera néanmoins une spécificité de SCHEME inutilisable en COMMON LISP, le fameux `call/cc`, *call with current continuation*, qui n'a pas d'équivalent.

3

Programmation en LISP

Ce chapitre développe les grands principes de la programmation en LISP, en étudiant successivement la récursion, les usages avancés des fonctions, la programmation des listes, ainsi que les transformations source-à-source avec des macros.

3.1 Programmation récursive

La récursion est donc le style de programmation usuel en LISP. C'est à la fois simple et intuitif, mais il est bon d'en comprendre les principes et les limites.

3.1.1 Récursions simples

Une récursion simple est une fonction récursive dont chaque activation ne conduit qu'à un seul appel récursif — il peut y avoir plusieurs appels récursifs dans le code, mais un seul est activé à chaque fois. Les récursions sur les listes plates sont en général simples.

Reprenons l'exemple de factorielle. La définition n'est pas tirée d'un chapeau comme un lapin : elle vient au contraire de la définition mathématique la plus directe :

$$0! = 1$$
$$n! = n(n-1)! \quad \text{si} \quad n > 0$$

La définition LISP est strictement isomorphe :

```
(defun fact (n)
    (if (= n 0)                 ; test d'arrêt
        1
      (* n
         (fact (- n 1)))))      ; appel récursif
```

Bien entendu, la récursion ne se terminera qu'à la condition qu'il y ait un test d'arrêt, et que la définition soit inductive et donc permette un raisonnement par induction (ou récurrence). Dans le cas de `fact`, ce n'est vrai que pour les entiers positifs.

Exemples sur les listes

Considérons deux autres exemples de fonctions récursives, cette fois-ci sur les *listes plates*.

La fonction `length` compte le nombre d'éléments d'une liste plate ou, de façon équivalente, son nombre de cellules au premier niveau.

```
(defun length (ll)
    (if (atom ll)                ; test d'arrêt
        0
      (+ 1
         (length (cdr ll)))))    ; appel récursif
```

Une autre fonction classique est `member` qui recherche un élément dans une liste.

```
(defun member (x ll)
    (if (atom ll)                ; test d'arrêt
        nil
      (if (eql x (car ll))       ; test d'égalité
          ll                     ; retourne la sous-liste
        (member x (cdr ll)))))   ; appel récursif
```

Remarque 3.1. Attention, la plupart des fonctions des exercices font partie de la bibliothèque standard : pour tester leur correction, il faut changer leur nom pour ne pas tout casser ! Comme plusieurs versions sont souvent demandées et que choisir un nom est difficile, le mieux est de les indicer ou suffixer.

Attention encore ! La plupart des exercices concernent des fonctions récursives : c'est bien de changer le nom de la fonction, mais il faut aussi le faire dans les appels récursifs, sinon vous ne testerez que la première application de la fonction.

Remarque 3.2. On remarque ici que la fonction retourne la sous-liste commençant par l'élément cherché car c'est plus informatif que de retourner `t` : on peut ainsi s'intéresser à l'ordre de 2 éléments ou au nombre d'occurrences d'un élément. Le test d'arrêt utilisé est `eql` : c'est le plus courant. On verra comment changer de test dans le manuel COMMON LISP (voir aussi la Section 3.4.2).

Remarque 3.3. Ces deux exemples de récursion sur des listes plates suivent un schéma de test d'arrêt qui est quasiment impératif :

- le test d'arrêt utilise la fonction `atom` qui est plus générale que `null` : les fonctions considérées ne produiront ainsi jamais d'erreur. Si nous avions utilisé `null`, cela n'aurait marché que pour les *listes propres* ;
- les tests d'arrêt doivent avoir une complexité en $\mathcal{O}(1)$ (temps constant) : si le test calculait la longueur de la liste courante avec la fonction `length`, il serait en $\mathcal{O}(n)$ (linéaire dans la longueur de la liste) et la fonction serait elle-même en $\mathcal{O}(n^2)$ (quadratique). Tous les tests de type sont bien sûr en temps constant et peuvent être utilisés comme tests d'arrêt.
- le test d'arrêt consulte la valeur courante du paramètre, c'est-à-dire la cellule courante dans le cas où c'est une cellule, et non pas le contenu hypothétique de son `cdr` ; c'est un principe absolu : il faut regarder où on est, pas où on veut aller !

EXERCICE 3.1. Définir la fonction `remove` qui 'enlève' d'une liste toutes les occurrences d'une valeur, c'est-à-dire qui retourne une copie de la liste en sautant les occurrences en question : `(remove 3 '(1 2 3 4 3 5))` → `(1 2 4 5)`. Est-ce une récursion simple ? □

EXERCICE 3.2. Définir la fonction `last` qui retourne la dernière cellule d'une liste plate. Attention ! C'est l'une des rares exceptions à la dernière partie de la Remarque 3.3. □

EXERCICE 3.3. Définir la fonction `nth` qui prend 2 paramètres, un nombre n et une liste, et retourne le n-ième élément de la liste. □

EXERCICE 3.4. Définir la fonction `nthcdr` qui est similaire à `nth` mais retourne le n-ième `cdr` de la liste. □

EXERCICE 3.5. Définir la fonction `position` qui est similaire à `member` mais retourne la position de l'élément cherché, ou `nil` s'il est absent. □

EXERCICE 3.6. Définir la fonction `count` qui compte le nombre d'occurrences de son premier élément dans le second. Faire deux versions, suivant qu'on utilise `member` ou pas. □

3.1.2 Récursions terminales et enveloppées

La comparaison des fonctions `length` et `member` est très instructive. Comment les évaluations de `(length '(1 2 3 4))` et `(member 5 '(1 2 3 4 5 6))` s'exécutent-elles ? Le tableau suivant montre les valeurs successives du paramètre `ll` de chacune des 2 fonctions, ainsi que la valeur retournée :

length		member	
(1 2 3 4)	→ 4	(1 2 3 4 5 6)	→ (5 6)
(2 3 4)	→ 3	(2 3 4 5 6)	→ (5 6)
(3 4)	→ 2	(3 4 5 6)	→ (5 6)
(4)	→ 1	(4 5 6)	→ (5 6)
()	→ 0	(5 6)	→ (5 6)

L'exécution se lit donc de la façon suivante : les appels récursifs se font en descendant dans la colonne de gauche et les valeurs retournées sont remontées dans la colonne de droite. Cela simule ce qui se passe dans la pile d'exécution.

La descente des deux fonctions est similaire, chaque appel récursif prenant le reste (`cdr`) du paramètre. En revanche, les remontées sont complètement différentes : `length` ajoute 1 à chaque étage, alors que `member` remonte toujours la même valeur. En fait, `member` pourrait retourner sa valeur sans faire étape à chaque étage. En pratique, cela signifie que `length` est obligé de consommer de la pile, alors que `member` pourrait s'en passer.

Si l'on examine maintenant le code des deux fonctions, on constate que ce comportement différent à l'exécution se traduit dans la structure textuelle : l'appel récursif de `length` est *enveloppé* par un appel de la fonction `+`, alors que `member` n'a pas d'enveloppe : le retour de l'appel récursif est donc directement retourné à l'étage au-dessus. NB. les `if` ne font pas une enveloppe.

On parlera donc de *récursion enveloppée* (`length`) et de *récursion terminale* [1] (`member`). D'un point de vue pratique, le programmeur doit savoir qu'une récursion enveloppée consomme de la pile alors que ce n'est *pas forcément* le cas pour les récursions terminales — mais cela dépend des implémentations. Par exemple, la fonction `fact` présente une récursion enveloppée et `(fact 100000)` — qui est possible parce que la fonction a une complexité linéaire et grâce à l'arithmétique en précision infinie — va vraisemblablement casser la pile. Ce problème n'est pas une exclusivité LISP, il existe dans tous les langages, en particulier en C.

3.1.3 Terminalisation d'une récursion enveloppée

Si l'on veut s'abstraire des exemples particuliers, une fonction récursive enveloppée simple a la structure suivante :

```
(defun foo-e (ll)
    (if (<arret> ll)                          ; test d'arrêt
        <init>                                ; valeur initiale
      (<enveloppe>                            ; enveloppe
        (foo-e (<suivant> ll)))))             ; appel récursif
```

et on utilise la fonction avec une expression de la forme `(foo-e <donnee>)`. NB. Il faudrait compliquer un peu le schéma pour tenir compte de paramètres supplémentaires, mais cela ne poserait pas de problème.

Cela s'applique directement à `length`, à la petite transformation près de l'appel récursif enveloppé en `((lambda (x) (+ 1 x)) (length (cdr ll)))` pour que l'enveloppe soit une fonction, la λ-fonction.

En admettant qu'un schéma de récursion terminal équivalent soit possible, ce schéma nécessiterait de faire les calculs en descendant, en appliquant l'enveloppe sur un paramètre supplémentaire qui serait descendu en parallèle avec le paramètre de récursion.

```
(defun foo-t (ll <retour>)
    (if (<arret> ll)                          ; test d'arrêt
        <retour>                              ; valeur terminale
      (foo-t (<suivant> ll)                   ; appel récursif
             (<enveloppe> <retour>))))
```

et l'on utiliserait la fonction avec une expression de la forme `(foo-t <donnee> <init>)` : la valeur initiale réapparaît ici.

1. En anglais, *tail recursion*.

EXERCICE 3.7. Appliquer ce schéma de terminalisation de la récursion enveloppée à `fact` et `length`. □

On constate que cela marche parfaitement bien. Appliqué à `length`, cela donne la séquence d'appels et de valeurs retournées suivante :

length-t	
(1 2 3 4)	0 → 4
(2 3 4)	1 → 4
(3 4)	2 → 4
(4)	3 → 4
()	4 → 4

Mais ces fonctions sont des cas particuliers : faire des additions ou des multiplications en montant ou en descendant donnera toujours le même résultat parce que ces opérations sont *associatives* et *commutatives* et `<init>` est leur *élément neutre*. Quand l'enveloppe n'utilise pas une opération associative et commutative, avec `cons` par exemple dès qu'il va s'agir de construire des listes, que va donner la transformation ? Considérons la fonction `copylist` qui fait la copie d'une liste plate.

```
(defun copylist (ll)
    (if (atom ll)
        ll
      (cons (car ll)
            (copylist (cdr ll)))))
```

EXERCICE 3.8. Que donne la transformation et que fait la fonction `copylist-t` que l'on obtient ? Faire le tableau des passages de paramètres et de valeurs retournées, comme pour `length-t`. Existe-t-il une version terminale de `copylist` ? Quelle en est la complexité ? □

EXERCICE 3.9. La fonction `fact-t` est maintenant récursive terminale. Il devrait donc être possible de calculer `(fact-t 100000)` sans casser la pile. Est-ce le cas ? Compiler ensuite la fonction `fact-t` avec `compile` (Section 5.7) et refaire le test. Qu'en déduire ? □

EXERCICE 3.10. Définir la fonction `reverse` qui fait une copie des éléments d'une liste, en sens inverse. Examiner les versions enveloppées et terminales. □

EXERCICE 3.11. Définir la fonction `makelist` qui prend en argument un entier positif et retourne la liste des entiers de 1 à cet entier (dans un ordre ou dans un autre : essayer les deux). □

Remarque 3.4. Après les termes de 'λ-fonction' et 'λ-expression' on peut parler de 'λ-*abstraction*', qui consiste à 'abstraire' une expression pour en faire une fonction. Soit une expression évaluable `<expr>` qui contient une sous-expression évaluable `<x>`. Sous certaines hypothèses, `<expr>` est exactement équivalent à `((lambda (y) <expr>`$_{[\texttt{<x>/y}]}$ `<x>)`, où les crochets désignent une substitution de `y` à `<x>`.

Les conditions sont au nombre de 2 : le symbole `y` ne doit pas déjà être une sous-expression évaluable de `<expr>` et les expressions `<expr>` et `<x>` doivent être *fonctionnelles*, c'est-à-dire qu'elles ne dépendent pas de l'ordre de l'évaluation de leurs sous-expressions.

3.1.4 Récursions doubles

Une récursion double est une fonction récursive dont *une* activation peut conduire à *deux* appels récursifs. Les récursions sur les arbres binaires de cellules sont en général doubles.

Fibonacci Un exemple simple de récursion double est la fonction de Fibonacci.

```
(defun fibo (n)
    (if (<= n 1)
        1
      (+ (fibo (- n 1)) (fibo (- n 2)))))
```

On constate que la récursion est enveloppée, puisque chaque appel récursif est enveloppé par l'appel d'une fonction qui a l'autre appel en paramètre. Plus généralement, toute récursion double est enveloppée et le schéma de terminalisation ne marche pas puisque l'enveloppe contient un appel récursif.

EXERCICE 3.12. Il existe néanmoins une version terminale de Fibonacci : trouvez-la. □

EXERCICE 3.13. La récursion se prête bien à l'évaluation de la complexité des fonctions. Quelle est celle de Fibonacci dans la version enveloppée présentée ci-dessus ? (Une borne inférieure est plus facile à obtenir qu'une formule exacte.) Mesurer le temps d'exécution de `fibo` à l'aide de la macro `time`. A partir de 2 mesures, en déduire la durée d'exécution de `(fibo 50)`. Le prof sera-t-il toujours vivant à la terminaison de la fonction ? Et l'univers ? Même question pour la version terminale ? □

Fonctions sur les arbres binaires La fonction `length` sur les listes plates a son pendant dans la fonction `size` sur les arbres binaires de cellules qui retourne le nombre de cellules d'une expression LISP :

```
(defun size (expr)
    (if (atom expr)
        0
      (+ 1 (size (car expr)) (size (cdr expr)))))
```

Remarque 3.5. Ici aussi on remarque que le test d'arrêt — qui ne peut pas être une autre fonction que `atom` — porte sur l'expression courante, c'est-à-dire le nœud courant de l'arbre. On ne regarde jamais ce qu'il y a dans le `car` et le `cdr` du nœud courant ! (cf. Remarque 3.3)

Remarque 3.6. La principale difficulté des récursions sur les arbres est de déterminer la fonction à utiliser pour 'recoller' les résultats des deux appels récursifs. Toutes les fonctions de récursion d'arbres binaires peuvent avoir ce schéma :

```
(defun foo (expr)
    (if (atom expr)
        <init>
      (<glue> (foo (car expr)) (foo (cdr expr)))))
```

Mais quel doit être leur fonction `<glue>` ?

EXERCICE 3.14. Définir la fonction `leaf-number` qui calcule le nombre de feuilles — c'est-à-dire d'atomes — d'un arbre. □

EXERCICE 3.15. Définir la fonction `equal` qui teste si deux expressions sont `eql` ou sont des cellules dont les `car` et `cdr` sont respectivement `equal`. □

EXERCICE 3.16. Définir la fonction `copytree` qui copie une expression. □

EXERCICE 3.17. Définir la fonction `subst` qui substitue son premier paramètre par le second dans le troisième qui est une expression quelconque. □

EXERCICE 3.18. Définir la fonction `maketree` qui qui prend un paramètre n entier positif et génère un arbre équilibré avec 2^n feuilles numérotées de 1 à 2^n. □

Notation pointée La notation pointée nécessite quelques fonctions d'entrée-sortie.

EXERCICE 3.19. Définir la fonction `print-full-dot` qui qui prend en paramètre une expression quelconque et l'imprime en *notation totalement pointée*. □

EXERCICE 3.20. Définir la fonction `print-min-dot` qui qui prend en paramètre une expression quelconque et l'imprime en *notation pointée minimale*. □

EXERCICE 3.21. Définir la fonction `proper-tree-p` qui qui prend en paramètre une expression quelconque et retourne `t` si cette expression s'écrit sans aucun point et `nil` sinon. □

EXERCICE 3.22. Définir la fonction `invert-tree` qui qui prend en paramètre une expression quelconque (un arbre) et retourne une copie de cet arbre où `car` et `cdr` ont été inversés. □

Remarque 3.7. Pour les fonctions d'entrée-sortie, on consultera la section 4.4 et le manuel en ligne. Noter que les entrées-sorties font des *effets de bord* qui nécessitent quelques précautions LISP décrites Section 2.6.1.

Récursions semi-terminales Le schéma de terminalisation des récursions enveloppées ne permet pas d'obtenir une récursion terminale. Il est cependant possible de l'appliquer pour rendre terminal l'un des deux appels récursifs. On appellera ça une récursion *semi-terminale*.

EXERCICE 3.23. Appliquer le schéma de terminalisation à l'une des deux récursions de `size`. □

EXERCICE 3.24. Définir la fonction `tree-leaves` qui fait la liste des feuilles d'un arbre. Quelle fonction `<glue>` faut-il utiliser ? Faire ensuite une récursion *semi-terminale* : par quoi faut-il remplacer `<glue>` ? Suivant la version, l'ordre des feuilles dans la syntaxe de l'expression est-il respecté dans la liste obtenue ? Comparer l'efficacité des différentes versions en les appliquant à de gros arbres, créés par `maketree` et en utilisant la macro `time`. □

Fonctions sur les arbres n-aires Tous les arbres ne sont pas binaires et une expression LISP peut aussi être considérée comme un arbre n-aire, dans lequel une liste est un nœud dont les éléments sont les fils.

La structure de la récursion est alors plus complexe, avec deux fonctions `foo` et `map-foo` qui s'appellent récursivement : la première traite une expression, alors que la seconde traite une liste d'expressions.

```
(defun foo (expr)
    (if (atom expr)
        <init>
      (map-foo expr)))

(defun map-foo (lexpr)
    (if (atom lexpr)
        ()
     (<glue> (foo (car expr)) (map-foo (cdr expr)))))
```

Une telle récursion repose sur une représentation dite premier-fils/frère des arbres, où la fonction `map-foo` fait appel à `foo` sur le premier fils du nœud courant puis récurse sur son frère. C'est typiquement la récursion utilisée pour parcourir les sous-expressions évaluables d'une expression évaluable LISP, et le polycopié de compilation [DL13] en donne de nombreux exemples, en particulier au Chapitre 6.

3.2 Fonctions locales et fonctions d'arité variable

3.2.1 Fonctions locales

Tel qu'il a été présenté, le schéma de terminalisation des récursions enveloppées pose un problème d'*interface fonctionnelle* (synonyme d'API). La fonction `length` est bien spécifiée : elle prend en argument une liste et en retourne la longueur. Être obligé de l'appeler par `(length '(1 2 3) 0)` sous prétexte qu'elle a été écrite en récursion terminale n'est pas raisonnable. Écrire une fonction globale `length-t` et définir `length` par

```
(defun length (ll)
    (length-t ll 0))
```

n'est pas beaucoup mieux car `length-t` reste toujours accessible.

La bonne solution consiste à définir des fonctions locales avec la forme syntaxique `labels` dont la syntaxe est :

<labels>	:=	« **(labels** (» <local-fun>+ «) » <expr-eval>+ «) »
<local-fun>	:=	« (» <symbol> <list-param> <expr-eval>+ «) »

Appliqué à factorielle, cela donne :

```
(defun fact-l (n)
    (labels ((fact-t (m r)                  ; fonction locale
                     (if (= m 0)
                         r
                       (fact-t (- m 1)
                               (* m r)))))
        (fact-t n 1)))                      ; corps du labels
```

Le premier paramètre de `labels` est une liste de définitions de fonctions — d'où la double « (» : ces fonctions peuvent être récursives et s'appeler les unes les autres. Le reste constitue le corps du `labels`. Une version plus simple, `flet`, définit des fonctions non récursives, qui ne peuvent pas s'appeler entre elles.

Remarque 3.8. Les fonctions locales produisent les mêmes imbrications d'environnement que les λ-fonctions (Section 2.3.2) : dans l'exemple, la variable `n` est donc connue dans `fact-t`. C'est souvent pratique — la fonction locale peut accéder aux paramètres de la fonction globale sans avoir besoin de les répéter comme paramètres locaux — mais cela peut avoir aussi des incidences négatives. Ainsi, dans le cas de `fact`, on avait le choix entre nommer le paramètre local `m` comme le global `n` — mais ce n'est pas très bien de réutiliser les mêmes noms, au risque de la confusion — ou au contraire utiliser des noms différents, comme ici, avec le risque d'oublier que l'on a changé le nom et d'utiliser par erreur le nom global. Bien entendu, si on remplace une occurrence de `m` par `n`, on sera déçu par le résultat.

EXERCICE 3.25. Rajouter un test de type à `fact-l` et à `fact`, puis comparer les efficacités respectives. □

EXERCICE 3.26. Ecrire `fact` avec 2 fonctions locales récursives qui s'appellent l'une l'autre. Peut-on économiser l'un des 2 tests d'arrêt ? □

EXERCICE 3.27. Écrire `fibonacci` avec 3 fonctions locales récursives dont chacune appelle les 2 autres. □

EXERCICE 3.28. Écrire `fibo` avec 2 fonctions locales récursives `f1` et `f2` qui sont elles-mêmes chacune définies avec 2 fonctions locales récursives `f11` et `f12` dont chacune récursivement appelle l'autre fonction interne et l'autre fonction externe : `f11` appelle `f12` et `f2`. □

EXERCICE 3.29. Étendre ensuite le dernier exercice en passant les constantes (0, 1, 2) comme paramètres des fonctions locales. Chaque fonction locale peut alors se servir de son propre paramètre ou de celui de la fonction englobante. □

Remarque 3.9. On se convainc aisément de la correction de ces fonctions proliférantes de la façon suivante : (1) une fonction récursive globale `foo`

```
(defun foo (x)
   (if ... (foo ...)))
```

est équivalente à la fonction locale récursive `f` de même définition en substituant `f` à `foo` :

```
(defun foo (x)
   (labels ((f (y) (if ... (f ...))))
       (f x)))
```

(2) On duplique alors le code de `f` dans le `labels`, sous les noms `f1` et `f2` : les deux fonctions sont identiques, au nom près, donc on peut remplacer dans `f1` un appel de `f1` par `f2` sans rien changer. (3) On réitère tout le processus, pour définir des fonctions locales dans des `labels` imbriqués : le code d'une fonction intérieure peut appeler toutes les fonctions des `labels` englobant.

Ces variations n'ont bien entendu aucun intérêt pratique mais ce sont de bons tests pour vérifier la correction d'un évaluateur ou d'un compilateur.

3.2.2 Fonctions d'arité variable

Les fonctions arithmétiques courantes (+,*,<) sont souvent d'arité variable et on a vu que `&rest` permet de définir de telles fonctions (Section 2.3.2). Cela introduit un nouveau type de récursion, qui se fait sur le nombre de paramètres.

Prenons par exemple la fonction `append` qui concatène un nombre quelconque de listes. Il n'est pas facile d'écrire proprement cette fonction, car elle présente un double schéma de récursion, sur le premier argument, un peu comme `copylist`, et sur les autres arguments d'une façon assez nouvelle. Essayons :

```
(defun append (l &rest ll)    ; l est la première liste
   (if (null ll)              ; ll est la liste des autres listes
       l
     (if (atom l)
         (append ??? ll)      ; comment appeler append
       (cons (car l)          ;  sur les autres arguments ?
             (append (cdr l) ??? ll)))))
```

Manifestement, là où il y a des « ??? », on ne sait pas passer à `append` les arguments dont on ne dispose pas individuellement car ils ne sont connus que globalement pas la liste `ll`.

Une solution passe par la fonction `apply` qui a le double rôle de permettre d'appliquer une fonction que l'on ne connaît pas ou d'appliquer une fonction à des arguments que l'on ne connaît pas (cf. Section 3.4.2).

```
(defun append (l &rest ll)
   (if (null ll)
       l
     (if (null l)
         (apply #'append ll)
       (cons (car l)
             (apply #'append (cdr l) ll)))))
```

Le dernier argument de `apply` est la *liste des arguments restants*. Mais cette façon de faire est un peu contournée et `apply` coûte cher (voir par exemple, dans le polycopié de compilation [DL13], la section 4.5.2).

Il est beaucoup plus simple et efficace de passer par une fonction locale qui assure la récursion :

```
(defun append (l &rest ll)
    (labels ((app-r (l ll)
                    (if (null ll)          ; test d'arrêt sur ll
                        l
                      (if (atom l)         ; test d'arrêt sur l
                          (app-r (car ll) (cdr ll))
                        (cons (car l)
                              (app-r (cdr l) ll))))))
        (app-r l ll)))
```

Remarque 3.10. Il y a deux récursions, une sur la listes des paramètres — le test d'arrêt peut être `null` car la liste d'arguments est forcément propre (Remarque **??**) — et une sur le premier argument — le test doit être `atom` parce que l'argument n'est pas forcément une liste propre. Bien qu'aucun arbre ne soit explicite[2], cette récursion n'est pas sans ressembler aux récursions d'arbres premier-fils/frère, voir paragraphe 3.1.4, page 32.

EXERCICE 3.30. Définir la fonction `list` qui prend des arguments en nombre quelconque et en retourne leur liste propre. □

EXERCICE 3.31. Définir la fonction `list*` qui prend $n \geq 2$ arguments et retourne la liste des $n-1$ premiers avec le dernier dans le `cdr` de la dernière cellule. Avec 2 arguments, c'est équivalent à `cons`. Quelle serait la bonne définition de la fonction avec un seul argument ? □

2. En fait, l'arbre est constitué par la liste des listes passées en arguments.

Pourquoi le couple `apply` et &rest coûte-t-il cher ? Le principe d'`apply` consiste à prendre une liste d'arguments et à les pousser séparément dans la pile d'exécution. Inversement, `&rest` va construire une liste à partir des arguments qui ont été poussés individuellement dans la pile. Une récursion basée sur `apply` va donc passer son temps à vider une liste dans la pile du côté de l'appelant, pour en reconstruire une autre du côté de l'appelé.

EXERCICE 3.32. Comparer les deux versions de `append`, sur des listes de grandes tailles créées par `makelist` pour vérifier le surcoût de `apply`. Utiliser la macro `time` pour mesurer la consommation mémoire. En déduire une formule de complexité. □

3.3 Listes : copie, partage et chirurgie

Jusqu'ici la seule façon de construire des listes ou des arbres consiste à appliquer la fonction `cons` à deux arguments qui sont des expressions quelconques, donc potentiellement des listes.

3.3.1 Copie de listes

Les fonctions qui sont écrites de cette manière font des *copies* des listes passées en argument, comme par exemple `copylist`, `copytree` et `remove` (Exercice 3.1).

3.3.2 Partage de listes

La copie n'est pas très économique. Par exemple, `(remove 5 '(1 2 3))` est exactement équivalent à `(copylist '(1 2 3))`. Une première amélioration consiste à faire des fonctions qui partagent au mieux les listes, c'est-à-dire qui réutilisent les cellules des données *sans les détruire*. La définition de `remove` deviendrait ainsi la suivante :

```
(defun remove-p (x ll)
    (if (atom ll)                  ; test d'arrêt
        ll
      (if (eql x (car ll))         ; test d'égalité
          (remove-p x (cdr ll))    ; appel récursif terminal
        (let ((rest (remove-p x (cdr ll)))) ; appel réc. enveloppé
           (if (eq rest (cdr ll))
               ll
             (cons (car ll) rest))))))
```

Le principe du partage consiste à faire l'appel récursif sur le `cdr` (et sur le `car` en cas de récursion d'arbre), à comparer le résultat de l'appel récursif avec le contenu courant du `cdr`, et à retourner la cellule courante ou à en construire une nouvelle suivant le cas.

Remarque 3.11. On constate qu'il y a bien 2 appels récursifs mais que la récursion est quand même simple, car les 2 appels ne sont jamais activés simultanément. En revanche, la variable locale `rest` est absolument nécessaire : si l'on supprimait le `let`, en remplaçant dans son corps `rest` par l'appel récursif, cela ferait une double évaluation et une double récursion. Comme le calcul est 'fonctionnel', le résultat resterait inchangé mais la complexité passerait de linéaire à exponentielle.

EXERCICE 3.33. Définir la fonction `subst-p` qui fait l'équivalent de `subst` en version partage (Exercice 3.17). □

3.3.3 Chirurgie de liste

La manipulation de listes peut enfin emprunter une voie destructrice dans laquelle la liste en donnée est complètement détruite pour produire la liste résultat. Il faut bien entendu s'assurer du fait que la donnée ne sert plus à rien.

La définition de `remove` deviendrait ainsi la suivante :

```
(defun remove-d (x ll)
    (if (atom ll)
        ll
      (if (eql x (car ll))
          (remove-d x (cdr ll))
        (progn (setf (cdr ll) (remove-d x (cdr ll)))
               ll))))
```

Le principe des *fonctions destructrices* sur les listes plates est de réutiliser la cellule courante au lieu d'en allouer une neuve par un appel à `cons`. Ainsi, un appel à `(cons (car ll) (foo (cdr ll)))` sera simplement remplacé par `(progn (setf (cdr ll) (foo (cdr ll))) ll)`. Si le `car` est aussi concerné, en cas de récursion d'arbre ou de construction d'une liste basé sur le contenu du `car`, on transformera `(cons (bar (car ll)) (foo (cdr ll)))` en `(progn (setf (car ll) (bar (car ll)) (cdr ll) (foo (cdr ll))) ll)`.

On remarque que le principe de `remove-d` consiste à refaire le chaînage des `cdr` même si c'est inutile car il n'y a pas eu de changement : c'est en fait moins coûteux — à écrire et à exécuter — que de tester à chaque pas s'il y a eu un changement ou pas.

Remarque 3.12. La chirurgie de liste est un effet de bord : il est donc normal de faire une évaluation en séquence et de rencontrer un `progn`, ici explicite puisque c'est dans un `if`.

Remarque 3.13. Bien entendu, pour insérer des éléments nouveaux, on ne peut pas réutiliser des cellules anciennes : une fonction destructrice peut donc aussi contenir des appels à `cons`.

Remarque 3.14. Les fonctions destructrices en général, et la fonction `remove-d` en particulier[3], peuvent créer des situations surprenantes. Ainsi, `(remove-d 1 '(1 2 3))` → `(2 3)`. Mais que va retourner l'expression `(let ((test '(1 2 3))) (remove-d 1 test) test)` ?

EXERCICE 3.34. Que faut-il faire pour 'corriger' ce comportement contre-intuitif ? □

EXERCICE 3.35. Définir la fonction `subst-d` qui fait une substitution (cf. Exercice 3.17) en version destructrice. □

EXERCICE 3.36. Définir la fonction `append-d` qui concatène physiquement des listes. □

EXERCICE 3.37. Définir la fonction `reverse-d` qui inverse physiquement une liste, en réutilisant les mêmes cellules.
Que va retourner l'expression `(let ((ll '(1 2 3 4))) (reverse-d ll) ll)` ? □

Remarque 3.15. La convention usuelle de nommage des fonctions LISP veut que la version destructrice d'une fonction `foo` s'appelle `nfoo`. Mais seules `nsubst` et `nreverse` suivent cette convention. Les vrais noms de `remove-d` et `append-d` sont `delete` et `nconc` !

3.3.4 Listes circulaires

La chirurgie permet de construire des structures circulaires. C'est aussi ludique que dangereux car les fonctions sur les listes peuvent plonger dans une boucle infinie. De plus certains environnements, en particulier CLISP, ne savent pas les imprimer par défaut et ça peut les mettre dans un drôle d'état.

EXERCICE 3.38. Définir la fonction `cirlist` qui construit une liste circulaire de son unique argument. Tester `(cirlist 1)`, `(length (cirlist 1))` et `(equal (cirlist 1) (cirlist 1))`. □

Remarque 3.16. La fonction `print` de COMMON LISP est paramétrable par un grand nombre de variables globales. Faire `*print-` suivi de « → » pour avoir la liste de ces variables. En particulier, `*print-length*` et `*print-level*` permettent de limiter la longueur et la profondeur maximales d'impression : lorsque le seuil est atteint, l'impression se termine par « . . . ». Par défaut, en CLISP, ces paramètres sont initialisés à `nil`, ce qui enlève toute limitation. Avant de jouer avec des structures circulaires, il est impératif de donner à ces variables une valeur entière positive, par exemple :

3. Qui s'appelle `delete` dans la bibliothèque COMMON LISP.

```
(setf *print-length* 1000 *print-level* 100)
```

3.3.5 Principe de transformation de copie en chirurgie et en partage

De façon générale, une fonction sur les listes plates en copie contient le schéma suivant, éventuellement compliqué par des tests additionnels :

```
(defun foo (l)           ; les autres paramètres sont omis
    (if (<arret> l)      ; en général atom
        <init>           ; en général nil ou l
      (cons (<truc> l)   ; en général qqchose tiré de (car l)
            (foo (cdr l)))))
```

La transformation en mode chirurgie se fait automatiquement comme suit :

```
(defun foo-d (l)
    (if (<arret> l)
        <init>
      (progn (setf (car l) (<truc> l))        ; sauf si <truc> = car
             (setf (cdr l) (foo-d (cdr l)))   ; toujours
             l)))          ; la fonction retourne la cellule courante
```

S'il s'agit d'une fonction sur les arbres, le principe est en gros le même : `(<truc> l)` doit alors être remplacé par `(foo (car l))`.

Le partage n'est possible que si `<truc>` égale `car` et la transformation s'obtient ainsi :

```
(defun foo-p (l)
    (if (<arret> l)
        <init>
      (let ((rest (foo-p (cdr l))))
         (if (eql rest (cdr l))
             l            ; la fonction retourne la cellule courante
           (cons (car l) rest)))))
```

Dans le cas d'une récursion sur les arbres, il faut aussi mémoriser et tester l'appel récursif sur le `car`.

EXERCICE 3.39. Définir la fonction `listcar` qui retourne la liste des `car` d'une liste plate dont tous les éléments sont des listes : `(listcar '((1) (2) (3)))` retourne `(1 2 3)`. Faire les versions copie et chirurgie. Peut-on faire une version partage ? □

3.4 Fonctions d'ordre supérieur et fermetures

3.4.1 Valeurs fonctionnelles

Une valeur fonctionnelle s'obtient à partir d'un nom de fonction ou d'une λ-fonction en leur appliquant la forme syntaxique `function`. Par exemple `(function car)` ou `(function (lambda (x) x))`. Le résultat est une valeur du type `function`, qui vérifie le prédicat `functionp`. On peut appliquer `function` à toute "vraie" fonction, qu'elle soit globale (`defun`), locale (`labels` et `flet`) ou λ-fonction. En revanche, il est impossible d'obtenir la valeur fonctionnelle d'une "fausse" fonction, forme spéciale ou macro. Si l'argument de `function` n'a pas de valeur fonctionnelle associée, on obtient bien entendu une erreur.

La forme syntaxique `function` s'abrévie en « `#'` », comme `quote` s'abrévie en « `'` » — dans les deux cas, c'est la fonction `read` qui effectue la transformation.

```
(function car)                → #<SYSTEM-FUNCTION CAR>
#'car                         → #<SYSTEM-FUNCTION CAR>
(function (lambda (x) x))     → #<FUNCTION :LAMBDA (X) X>
#'(lambda (x) x)              → #<FUNCTION :LAMBDA (X) X>
'(lambda (x) x)               → (LAMBDA (X) X)
'car                          → CAR
(functionp '(lambda (x) x))   → NIL
(functionp 'car)              → NIL
(functionp #'(lambda (x) x)) → T
(functionp #'car)             → T
```

Il ne faut donc pas confondre la valeur fonctionnelle avec son expression syntaxique.

Remarque 3.17. La syntaxe « `#<...>` » représente en LISP les valeurs qui ne peuvent pas être lues : toute expression LISP est une valeur mais il est des valeurs qui ne sont pas des expressions LISP.

3.4.2 Fonctions prédéfinies d'ordre supérieur

Le langage prédéfinit un ensemble de fonctions d'ordre supérieur — donc qui prennent une fonction en paramètre. On distingue 3 groupes de fonctions.

Appliquer une fonction à des arguments

Une valeur fonctionnelle n'a de sens que si on peut l'*appliquer*. La primitive d'application d'une fonction est `apply` : c'est une vraie fonction donc tous ses arguments sont des expressions évaluables. Toutes les autres fonctions d'ordre supérieur utilisent `apply`. La syntaxe de `apply` est un peu particulière.

```
(apply <fonction> <arg-individuel>* <liste-arg-restant>)
```

Le premier argument est la valeur fonctionnelle à appliquer. Les arguments suivants sont les arguments à passer à cette valeur fonctionnelle, mais il faut distinguer les premiers arguments, qui sont des arguments individuels, et le dernier qui est "la liste des arguments restants". La fonction `apply` a donc le double rôle de permettre d'appliquer une fonction inconnue à des arguments connus, ou une fonction connue à des arguments inconnus individuellement.

Lorsque tous les arguments sont connus, on utilise `apply` avec un dernier argument `nil` — ou de façon équivalente la fonction `funcall`.

Exemple Toutes les expressions suivantes sont équivalentes :

```
(apply #'+ 1 2 3 4 ())
(apply #'+ 1 2 3 '(4))
(apply #'+ 1 2 '(3 4))
(apply #'+ 1 '(2 3 4))
(apply #'+ '(1 2 3 4))
```

Et elles sont aussi équivalentes à la définition

```
(defun foo (fun &rest args)
    (apply fun args))
```

utilisée avec l'appel `(foo #'+ 1 2 3 4)`, ou encore avec la fonction

```
(defun bar (fun &rest args)
    (apply #'apply fun args))
```

utilisée avec l'appel `(bar #'+ 1 2 '(3 4))`.

Remarque 3.18. Le principe de `apply` et `&rest` (Section 3.2.2) repose sur '*la liste des arguments restants*' mais les deux mécanismes sont néanmoins à peu près inverses l'un de l'autre. `&rest` préfixe le dernier *paramètre* d'une fonction de telle sorte que la liste des arguments restants (c'est-à-dire non liés aux paramètres obligatoires qui précèdent) soit liée à ce paramètre unique. En sens inverse, avec `apply`,

le dernier *argument* de l'appel à `apply` a pour valeur la liste des arguments restants qui seront liés aux paramètres correspondants de la fonction appelée. En pratique, `apply` oblige à décoder la liste pour mettre ses éléments un par un dans la pile d'exécution, alors que `&rest` oblige à reconstruire une liste à partir des derniers arguments qui sont dans la pile. La version de `append` qui utilise `apply` (Section 3.2.2) est donc passablement inefficace. Il n'y a bien entendu aucune nécessité à ce que le paramètre préfixé par `&rest` soit lié à la valeur du dernier argument de `apply`, cela peut arriver mais ce n'est qu'une coïncidence.

Fonctions de mapping et de séquences

Les fonctions dites de *mapping* permettent d'appliquer une fonction aux éléments successifs d'une ou plusieurs listes, autant de listes que la fonction argument a de paramètres. Les listes sont donc parcourues en parallèle, et la fonction argument est appliquée aux éléments courants. La syntaxe est la suivante :

```
(<mapping> <fonction> <expr-list>+)
```

Chaque `<expr-list>` est une expression évaluable dont la valeur doit être une liste et `<fonction>` est une expression évaluable dont la valeur doit être une fonction dont l'arité correspond au nombre de listes. Le *mapping* s'arrête à la terminaison de la liste la plus courte.

Ces fonctions s'appellent `map`, `mapcar`, `mapcan`, etc. Elles diffèrent par ce qu'elles retournent et par le fait que l'élément courant peut être la cellule courante ou son `car`.

Les fonctions de séquences (Section 2.5.5) appliquent une fonction aux éléments successifs d'une séquence : par exemple, `reduce`, `some`, etc. Quelques fonctions s'appliquent exclusivement aux listes, comme `member-if`.

EXERCICE 3.40. Définir la fonction `reduce` qui cumule l'application d'une fonction binaire sur les éléments d'une liste. Par exemple, `(reduce #'+ '(1 2 3) :initial-value 1)` retourne `7`. □

EXERCICE 3.41. Définir la fonction `member-if` qui est similaire à `member` mais passe en premier argument un prédicat à vérifier et non un élément à rechercher : elle retourne la sous-liste commençant par le premier élément qui vérifie le prédicat argument. □

EXERCICE 3.42. Quelle est la différence entre `member-if` et `some` ? □

Argument fonctionnel optionnel

En plus de ces différentes fonctions, plusieurs fonctions de la bibliothèque standard ont des arguments optionnels (`&optional` ou `&key`) dont la valeur doit être fonctionnelle. Dans les fonctions de recherche comme `member` ou `assoc`, il est possible de préciser :

— avec le mot-clé `:test`, la fonction d'égalité à utiliser (par défaut c'est `eql`) ;
— avec le mot-clé `:key`, la fonction à utiliser pour accéder à la valeur à comparer : au lieu de chercher un élément égal à l'argument, on peut chercher un élément dont le `car` est égal à l'argument.

EXERCICE 3.43. Définir la fonction `member` qui implémente les deux mots-clés `:test` et `:key`. □

EXERCICE 3.44. Définir la fonction `assoc` qui recherche une clé dans une *liste d'association*, c'est-à-dire une liste de paires clés-valeurs : en cas de succès, `assoc` retourne la paire. Faire une première version simple et une seconde qui implémente les deux mots-clés `:test` et `:key`. □

3.4.3 Valeurs fonctionnelles et fermetures

La notion de fermeture repose sur la *capture* d'un environnement par une expression syntaxique, en l'occurrence une λ-fonction.

Variables libres, variables liées

Une variable — c'est-à-dire une occurrence d'un symbole dans du code LISP — est dite *liée* si elle est incluse dans une définition de fonction (`defun`, `lambda`, `labels`, `let`) qui l'*introduit* comme paramètre. Dans le cas contraire, la variable est dite *libre*.

La notion de *variable libre* ou *liée* peut aussi être relative à une expression : une (occurrence de) variable est libre dans une expression `expr` si l'expression qui introduit la variable n'est pas incluse dans `expr`.

Une *fermeture* est alors une définition de fonction qui *capture* l'environnement de ses *variables libres*, ce qui aboutit à lier toutes ses variables. Il est important de noter que ce sont les liaisons (ou les environnements) qui sont capturées et non pas les valeurs : la fermeture peut modifier ces liaisons (par une affectation), en affectant par la même occasion toutes les fermetures qui se partagent cet environnement.

Exemple

On utilise des fermetures lorsque l'on souhaite permettre à plusieurs fonctions de partager des variables communes. Un générateur de nombres (ou compteur) peut ainsi s'implémenter avec 3 fonctions pour consulter le compteur, l'incrémenter ou le remettre à 0.

```
(let ((n 0))
   (defun counter () n)
   (defun counter++ () (setf n (1+ n)))
   (defun counter-reset () (setf n 0)))
```

Le `let` crée un environnement que les 3 fonctions se partagent.

EXERCICE 3.45. Vérifier le comportement du compteur. □

EXERCICE 3.46. Définir la fonction `next-fibo` qui calcule la valeur de Fibonacci suivante, et `reset-fibo` qui la réinitialise :

```
(next-fibo)  → 0  0
(next-fibo)  → 1  1
(next-fibo)  → 2  1
(next-fibo)  → 3  2
(next-fibo)  → 4  3
(next-fibo)  → 5  5
(reset-fibo) → 0  0
(next-fibo)  → 1  1
```

Bien entendu, la complexité de la fonction doit rester constante (en $\mathcal{O}(1)$). □

EXERCICE 3.47. Définir la fonction `next-prime` qui calcule le nombre premier suivant, et `reset-prime` qui la réinitialise. On se rappellera que la définition la plus opérationnelle d'un nombre premier est celle d'un nombre qui n'est pas divisible par les nombres premiers précédents. On pourra raffiner cette définition en ne testant que les nombres premiers dont le carré est inférieur au nombre testé, et en n'effectuant pas ce nouveau test pour les nombres premiers qui ont déjà été testés comme diviseur. Pour des raisons d'efficacité, l'implémentation de *file* décrite par ailleurs (Section 6.6.1) sera très utile. □

Portée et extension

On a parlé de *portée lexicale* [4] pour qualifier le mécanisme de liaison des variables. La portée concerne l'espace du programme, et la portée lexicale se caractérise par sa *localité*. Il est important aussi de considérer le temps : on parle alors d'*extension* [5]. L'existence des fermetures fait que l'extension des variables est illimitée. Alors que l'on aurait pu croire, dans l'exemple du compteur, que l'environnement contenant `n` disparaîtrait en sortant du `let`, sa capture par une fonction globale le fait durer 'éternellement'.

4. En anglais *lexical scope*.
5. En anglais *extent*.

λ-fonction

Une λ-fonction est toujours une fermeture : c'est pour cela que les λ-expressions s'évaluent en passant l'environnement d'appel, qui n'est autre que l'environnement de définition (Section 2.3.2).

Les λ-fonctions sont aussi le moyen de définir des *valeurs fonctionnelles* qui capturent leur environnement de définition et peuvent être passées en argument à d'autres fonctions. C'est une façon très puissante de programmer des mécanismes asynchrones comme les *callbacks* mais on peut aussi s'en servir plus simplement avec les fonctions d'ordre supérieur. Ainsi, dans un environnement ou les variables `y` et `ll` sont liées, par exemple $\{Y \rightarrow 3, LL \rightarrow ...\}$, les deux expressions suivantes sont équivalentes :

```
(member y ll :test #'= :key #'car)
(member-if #'(lambda (x) (= (car x) y)) ll)
```

EXERCICE 3.48. Vérifier cette affirmation d'équivalence en consultant les spécifications exactes de ces fonctions dans le manuel et en faisant un test. □

EXERCICE 3.49. Quelles sont les différences entre les fonctions `assoc` et

```
(lambda (x ll) (member-if #'(lambda (y) (eql (car y) x)) ll))
(lambda (x ll) (some #'(lambda (y) (eql (car y) x)) ll))
(lambda (x ll) (member x ll :key #'car)) ?
```

□

Capture d'environnement

Dans l'expression `member-if` ci-dessus (voir aussi Exercice 3.41), la λ-fonction a capturé la liaison de sa variable libre `y`.

La 'capture' se comprend ainsi : l'évaluation de `#'(lambda (x) (= (car x) y))` retourne un objet 'fermeture', de type `function`, qui contient 2 éléments distincts : la λ-fonction `(lambda (x) (= (car x) y))` et l'environnement capturé $\{Y \rightarrow 3, LL \rightarrow ...\}$. Cet objet 'fermeture' s'imprime par « `#<FUNCTION :LAMBDA (X) (= (CAR X) Y)>` » (voir Remarque 3.17). Lorsque la fermeture sera appliquée à une valeur `(4 . 5)`, le corps de la λ-fonction sera évalué dans l'environnement $\{X \rightarrow (4\ .\ 5), Y \rightarrow 3, LL \rightarrow ...\}$. Il est important de noter que chaque évaluation de `#'(lambda (x) (= (car x) y))` va produire un nouvel objet 'fermeture', de même que chaque application d'une fonction produit un nouvel environnement.

Le cas du compteur est similaire : le `let` crée un environnement $\{N \rightarrow 0\}$ à l'intérieur duquel chaque `defun` associe au nom de la fonction, par exemple `counter`, la fermeture `#'(lambda () n)`, qui englobe donc la λ-fonction `(lambda () n)` et l'environnement $\{N \rightarrow 0\}$. Les 3 fonctions se partagent donc le même environnement et l'application de chaque fonction se fait dans cet unique environnement, puisqu'aucune n'a de paramètre. Si l'on avait fait 3 `let`, un par fonction, il y aurait 3 environnements $\{N \rightarrow 0\}$ physiquement différents, non partagés, et le compteur ne marcherait pas. De la même manière, si l'on refait maintenant la même définition en changeant légèrement le nom des fonctions, par exemple en le suffixant par « `2` », chaque triplet a son propre environnement et l'action de `counter2++` n'a aucun effet sur `counter`, de même que `counter++` n'a aucun effet sur `counter2`.

3.4.4 Terminalisation des récursions enveloppées par continuation

Une *continuation* est une valeur fonctionnelle passée en paramètre à une fonction qui va s'en servir pour *continuer* son calcul. L'un de ses emplois les plus saisissants est la terminalisation des récursions enveloppées, même lorsqu'elles sont multiples.

Reprenons le schéma des récursions simples enveloppées :

```
(defun foo-e (ll)
    (if (<arret> ll)                          ; test d'arrêt
        <init>                                ; valeur initiale
      (<enveloppe>                            ; enveloppe
        (foo-e (<suivant> ll)))))             ; appel récursif
```

L'idée des continuations consiste à transformer `foo-e` en une version terminale dont le paramètre supplémentaire est une fonction qui va être appelée à la fin du calcul, c'est-à-dire au moment de l'arrêt de la récursion :

```
(defun foo-tc (ll c)
    (if (<arret> ll)
        (apply c <init> ())          ; application de la continuation
     (foo-tc (<suivant> ll)
             #'(lambda (x)           ; nouvelle continuation
                       (apply c (<enveloppe> x) ())))))
```

La fonction `foo-tc` doit alors être appelée avec une première continuation qui doit retourner `<init>` lorsqu'on l'applique à `<init>` : la fonction identité `#'(lambda (x) x)` va marcher dans tous les cas. Bien entendu, comme pour le schéma de terminalisation vu en Section 3.1.3, on fera une fonction récursive locale.

EXERCICE 3.50. Appliquer ce schéma à `length` et à `fact`. □

EXERCICE 3.51. Développer à la main l'appel de `(length-tc '(1 2 3))` en explicitant complètement les fermetures et les environnements capturés. □

EXERCICE 3.52. Appliquer ce schéma à `copylist` et `reverse`. □

Remarque 3.19. Lorsque l'enveloppe n'est pas associative et commutative, il est important de distinguer `(apply c (<enveloppe> x) ())` et `(<enveloppe> (apply c x ()))`. Mais on remarque qu'il est toujours possible d'obtenir le résultat souhaité. Contrairement au schéma de terminalisation présenté en Section 3.1.3, celui-ci est complet : on peut obtenir à la fois `copylist` et `reverse`.

Remarque 3.20. Le mot *continuation* est utilisé ici dans un sens un peu différent de son usage dans la construction `call/cc` de SCHEME (Section 2.9).

3.4.5 Application aux récursions doubles

Ce schéma s'applique aussi très bien aux récursions doubles : il suffit de l'appliquer 2 fois de suite, mais cela nécessite un peu de doigté. On va donc décomposer toutes les étapes.

On commence par définir une première version avec une fonction locale récursive, `foo` :

```
(defun size-tc0 (tr)
  (labels ((foo (tr)
                (if (atom tr)
                    0
                  (+ 1 (foo (cdr tr)) (foo (car tr))))))
          (foo tr)))
```

On note que l'on peut mélanger des appels récursifs locaux (à `foo`) et globaux (à `size-tc`), puisque les deux fonctions sont rigoureusement équivalentes :

```
(defun size-tc1 (tr)
  (labels ((foo (tr)
                (if (atom tr)
                    0
                  (+ 1 (foo (cdr tr)) (size-tc1 (car tr))))))
          (foo tr)))
```

On abstrait alors l'enveloppe de l'appel récursif à `foo` :

```
(defun size-tc2 (tr)
  (labels ((foo (tr)
                (if (atom tr)
                    0
                  ((lambda (x) (+ 1 x (size-tc2 (car tr))))  ; enveloppe
                   (foo (cdr tr))))))
          (foo tr)))
```

On applique ensuite le schéma de terminalisation à `foo` :

```
(defun size-tc3 (tr)
  (labels ((foo (tr c)
                (if (atom tr)
                    (apply c 0 ())
                  (foo (cdr tr)
                       #'(lambda (x) (apply c (+ 1 x (size-tc3 (car tr))) ()))))))
          (foo tr #'(lambda (x) x))))
```

On abstrait alors l'enveloppe de l'appel récursif à `size-tc` :

```
(defun size-tc4 (tr)
  (labels ((foo (tr c)
                (if (atom tr)
                    (apply c 0 ())
                  (foo (cdr tr)
                       #'(lambda (x)
                           ((lambda (y) (apply c (+ 1 x y) ()))
                            (size-tc4 (car tr))))))))
          (foo tr #'(lambda (x) x))))
```

Et l'on finit en appliquant le schéma de terminalisation à la deuxième récursion, en renommant `size-tc` en `foo`.

```
(defun size-tc5 (tr)
  (labels ((foo (tr c)
                (if (atom tr)
                    (apply c 0 ())
                  (foo (cdr tr)
                       #'(lambda (x)
                           (foo (car tr)
                                #'(lambda (y) (apply c (+ 1 x y) ()))))))))
          (foo tr #'(lambda (x) x))))
```

Bien entendu, toutes ces transformations sont automatisables et un compilateur s'en sortirait beaucoup mieux qu'un programmeur humain ! C'est une optimisation : il faut la garder pour la fin et ne pas l'appliquer avant d'avoir parfaitement débuggué la version enveloppée. [SJ93] développe longuement ces transformations.

EXERCICE 3.53. Appliquer ce schéma à `copytree`, `tree-leaves` et `fibonacci`. □

EXERCICE 3.54. Définir la fonction `terminalise` qui transforme une définition de fonction récursive enveloppée en une définition de fonction récursive terminale sur la base de cette transformation. □

Remarque 3.21. Cette transformation est vraiment magique : elle permet de transformer une fonction qui consomme une ressource (la pile) en une fonction qui n'en consomme apparemment pas. Mais en fait le résultat ne consomme-t-il vraiment pas de pile ? Avec CLISP ce n'est en réalité pas vrai : il suffit de le tester avec la macro `time`. Et on verra, en implémentant les fermetures dans le méta-évaluateur, que la fermeture, c'est-à-dire la capture d'un environnement, consomme en réalité du *tas* (*heap*).

EXERCICE 3.55. Se servir de la macro `time` pour mesurer la quantité de mémoire consommée par ces fonctions, en version interprétée et compilée. □

EXERCICE 3.56. Comparer le comportement des diverses versions de `fact` (enveloppée ou terminale suivant les 2 schémas de terminalisation), et le tout en version compilée ou interprétée. Utiliser `time` pour mesurer l'efficacité en espace et en temps. Déterminer à partir de quelle valeur d'argument la récursion casse la pile. Pour s'abstraire du bruit (en fait un vrai vacarme) provoqué par l'arithmétique sur les `bignum`, transformer la fonction pour qu'elle fasse `+ 1` au lieu de `* n`. □

3.5 Les macros

Une macro est une fonction à usage syntaxique, qui prend en paramètre de la syntaxe LISP et retourne de la syntaxe LISP, ou plus exactement qui prend en entrée une expression évaluable et qui retourne une expression évaluable. On appelle cela des transformations source-à-source : tous les compilateurs en sont friands car cela évite des redondances assez lourdes. Par exemple on pourrait définir `cond` comme une macro fonction de `if`, ou l'inverse, et `let` et `let*` comme des macros fonctions de `lambda` [6].

Les macros sont définies par la forme syntaxique `defmacro`, qui a la même syntaxe que `defun`.

3.5.1 Principe et double évaluation

Le principe d'évaluation des macros repose sur la double évaluation. Lorsque l'évaluateur repère, dans l'expression à évaluer, que le symbole en 'position de fonction' est une macro,

— il n'évalue pas les arguments,
— il applique la définition associée à la macro aux arguments non évalués comme pour l'application d'une vraie fonction ;
— cette application retourne une expression LISP qui est censée être évaluable : cette première évaluation est appelée l'*expansion* ;
— cette nouvelle expression est à son tour évaluée : c'est la *deuxième évaluation*.

Le point clé pour l'efficacité est que le résultat de l'expansion remplace physiquement l'expression d'origine, ce qui fait que l'expansion ne se fait jamais qu'une seule fois.

Remarque 3.22. La double évaluation représente le principe abstrait de l'évaluation des macros. En pratique, il n'est souvent pas possible de déterminer à quel moment a lieu la première évaluation (l'expansion). Si le code est compilé, c'est à la compilation et l'expansion de macro ne représente alors que l'une des nombreuses transformations source-à-source utilisées par le compilateur. Si le code est interprété, l'expansion peut avoir lieu aussi bien à la définition des fonctions (`defun`). Comme l'expansion des macros ne dépend pas de l'environnement d'exécution, il est difficile de voir la différence.

Exemple : la 'macro' `let`

Bien que ce soit en fait une forme syntaxique, on pourrait définir `let` comme une macro de la façon suivante (on se restreint à une seule variable pour simplifier) :

```
(defmacro let (lvar-val &rest body)
    (list (list* 'lambda
                 (list (caar lvar-val))
                 body)
          (cadar lvar-val)))
```

Si l'on applique cette transformation à la fonction `g` ci-dessous

```
(defun g (v) (* 5 (let ((x  (+ v 2))) (+ v x))))
```

l'expansion du `let` produit le code de la section 2.3.2 :

```
(defun g (v) (* 5 ((lambda (x) (+ v x)) (+ v 2))))
```

Deux fonctions sont très utiles pour débugguer les macros, `macroexpand` et `macroexpand-1` : la seconde fait un coup de macro-expansion sur son argument alors que la première fait une macro-expansion complète. Il est conseillé de toujours tester ses macros avec `macroexpand-1` avant de chercher à évaluer le code qui les utilise.

```
(macroexpand-1 '(let ((x  (+ v 2))) (+ v x)))
                        → ((lambda (x) (+ v x)) (+ v 2))
```

EXERCICE 3.57. Généraliser `let` à un nombre quelconque de variables. □

6. Pourtant, en COMMON LISP, tous ces exemples sont des formes syntaxiques.

3.5.2 Backquote

La `backquote` est une abréviation syntaxique qui permet de construire aisément des listes à partir d'un squelette constant dans lequel on insère le résultat d'évaluations. Ainsi, pour `let`, on écrirait :

```
(defmacro let (lvar-val &rest body)
    `((lambda (,(caar lvar-val))
              ,@body)
      ,(cadar lvar-val)))
```

Le squelette est formé de l'expression *backquotée* `((lambda - -) -)`, où les « - » désignent les places où doivent s'insérer le résultat d'évaluations. Les expressions à évaluer sont préfixées par « , » (virgule), et « ,@ » (virgule-arobase) indique que la liste à insérer doit perdre une paire de parenthèses.

La syntaxe d'une expression *backquotée* se définit formellement comme suit :

<backquote>	:=	« ` » <expr-backquote>
<expr-backquote>	:=	<atome> | <expr-virgule> | « (» <expr-backquote>+ [« . » <expr-backquote>] «) »
<expr-virgule>	:=	« , » <expr-eval> | « ,@ » <expr-eval>

Il faut enfin rajouter à la syntaxe de `<expr-eval>` le fait que ce peut être aussi une `<backquote>`.

Remarque 3.23. La *backquote* est le caractère « ` » (accent grave) à ne pas confondre avec la *quote* « ' » (accent aigu). C'est aussi une généralisation de la *quote*, dans la mesure où les deux retournent le même résultat lorsque le squelette ne contient aucune « , ».

Remarque 3.24. L'usage de la virgule est interdit en-dehors d'une expression *backquotée*.

EXERCICE 3.58. Écrire la version générale de `let` avec *backquote*. □

Tester backquote

Pour comprendre ce que fait *backquote*, il est intéressant d'analyser la valeur de `` `(toto ,titi truc ,@tata tutu) `` ou de toute autre expression *backquotée*.

EXERCICE 3.59. Regarder sous l'interprète la valeur de `` '`(toto ,titi truc ,@tata tutu) `` : comme `print` restitue la forme *backquotée* d'origine, on analysera la valeur en en prenant les `car` et `cdr` successifs. Si l'on ne trouve pas de cette manière l'expansion de *backquote*, on peut aussi appliquer `macroexpand` à l'expression *backquotée*. Expliquer. □

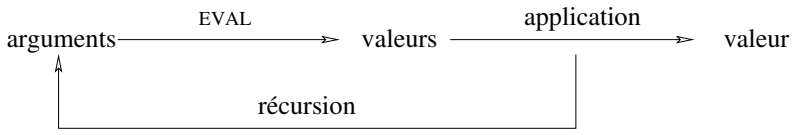
EXERCICE 3.60. Vérifier que les deux expressions `` `(toto ,@titi) `` et `` `(toto . ,titi) `` sont équivalentes. □

Implémenter backquote

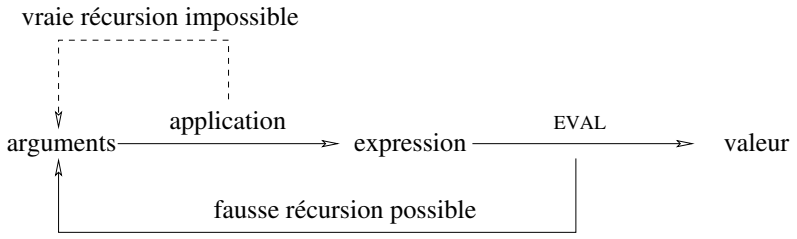
L'objectif est de définir le fonctionnement du caractère *backquote* (« ` ») en définissant une fonction `backquotify` qui transforme l'expression backquotée en une expression de construction de liste. Pour cela, il faut d'abord supposer que « ,x » est lu (fonction `read`) comme la liste `(unquote x)`, de la même manière que « 'x » est lu `(quote x)`. Idem pour « ,@x » qui est lu `(splice-unquote x)`.

EXERCICE 3.61. Définir `backquotify`. On procède par étapes

1. dans un premier temps, on définit `backquotify` pour qu'il transforme l'expression `` `(e1 e2 .. en) `` en `(list 'e1 ... 'en)` ;
2. on traite ensuite le cas où des `ei` sont de la forme `(unquote fi)` de telle sorte que l'expression produite comporte alors `fi` à la place de `'ei` ;
3. on généralise alors à un arbre : la récursion ne se fait pas seulement sur le `cdr` mais aussi sur le `car` ;
4. on traite ensuite le cas des `(splice-unquote x)` : attention, il faut alors faire une récursion de la cellule "du dessus" (rare contre-exemple à la règle consistant à ne tester dans une récursion que la cellule courante, cf Remarque 3.3) ;



(a) “Vraie” fonction



(b) Macro

FIGURE 3.1 – Commutation des évaluations et des applications entre “vraie” fonction et macro.

5. on finit par la définition d’une fonction de simplification qui fait l’inverse de l’étape 1 en reconstruisant des constantes et qui simplifie les appels imbriqués à `append`, `cons` et `list*` ou `list`, etc.

On constate que ce schéma effectue une analyse par cas qui reprend pour l’essentiel les divers cas de la syntaxe de `<backquote>`. On peut alors vérifier, que ça marche, ici en appliquant `backquotify` au corps de la définition précédente de `let` :

```
(backquotify '((lambda ((unquote (caar lvar-val)))
                       (splice-unquote body))
                 (unquote (cadar lvar-val))))
```

doit retourner quelque-chose comme :

```
(list (list* 'lambda (list (caar lvar-val)) body) (cadar lvar-val))
```

Comparer le résultat avec la version du système (Exercice 3.59). □

3.5.3 Les macros ‘récursives’

Comme toute fonction, une macro pourrait *a priori* être récursive. Cependant, contrairement aux vraies fonctions, une macro n’évalue pas ses arguments. Aussi, si la récursion conduit à réévaluer exactement la même expression, une boucle infinie est inévitable. Donc, en pratique, les cas de récursion sont excessivement rares et l’on considérera qu’*une macro n’est jamais récursive*.

Cependant, les macros mettent en œuvre une forme particulière de récursion qui se traduit par le fait que la macro figure dans le code résultant de l’expansion : on dira que *la macro s’expanse récursivement*. On l’utilisera en particulier, mais pas seulement, lorsque la macro est d’arité variable.

Commutation de l’évaluation et de l’application On peut analyser différemment le principe des macros en remarquant qu’il ne s’agit en fait que de faire commuter l’*évaluation* et l’*application*.

Pour une ‘vraie’ fonction, le principe de l’évaluation consiste à évaluer les arguments et à appliquer la fonction aux valeurs résultantes (Figure 3.1a). Pour une macro, on applique d’abord la ‘fonction’ associée à la macro et on évalue ensuite le résultat (Figure 3.1b). Dans tous les cas, l’évaluation d’une expression passe par exactement un appel récursif sur les sous-expressions qui sont à évaluer — le terme de ‘double évaluation’ est donc impropre.

On voit bien en quoi la récursion diffère entre fonctions et macros. Dans les fonctions, l’appel récursif peut se faire dans l’application, car l’appel s’applique aux mêmes arguments syntaxiques dont l’évaluation va produire de nouvelles valeurs. Avec une macro, l’appel récursif dans l’application n’évalue pas les arguments qui vont donc rester inchangés, d’où l’impossibilité. En revanche, si la macro apparaît dans l’expression résultant de l’application, les arguments syntaxiques ont pu changer.

Exercices Dans les exercices qui suivent, soit la macro est d'arité quelconque (`and`, `or`, `cond`), soit elle a un paramètre de longueur quelconque (`let*`) : la récursion s'effectue sur cet argument particulier ou sur la liste des arguments.

EXERCICE 3.62. Définir la macro `let*` qui implémente les spécifications exactes du `let` *séquentiel* et s'expanse par des `lambda` imbriqués (Section 2.4.4). □

EXERCICE 3.63. Définir la macro `and` qui implémente les spécifications de la conjonction logique : l'arité est quelconque et les arguments sont évalués en séquence jusqu'au premier qui retourne `nil` — et `and` retourne `nil` — ou jusqu'à la fin, en retournant alors la valeur du dernier argument. □

EXERCICE 3.64. Définir la macro `cond` qui implémente une cascade de conditionnelles suivant les spécifications de la Section 2.4.1. □

3.5.4 Comment définir une macro

La première chose à faire quand on veut définir une macro consiste à déterminer comment on pourrait la remplacer, *à la main* : tout ce que fait une macro peut être fait à la main par le programmeur. Ainsi, pour faire de `prog1` ou `prog2` une macro, il faut d'abord répondre à l'exercice 2.4.

Pour répondre à cette première question, il faut souvent commencer par faire une *analyse par cas*, pour déterminer les différentes configurations syntaxiques possibles de l'usage de la macro, et décider de l'expansion de chaque cas. Ainsi pour la macro `or` (la duale de `and`), il faut distinguer les cas suivant qu'il y a 0, 1 ou plus d'un argument. Le principe de l'expansion est alors le suivant :

```
                (or) → ()                                   élément neutre de l'opération
   (or <expr-eval>) → <expr-eval>
(or <e_1> .. <e_n>) → (if <e_1> <e_1> (or <e_2>.. <e_n>))   récursion
```

La définition de la macro va donc reposer sur une série de tests pour discriminer les cas (`cond`, `if` emboîtés, ou autre). Chaque cas conduit à une expansion particulière, en général par une expression *back-quotée*. Il n'est en général pas utile de chercher à factoriser le code entre plusieurs cas : cela contribue à obscurcir le code.

EXERCICE 3.65. Écrire le code correspondant à la transformation du `or`. □

3.5.5 Les pièges des macros

Les macros constituent un outil très puissant, mais il faut bien maîtriser leurs dangers dont l'origine vient du fait que ce sont des fonctions qui manipulent de la syntaxe.

Double évaluation

La macro `or` illustre parfaitement ces dangers. Si l'on examine le flot de contrôle résultant de l'expansion précédente, on constate que `<e_1>` est évalué deux fois si la première évaluation retourne *vrai*.

On a là un phénomène de *double évaluation* qui ne correspond pas au désir du programmeur — la spécification de `or` ne parle pas d'évaluer une sous-expression plusieurs fois — et qui peut avoir un effet notable sur le comportement du programme :

— si l'expression `<e_1>` n'est pas *fonctionnelle* mais fait des effets de bord, ces effets de bord seront effectués 2 fois : incrémenter deux fois une variable ou lire 2 caractères sur le canal d'entrée n'est pas neutre ;

— si l'expression `<e_1>` est fonctionnelle, l'effet peut être aussi dramatique : la suite de Fibonacci (Section 3.1.4) montre bien que c'est la répétition d'un calcul (même si approché) qui transforme une complexité linéaire en une complexité exponentielle.

EXERCICE 3.66. La définition suivante de factorielle est correcte :

```
(defun fact2 (n)
    (if (= n 0)
        1
      (/ (* n (+ (fact2 (- n 1)) (fact2 (- n 1)))) 2)))
```

Quelle est sa complexité ? □

Remarque 3.25. L'expression 'double évaluation' a un double sens dans le contexte des macros : c'est à la fois le principe de l'évaluation des macros et l'un de leurs chausse-trappes les plus amusants.

Capture de variable

Revenons à notre macro `or`. Lorsqu'un programmeur a besoin d'utiliser le résultat d'une évaluation à deux endroits différents, il lui faut passer par une *variable locale* pour en mémoriser la valeur ou, de façon équivalente, par une fonction. Par exemple la fonction `square` (Section 2.3.1). Dans le cas de `or`, on obtiendrait donc :

```
(defmacro or (&rest exprs)
    (cond ((null exprs)
           ())
          ((null (cdr exprs))
           (car exprs))
          (t
           `(let ((x ,(car exprs)))
               (if x x (or ,@(cdr exprs)))))))
```

Le squelette est maintenant `(let ((x -)) (if x x (or -)))`.

Hors contexte, cette nouvelle définition est parfaite. Mais tout se complique si on cherche à s'en servir :

```
(defun foo (x y)
    (if (or (null x) (> x y)) ..)
        ..
      ..))
```

L'évaluation de `(foo 2 3)` va entraîner l'erreur : `"> : nil n'est pas un nombre"` qui va interloquer le programmeur. L'expression `or` va en effet s'expanser par

`(let ((x (null x))) (if x x (or (> x y) ..))).`

A l'entrée de `foo`, l'environnement est $\{x \to 2, y \to 3\}$, donc `(null x)` s'évalue à `nil` et le corps du `let` est évalué dans l'environnement $\{x \to nil, y \to 3\}$. D'où l'erreur. Mais bien entendu, avant l'erreur, le second `or` résultant de la première expansion se sera lui-même expansé, ce qui donne au total :

```
(defun foo (x y)                          ; {x → 2, y → 3}
    (if (let ((x (null x)))               ; {x → nil, y → 3}
          (if x
              x
            (let ((x (> x y)))            ; erreur !
              (if x x (or ..)))))
        ..
      ..))
```

Bien sûr on peut chercher des symboles moins courant que `x`, mais le problème risque toujours de se reproduire. La bonne solution repose sur la génération automatique de symboles non ambigus, avec la fonction `gensym` (voir le manuel). Le code devient le suivant :

```
(defmacro or (&rest exprs)
    (cond ((null exprs)
           ())
          ((null (cdr exprs))
           (car exprs))
          (t
           (let ((var (gensym)))
              `(let ((,var ,(car exprs)))
                  (if ,var ,var (or ,@(cdr exprs)))))))))
```

Cet exemple de macro est déjà un peu plus compliqué et permet des commentaires plus approfondis sur la façon de définir une macro :

— la structure de `cond` fait bien apparaître l'*analyse par cas* discutée plus haut ;
— chaque cas, mais le dernier est le plus révélateur, se décompose en un certain *travail effectué par la macro* (l'appel à `gensym`) et l'*expansion proprement dite* (expression *backquotée*).

EXERCICE 3.67. Appeler la fonction `gensym` plusieurs fois et voir ce qu'elle retourne. Expliquer. □

Remarque 3.26. Il faut bien placer la « ` » (*backquote*) et il est impératif, pour cela, de bien distinguer ce que j'ai appelé le *travail de la macro* et son *expansion* : la remontée de la « ` » à l'expression `let` englobante aurait bien sûr un effet complètement néfaste.

Remarque 3.27. Il faut bien placer les « , » (virgule) et il y a pour cela quelques règles simples : une virgule ne peut pas apparaître hors d'une expression *backquotée* et, inversement, les paramètres de la macro (ainsi que toute variable locale introduite par le travail de la macro) doivent être inclus dans une expression *virgulée*. Un dernier réglage fin, qui relève du cas d'espèce, fera choisir entre, par exemple, « `,(car exprs)` » et « `(car ,exprs)` » : veut-on insérer le premier élément de la valeur courante de `exprs`, ou bien générer un appel au `car` de la valeur retournée par l'expression `exprs` ?

Capture de fonction

La capture de fonction n'est pas un réel problème, mais le programmeur est néanmoins en droit de se poser des questions.

```
(defmacro mfoo (...)
    (labels ((foo (...) ...))
            `(... (foo ...) ...)))

(defun bar (...)
    (labels ((foo (...) ...))
            (mfoo ...)))
```

Dans cette séquence de code, la macro `mfoo` s'expanse, dans un environnement fonctionnel local contenant une définition de `foo` et par une expression contenant un appel à `foo`. De son côté, la fonction `bar` se sert de `mfoo` dans un environnement fonctionnel local contenant une autre définition de `foo`.

EXERCICE 3.68. Quelle est la définition de `foo` qui va être appelée par l'évaluation finale ? En l'absence d'argument convaincant dans un sens ou dans un autre, construire un petit test. □

Pour bien voir le problème, on peut le compliquer très légèrement :

```
(defmacro mfoo (...)
    (labels ((foo (...) ...))
            `(... (foo ...) ,(foo ...) ...)))
```

Quel `foo` désigne quoi ?

3.5.6 De l'usage des macros

De façon générale, les macros s'utilisent pour faire du *sucre syntaxique*, c'est-à-dire pour simplifier l'usage de certaines constructions couramment utilisées. Deux grands usages apparaissent.

Structures de contrôle

Les langages de programmation nécessitent un très petit nombre de structures de contrôle primitives, mais des structures de contrôle de plus haut niveau sont plus agréables. Les *macros* sont utilisées intensivement en LISP pour définir ces structures de contrôle de plus haut niveau, par exemple `cond`, `let`, `case`, `loop`, etc. Cet exercice n'est pas réservé aux concepteurs du langage : chaque programmeur peut aussi définir les structures de contrôle dont il a besoin en cas d'usage spécifique répété.

Remarque 3.28. Dans cet usage, la syntaxe d'utilisation d'une macro est absolument quelconque : cela dépend complètement de sa spécification et il est absolument impossible de donner une spécification générale de `<forme-macro>`, telle qu'elle est spécifiée dans la syntaxe des expressions évaluables (page 8). En particulier, si une `<forme-macro>` doit absolument s'expanser en une `<expr-eval>`, il n'est pas nécessaire que ses sous-expressions soit elles-mêmes évaluables.

Encapsulation de structures de données

Comme tous les langages évolués, LISP propose de nombreuses structure de données. Le programmeur peut les utiliser pour un usage très spécifique, en ce sens que la structure de donnée LISP est utilisée pour implémenter la structure de donnée spécifique de l'utilisateur. Par exemple, une liste `(a b . c)` constitue une structure à 3 champs. Il serait dangereux de mélanger les fonctions d'accès générales proposées par LISP avec l'usage spécifique qu'en fait le programmeur,

— d'une part parce qu'il est plus lisible de distinguer les 2 : comment reconnaître les `car` utilisés pour accéder au premier élément d'une liste quelconque, et les `car` utilisés pour accéder au premier champ de la structure de donnée spécifique implémentée par une liste ?

— d'autre part, le programmeur peut vouloir changer sa structure de donnée, par exemple utiliser un tableau (`array`), une liste dans un ordre différent (par exemple `((a . b) . c)`), ou toute autre structure. Le remplacement de tous les `car` par le nouvel accès serait à la fois fastidieux et dangereux.

Il faut donc impérativement *encapsuler* ses propres structures de données. Une première façon de faire est d'utiliser des fonctions, par exemple :

```
(defun get-a (x) (car x))
```

Cependant cet usage a le défaut de ne pas permettre de faire des `setf` : il faut donc faire aussi un accesseur en écriture :

```
(defun set-a (x y) (setf (car x) y))
```

C'est aussi inefficace : le compilateur LISP compile `car` beaucoup plus efficacement que n'importe quelle autre fonction et passer par l'intermédiaire de `get-a` est un frein non négligeable. Une macro

```
(defmacro get-a (x) `(car ,x))
```

a le double avantage de n'entraîner aucun surcoût et de permettre des `setf`. Elle a cependant l'inconvénient de ne pas permettre `function` et `apply`.

Remarque 3.29. Dans cet usage, la syntaxe d'utilisation d'une macro est la même que celle d'une vraie fonction.

Macros de compilation

L'idéal est donc de pouvoir définir des macros dites *ouvertes*, ou macro de compilation, qui ont le même nom qu'une fonction — les deux sont supposées équivalentes — mais qui ne sont utilisées que par le compilateur. Il est donc possible à la fois de faire des `setf` et des `function` et `apply`.

On définit une macro ouverte avec la forme syntaxique `define-compiler-macro` qui a la même syntaxe que `defmacro`.

3.5.7 Des macros à la compilation et à l'interprétation de LISP...

... il n'y a qu'un pas.

Le principe de la compilation (génération de code) ou de l'interprétation d'une expression revient à analyser cette expression (*analyse par cas*), et, pour chaque cas, à générer une nouvelle expression dans le langage cible (compilation) ou enfin à effectuer un calcul (interprétation). Dans cette optique, les macros représentent un mécanisme de génération de code dans lequel le langage cible est un sous-ensemble simple du langage source. Toutes les techniques utilisées pour les macros (*backquote*, introduction de variables, etc.) peuvent servir pour la compilation.

Inversement, tous les compilateurs et interprètes utilisent des *transformations source-à-source*, similaires aux macros, pour ramener les constructions complexes à des cas plus simples et éviter d'avoir à traiter de façon spécifique de trop nombreux cas.

4

Les Fonctions principales

Ce chapitre décrit succinctement les principales fonctions à connaître. C'est un index pour les deux chapitres précédents mais il liste aussi quelques fonctions moins importantes qui n'ont pas été décrites. Il doit être complété par la consultation du manuel en ligne, aussi bien pour une spécification plus précise et des exemples d'utilisation que pour les fonctions non décrites ici.

4.1 Les primitives

Ce paragraphe décrit tout (enfin, presque ...) ce qui ne peut pas s'écrire en LISP.

4.1.1 Évaluation et structures de contrôle

Dans la description des fonctions, les noms des *formes spéciales* et des *macros* sont en italiques, ainsi que ceux de leurs paramètres qui ne sont jamais évalués.

(EVAL `expr`) → ***résultat de l'évaluation de* `expr`**
Évalue l'expression `expr` et retourne le résultat de cette évaluation.

(*QUOTE* `expr`) → **`expr` *non évalué***
Retourne `expr` sans l'évaluer. S'abrévie par le macro-caractère « `'` ».

(*FUNCTION* `expr`) → ***valeur fonctionnelle de* `expr`**
Retourne la valeur fonctionnelle de `expr`, sans l'évaluer. L'argument `expr` doit être un nom de fonction (définie par `defun`) ou une λ-fonction. S'abrévie par le macro-caractère « `#'` ».

(*LAMBDA* `params` $\texttt{expr}_1 \ldots \texttt{expr}_n$) →
« Définit » la λ-fonction de paramètres `params` et de corps $\texttt{expr}_1$ ... $\texttt{expr}_n$, et la retourne.

((*LAMBDA* `params` $\texttt{expr}_1 \ldots \texttt{expr}_n$) $\texttt{arg}_1 \ldots \texttt{arg}_k$) → $\texttt{expr}_n$
Applique la λ-fonction de paramètres `params` et de corps $\texttt{expr}_1$ ... $\texttt{expr}_n$, aux arguments $\texttt{arg}_1$, ..., $\texttt{arg}_k$, c'est-à-dire évalue le corps $\texttt{expr}_1$ ... $\texttt{expr}_n$ de la λ-fonction, dans l'environnement des liaisons des paramètres `params` aux arguments $\texttt{arg}_i$.

(*DEFUN* `name params` $\texttt{expr}_1$ ... $\texttt{expr}_n$) → **`name`**
Définit la fonction globale de nom `name` en associant au symbole `name` la λ-fonction de paramètres `params` et de corps $\texttt{expr}_1$ ... $\texttt{expr}_n$. Retourne `name`. En général utilisé au *toplevel* de l'interprète et jamais à l'intérieur d'un autre `defun` (sauf à entraîner des effets difficilement prédictibles). Utilisé à l'intérieur d'un `let` ou assimilé, `defun` capture l'environnement englobant et fait une *fermeture*. La liste des paramètres `params` peut utiliser les mots-clés `&optional`, `&rest` et `&key`.

(*LABELS* `((`$\texttt{fn}_1$ λ`-list`$_1$ `body`$_1$`)...(`$\texttt{fn}_n$ λ`-list`$_n$ `body`$_n$`))` `body`)

(*FLET* `((`$\texttt{fn}_1$ λ`-list`$_1$ `body`$_1$`)...(`$\texttt{fn}_n$ λ`-list`$_n$ `body`$_n$`))` `body`) →
Permet de définir des fonctions locales (de portée lexicale) $\texttt{fn}_i$ utilisées dans le corps `body` de l'expression. Chaque `body` est un `progn` implicite. Avec `labels` les fonctions locales peuvent être récursives ou s'appeler l'une l'autre, alors que ce n'est pas possible avec `flet`.

(*DEFMACRO* name params $\texttt{expr}_1$ ... $\texttt{expr}_n$) $\rightarrow$ **name**

Définit la macro de nom `name` en associant au symbole `name` la λ-fonction de paramètres `params` et de corps $\texttt{expr}_1$ ... $\texttt{expr}_n$. Retourne `name`. Comme `defun` pour les imbrications. La liste de paramètres `params` autorise les mêmes mots-clés que `defun`, ainsi que la *déstructuration* (cf. `destructuring-bind`).

(*DEFCONSTANT var val*)

(*DEFPARAMETER var val*)

(*DEFVAR var val*) $\rightarrow$ **var**

Ces 3 formes spéciales ou macros introduisent le symbole `var` comme une variable dynamique ou globale, en lui affectant le résultat de l'évaluation de `val`. Retourne `var`. Ne peut être utilisé qu'au *toplevel* de l'interprète : ce ne peut pas être une sous-expression d'une autre expression.

La différence entre ces 3 formes est la suivante : `defconstant` introduit une constante qui ne doit plus être modifiée, π par exemple ; `defparameter` introduit un paramètre qui peut être modifié mais ne devrait pas être lié ; enfin, `defvar` introduit une variable pas forcément liée mais qualifiée de « spéciale » car ses liaisons devraient être dynamiques.

(*IF* cond then else) $\rightarrow$ **then / else**

Évalue `cond` puis, suivant que le résultat de l'évaluation est non `nil` ou pas, évalue `then` ou `else`. Retourne respectivement la valeur de `then` ou de `else`.

(*SETF var val*) $\rightarrow$ **val**

Expression d'affectation généralisée : évalue `val` et modifie la « liaison » de l'expression `var` — non évaluée et qui peut être n'importe quelle forme d'accès en lecture à une structure de donnée pour laquelle une forme en écriture existe aussi — pour lui affecter la nouvelle valeur. Retourne `val`.

À n'utiliser que dans le contexte d'une liaison, c'est-à-dire lorsque la variable a été introduite comme un paramètre de fonction, par `defun`, `lambda`, `labels` ou `let` !

`var` peut être une variable, une expression d'accès à une cellule `(car <expr>)` ou `(cdr <expr>)`, etc. Lorsque c'est un symbole, `var` doit être un paramètre d'une fonction ou λ-fonction qui englobe le `setf`, ou bien une variable dynamique ou globale qui a été introduite par `defvar`.

(FUNCALL fn $\texttt{arg}_1$...$\texttt{arg}_n$, n $\geq$ 0) $\rightarrow$ ***résultat de l'application de* fn**

Applique la fonction `fn` (le résultat de l'évaluation de) aux arguments $\texttt{arg}_1$ ... $\texttt{arg}_n$.

(APPLY fn $\texttt{arg}_1$...$\texttt{arg}_n$, n > 0) $\rightarrow$ ***résultat de l'application de* fn**

Applique la fonction `fn` (le résultat de l'évaluation de) aux arguments $\texttt{arg}_1$ ... $\texttt{arg}_n$, dont le dernier n'est pas *un* argument, mais la liste des arguments restants.

(PROGN $\texttt{expr}_1$...$\texttt{expr}_n$) $\rightarrow$ $\texttt{expr}_n$

Évalue successivement les expressions, et retourne le résultat de la dernière. `progn` sert à transformer une liste d'expressions en expression.

(PROG1 $\texttt{expr}_1$...$\texttt{expr}_n$) $\rightarrow$ $\texttt{expr}_1$

Évalue successivement les expressions, et retourne le résultat de la première. `prog1` évite d'introduire une variable auxiliaire pour mémoriser la première évaluation.

(UNWIND-PROTECT $\texttt{expr}_1$...$\texttt{expr}_n$) $\rightarrow$ $\texttt{expr}_1$

Comme `prog1`, mais en cas d'échappement dans l'évaluation de $\texttt{expr}_1$, les expressions suivantes sont quand même évaluées.

(*BLOCK name* $\texttt{expr}_1$...$\texttt{expr}_n$)

(*CATCH name* $\texttt{expr}_1$...$\texttt{expr}_n$) $\rightarrow$ $\texttt{expr}_n$ **/ *échappement***

Positionne une étiquette dynamique (dans la pile) de nom `name`, et évalue son corps comme un `progn`. Sans échappement, retourne la valeur de $\texttt{expr}_n$. En cas d'échappement, retourne la valeur retournée par l'échappement.

(*RETURN-FROM name* $\texttt{expr}_1$...$\texttt{expr}_n$)

(*THROW name* $\texttt{expr}_1$...$\texttt{expr}_n$) $\rightarrow$ $\texttt{expr}_n$

Évalue les expressions $\texttt{expr}_i$ puis fait un *échappement* à l'étiquette de nom `name` en « retournant » la valeur de $\texttt{expr}_n$ qui sera la valeur retournée par l'expression `(BLOCK name ...)`.

RETURN-FROM/THROW ne retourne rien au sens propre du terme. Le couple BLOCK/RETURN-FROM correspond à une liaison lexicale, alors que CATCH/THROW correspond à uen liaison dynamique.

4.1.2 Les listes

(CONS a d) → ***nouvelle cellule***
Alloue une nouvelle cellule, en faisant pointer son car sur a, et son cdr sur d. Retourne cette cellule.

(CAR list) → ***contenu du car***
Retourne le premier élément de la liste list. Le CAR d'une cellule (mais pas de nil) peut être modifié par SETF.

(CDR list) → ***contenu du cdr***
Retourne le reste de la liste list. Le CDR peut être modifié par SETF.

NB En COMMON LISP, comme dans beaucoup de dialectes LISP, les fonctions car et cdr sont étendues à nil, pour laquelle elles retournent nil.

4.1.3 Prédicats

Normalement, les prédicats de type sont constitués du nom du type, suffixé par la lettre « p » pour prédicat [1]. Tous les prédicats de ce genre (intergep, stringp, etc.) ne sont pas énumérés. Deux exceptions, atom et null.

(ATOM x) → **x / nil**
Teste si x est un atome ou une cellule (c'est-à-dire une liste non vide) : dans le premier cas, retourne t, sinon retourne nil.

(CONSP x) → **x / nil**
Teste si x est une cellule (une liste non vide) ou un atome : dans le premier cas, retourne x, sinon retourne nil.

(LISTP x) → **t / nil**
Teste si x est une liste (vide ou non) : si oui, retourne t, sinon retourne nil.

(NULL x) → **t / nil**
Teste si x est la liste vide (nil) : si oui, retourne t, sinon retourne nil.

(EQ x y)

(EQL x y)

(EQUAL x y) → **t / nil**
Les fonctions eq et equal testent respectivement l'égalité physique et logique de x et y. Elles retournent t ou nil.

La fonction eql est une version intermédiaire qui retourne t si ses arguments sont eq ou si ce sont des nombres de même type égaux (fonction =) ou des caractères égaux (fonction char-equal).

4.2 Les constructions usuelles non primitives

Tout ce qui peut s'écrire en LISP.

4.2.1 Contrôle

Toutes ces constructions sont (ou pourraient être) des macros.

(*AND* $\mathbf{expr}_1 \ldots \mathbf{expr}_n$) → **nil / $\mathbf{expr}_n$**
Évalue les expressions $\mathtt{expr}_i$ tant qu'elles retournent non nil. S'arrête à la première qui retourne nil en retournant nil. S'il n'y en n'a pas, retourne le résultat de $\mathtt{expr}_n$.

1. En SCHEME, la convention utilise le suffixe « ? ».

(*OR* expr$_1$...expr$_n$) → nil / expr$_i$
Évalue les expressions expr$_i$ tant qu'elles retournent nil. S'arrête à la première qui retourne non nil en retournant cette valeur.

(NOT expr) → t / nil
Évalue expr et retourne sa négation : nil si non nil, t si nil.

(*LET* ((*var*$_1$ val$_1$)...(*var*$_k$ val$_k$)) expr$_1$ ... expr$_n$) → expr$_n$
Cette macro s'expanse par l'application de la λ-fonction de paramètres var$_i$ et de corps expr$_1$ ... expr$_n$ aux arguments vali.

(*LET ((*var*$_1$ val$_1$)...(*var*$_k$ val$_k$)) expr$_1$ ... expr$_n$) → expr$_n$**
Cette macro s'expanse par l'application de la λ-fonction de paramètres var$_1$ et de corps (let * ((var$_2$ val$_2$) .. (var$_k$ val$_k$)) expr$_1$ ... expr$_n$) à l'argument val$_1$. C'est la version *séquentielle* du let.

(*DESTRUCTURING-BIND treevar* val expr$_1$ ... expr$_n$) → expr$_n$
Comme let mais en liant l'arbre des paramètres treevar à l'argument val par déstructuration (cf. paragraphe 6.3).

(*COND* (cond$_1$ expr$_{11}$...expr$_{1k_1}$)...(cond$_n$ expr$_{n1}$...expr$_{nk_n}$)) → expr$_{jk_j}$ / cond$_j$

Version généralisée du if. Le premier cond$_i$ dont l'évaluation retourne non nil voit la séquence d'expressions expr$_{ij}$ qui suit évaluée : le cond retourne la valeur de expr$_{ik_i}$ si $k_i > 0$, sinon la valeur de cond$_i$.

(*CASE* expr (*pat*$_1$ expr$_{11}$...expr$_{1k_1}$) ...(*pat*$_n$ expr$_{n1}$...expr$_{nk_n}$))

(*ECASE* expr (*pat*$_1$ expr$_{11}$...expr$_{1k_1}$) ...(*pat*$_n$ expr$_{n1}$...expr$_{nk_n}$)) → expr$_{jk_j}$
Évalue l'expression expr, puis compare son résultat avec chacun des motifs pat$_i$ qui sont soit un atome (qui doit être égal à la valeur), soit une liste (qui doit contenir la valeur), soit être t pour le dernier, pour que le test réussisse : à ce moment là, le corps de la clause (expressions expr$_{i1}$...expr$_{ik_i}$ est alors évalué comme un progn. La macro ecase retourne une erreur si aucun des motifs n'est satisfait.

4.2.2 Les formes itératives

Elles n'ont rien de primitif car on peut les obtenir à partir de la forme while.

Les fonctions de mapping

Ces fonctions, dites de *mapping*, sont classiques en LISP et COMMON LISP en donne une grande variété. Leur structure commune est la suivante :

(<mapfn> fn list$_1$...list$_n$), n ≥ 1 →
Application successive de la fonction fn d'arité n, soit à tous les éléments successifs des n listes (mapping sur les car), soit aux n listes elles-mêmes et à leurs cdr successifs (mapping sur les cdr), en construisant un résultat par application d'un opérateur sur les résultats, suivant le tableau ci-dessous :

	car	cdr
nil list nconc	mapc mapcar mapcan	mapl maplist mapcon
list or and	map some every reduce	-- -- -- --

NB. Les fonctions de *mapping* map, some, every et reduce prennent pour argument non seulement des listes mais, plus généralement, n'importe quelles *séquences* (liste, vecteur ou chaîne, cf. paragraphe 4.3.2).

La macro loop

(*LOOP* `iterateurs` $\texttt{expr}_1 \ldots \texttt{expr}_n$) → `nil`

Macro générale d'itération : pour plus de détails, voir le chapitre spécial qui lui est consacré dans le manuel en ligne.

4.2.3 Les fonctions usuelles sur les listes

Toutes ces fonctions peuvent s'écrire en LISP, sont écrites en LISP, et la plupart ont été faites en cours ou en TD. Les fonctions qui utilisent un test d'égalité peuvent en général le préciser en premier argument `&optional`. Les fonctions qui commencent par N font de la modification physique, les autres de la copie ou du partage.

(APPEND $\texttt{l}_1 \ \ldots \texttt{l}_n$**)**

(NCONC $\texttt{l}_1 \ldots \texttt{l}_n$**) → *concaténation des listes***

Concatène ses arguments l_i qui doivent être des listes. `append` fait une copie de ses $n-1$ premiers arguments, contrairement à `nconc` qui les modifie physiquement.

(MEMBER `x list &key(test #'eql) key from-the-end`) → *sous-liste ou* `nil`

Recherche l'élément `x` dans la liste `list`, et retourne la sous-liste commençant par `x`, `nil` sinon. `test` est la fonction de comparaison utilisée (par défaut `eq`).

À compléter.

4.3 Les autres types

4.3.1 Les symboles et les packages

Sur ces notions, et les fonctions associées, voir le manuel de référence COMMON LISP. Ces fonctions provoquent une erreur lorsque l'argument n'est pas un symbole.

(SYMBOLP `x`) → `t / nil`

Teste si `x` est un symbole : si oui retourne `t`, sinon retourne `nil`.

(KEYWORDP `x`) → `t / nil`

Teste si `x` est un `keyword`, c'est-à-dire un symbole constant préfixé par « `:` » : si oui retourne `t`, sinon retourne `nil`.

(GET `symb prop`) →

Les symboles sont des structures de données auxquelles ont peut associer des *propriétés*, c'est-à-dire des symboles qui servent à accéder à des valeurs. La fonction `get` retourne la valeur associée à la propriété `prop` du symbole `symb`. Si la propriété est absente, elle retourne `nil`. `get` est *`setf`-able* (affectable par `setf`). Les 2 paramètres `prop` et `symb` sont des symboles : en pratique, `symb` est le résultat d'un calcul alors que `prop` est une constante pour laquelle on utilise en général des *keywords*.

(FBOUNDP `symb`) → `t / nil`

Teste si le symbole `symb` a une définition de fonction : si oui retourne `t`, sinon retourne `nil`.

(SYMBOL-FUNCTION `symb`) → *valeur fonctionnelle ou exception*

Retourne la valeur fonctionnelle associée au symbole `symb`. Retourne une exception si `symb` n'a pas de définition de fonction. Est affectable par `setf`.

(MACRO-FUNCTION `symb`) → *valeur fonctionnelle ou* `nil`

Retourne la valeur de macro associée au symbole `symb`. Utilisable de façon booléenne et affectable par `setf`.

(INTERN `pkg str`) → *symbole*

Crée, s'il n'existe pas déjà, et retourne le symbole de nom `str` dans le package `pkg` (par défaut, `nil`).

(GENSYM) → ***symbole***
Crée un nouveau symbole par incrémentation d'un compteur.

(*PRETTY* `fn`) → **`fn`**
Imprime joliment la définition de la fonction `fn`, en appliquant `pprint` à l'expression de définition de `fn`.

4.3.2 Les séquences

Les séquences sont une généralisation des listes, vecteurs et chaînes. De nombreuses fonctions polymorphes leurs sont dédiées, en plus des fonctions spécifiques à chacun de ces types.

(ELT `seq n`) →
Retourne la valeur de la `n`-ième position de la séquence `seq`. Les indices sont comptés à partir de 0. Est affectable par `setf`.

(LENGTH `seq`) → ***integer***
Retourne la longueur de la séquence `seq`.

(REVERSE `seq`)

(NREVERSE `seq`) → ***liste***
Inverse une séquence physiquement (`nreverse`), ou retourne une copie inversée de la séquence (`reverse`).

Les chaînes de caractères

Outre les primitives qui sont décrites ci-dessous, LISP propose un ensemble de fonctions de manipulation de chaînes.

(STRINGP `x`) → **`x / nil`**
Teste si `x` est une chaîne de caractères : si oui, retourne `x`, sinon `nil`.

(MAKE-STRING `n cn`) → ***string***
Crée une chaîne de longueur `n`, initialisée avec le caractère (code ASCII) `cn`.

(STRING `x`) → ***string***
Retourne la chaîne associée à `x` qui doit être un nombre ou un symbole. C'est l'équivalent de `toString` de JAVA.

(CHAR `str n`) → ***caractère***
Retourne le `n`-ième caractère de la chaîne `str`. Les indices sont comptés à partir de 0. Est affectable par `setf`.

Ces fonctions sont des primitives : elles permettent d'écrire de nombreuses fonctions utiles non primitives.

Les vecteurs et tableaux

LISP propose aussi des tableaux et vecteurs (c'est-à-dire des tableaux de dimension 1)[2]. Outre les primitives qui sont décrites ci-dessous, LISP propose un ensemble de fonctions de manipulation de vecteurs.

(ARRAYP `x`) → **`x / nil`**
Teste si `x` est un tableau : si oui, retourne `x`, sinon `nil`.

(VECTORP `x`) → **`x / nil`**
Teste si `x` est un vecteur : si oui, retourne `x`, sinon `nil`.

(MAKE-ARRAY `ln`) → ***tableau***
Crée un tableau de dimensions `ln` où `ln` est une liste d'entiers dont chacun représente une dimension du tableau.

(VECTOR `val`$_1$`...val`$_n$) → ***vecteur***
Retourne le vecteur de longueur `n` contenant les valeurs des expressions `val`$_i$.

2. Il n'y a pas en LISP la distinction bizarre de JAVA entre les 2 termes.

(AREF tab n_1 .... n_k) →
Retourne la valeur de la case d'indices n_j du tableau `tab`. Les indices sont comptés à partir de 0. Est affectable par `setf`.

Ces fonctions sont des primitives : elles permettent d'écrire de nombreuses fonctions utiles non primitives.
Voir aussi la spécification de la macro `loop` pour voir comment itérer sur des séquences.

Les tables de hachage

(MAKE-HASH-TABLE) → ***table de hachage***
Crée une table de hachage. Voir le manuel pour les paramètres optionnels, par exemple le test d'égalité à utiliser.

(GETHASH key ht &optionaldefault) →
Retourne la valeur associée à la clé `key`, par défaut la valeur du paramètre `default`, par défaut `nil`. Est affectable par `setf`.

4.3.3 L'arithmétique

COMMON LISP propose plusieurs arithmétiques spécifiques différentes — entière, flottante, rationnelle, `bignum`, complexe — ainsi qu'une arithmétique générique. Se reporter au manuel pour plus de détails.

(INTEGERP x) → **x / nil**
Teste si `x` est un petit entier : si oui, retourne `x`, sinon `nil`.

À compléter.

4.4 Les entrées-sorties

4.4.1 Lecture et écriture

Toutes ces fonctions ont pour premier argument optionnel le canal d'entrée ou sortie, par défaut le canal d'entrée ou sortie courant.

Lecture

(READ &optional stream eof-error eof-value) →
Lit une expression et la retourne.

(READ-CHAR &optional stream) → ***caractère***
Lit un caractère et le retourne : retourne le caractère courant, et avance.

(READ-LINE &optional stream) →
Lit une ligne et la retourne comme une chaîne de caractères.

(PEEK-CHAR &optional stream) → ***caractère***
Consulte un caractère et le retourne : retourne le caractère courant, et n'avance pas.

Écriture

(PRIN1 expr &optional stream)

(PRINC expr &optional stream) → **expr**
Ces fonctions impriment l'expression et retournent sa valeur. La première fonction, au contraire de la seconde, imprime les caractères d'échappement nécessaires à une relecture de la forme imprimée par la fonction `read`.

(PRINT expr &optional stream) → **expr**
C'est `prin1` précédé de `terpri` et suivi d'un espace.

(PRIN-CHAR cn &optional stream) → **cn**
Imprime le caractère `cn`.

(TERPRI &optional stream) → **t**
Passe à la ligne (et vide le tampon).

(PPRIN expr &optional stream) → **expr**
Comme `print` mais l'impression est indentée (*pretty-print*).

(FORMAT stream form &rest args) → **nil** *ou string*
Imprime sur le canal de sortie `stream`, les arguments `args`, d'après le format `form`. Ce dernier paramètre est une chaîne de caractères qui contient des « directives » : ces directives sont préfixées par « ∼ ». La directive universelle pour insérer simplement un argument est « ∼S ». Les possibilités, innombrables, ne seront pas décrites ici.

PRINT-LENGTH

PRINT-LEVEL → **nil** variable
Variables dynamiques qui limitent l'impression de listes en profondeur ou longueur. Doivent avoir une valeur entière positive pour que l'on puisse imprimer des listes circulaires.
Faire `*print-` suivi de « → » pour avoir la liste d'autres variables pour paramétrer `print`.

4.4.2 Streams et fichiers

(OPEN filename &key:direction :if-exists :if-does-not-exist) →
Ouvre un canal en lecture (resp. écriture, lecture-écriture) suivant la valeur de l'argument `:direction`, `:input` (resp. `:output` ou `:io`) sur le fichier de nom `filename` et retourne ce canal. L'argument `:if-exists` (resp. `:if-does-not-exist`) permet de préciser ce qu'il faut faire si le fichier existe déjà en écriture (resp. n'existe pas en lecture), avec des valeurs possibles comme `:error` ou `:append`.

(CLOSE &optional can) → **nil**
Ferme le canal `can`, ou tous les canaux s'il n'y a pas d'arguments.

STANDARD-INPUT

STANDARD-OUTPUT

ERROR-OUTPUT

TRACE-OUTPUT

TERMINAL-IO

DEBUG-IO → **stream** variable
Ces variables dynamiques désignent les canaux prédéfinis les plus courants. On s'en sert habituellement en les liant (par `let`) aux canaux que l'on désire utiliser.

4.5 Gestion des exceptions

(*ASSERT* cond &rest body) → **body / nil** *ou exception*
Lorsque le corps (`body`) du `assert` n'est pas vide, `assert` équivaut à `unless`. Lorsque le corps est vide, l'évaluation à `nil` de la condition `cond` entraîne une erreur. `assert` correspond donc à une précondition.

(ERROR format) &rest args

(CERROR format &rest args)

(WARN format &rest args) → **nil**
Ces fonctions signalent une erreur en imprimant un message constitué à partir d'un `format` et des arguments `args`, sur le canal d'erreur (par défaut le canal de sortie courant). De plus, les fonctions `error` et `cerror` provoquent une interruption de l'évaluation en cours et l'ouverture d'une boucle d'inspection. Dans le cas de `cerror`, l'erreur est continuable.

(EXIT) →
Pour sortir de COMMON LISP.

5

L'environnement spécifique à CLISP

Il y a de nombreuses implémentations de COMMON LISP, souvent commerciales et payantes. Le cours utilise l'implémentation gratuite de GNU, nommée CLISP. Pour plus de détails, consulter le manuel CLISP [CLI97a, CLI97b] disponible sur le site `http://clisp.cons.org/`.

CLISP fait partie des distributions Linux usuelles ou est téléchargeable sur le site Web `http://clisp.cons.org/`. Il est aussi porté sur MAC et sur CYGWIN, mais pas directement sur Windows. Pour Windows, il doit aussi exister des implémentations alternatives, qui ne sont pas forcément gratuites...

Remarque 5.1. Ces informations ne sont clairement pas définitives. Celui qui souhaite installer CLISP sous Windows doit regarder sur le site de CLISP, et éventuellement faire une recherche systématique sur le Web.

5.1 Manuel en ligne de COMMON LISP et de son implémentation

Le manuel de COMMON LISP est disponible sur le réseau de l'UFR à l'URL `file:/net/local/doc/cltl/clm/clm.htm` (ou à une adresse voisine). C'est le contenu exact de [Ste90] et la norme ANSI. Vous pouvez y accéder par la table des matières ou par l'index de toutes les fonctions.

Pour un usage plus avancé, [CLI97b] décrit les écarts de l'implémentation de CLISP par rapport à cette norme.

5.2 Lancement de COMMON LISP

Des PCs sous Linux on le lance par la simple ligne de commande : `clisp`.

On sort en tapant `C-d` ou par l'appel de la fonction (`exit`).

Pour plus de détails sur les paramètres d'appel, voir [CLI97a] ou faire `man clisp`.

5.3 Editeur de ligne

Le *toplevel* CLISP est muni d'un éditeur de ligne, similaire à un éditeur de commandes sous *shell* (`ksh`, `bash` ou `tcsh`), avec des commandes à la Emacs.

Parmi les fonctionnalités précieuses,

— la complétion de symboles, par le caractère `TAB` (→), permet de retrouver facilement les fonctions existantes ;

— on remonte aux expressions tapées précédemment avec la ↑ ou `C-p`, ou enfin avec une recherche en arrière `C-r` ;

L'éditeur de ligne de CLISP utilise la fonctionnalité *readline* de GNU : `http://tiswww.case.edu/php/chet/readline/rluserman.html`.

5.4 Fonctions de traçage

(*TRACE* `fn`$_1$ `...` `fn`$_n$`, n > 0`) $\rightarrow$
Trace les fonctions en argument — chaque appel d'une fonction tracée affiche un message à l'entrée et à la sortie de la fonction, avec les valeurs des paramètres et du retour. On peut aussi tracer certaines méthodes des fonctions génériques (pour les grands).

(*UNTRACE* `fn`$_1$ `...` `fn`$_n$`, n` $\geq$ `0`) $\rightarrow$
Enlève la trace des fonctions en argument, de toutes les fonctions tracées s'il n'y a pas d'argument.

5.5 Le mode *debug*

Chaque exception ouvre une boucle d'inspection `read-eval-print` dans l'environnement de l'erreur : c'est le mode *debug*. Si le code erroné est compilé, il n'y a rien à faire — si ce n'est charger le code interprété pour (essayer de) faire apparaître l'erreur. En mode interprété, on a accès aux valeurs de toutes les variables et on peut se déplacer dans la pile. En mode *debug*, le prompt est précédé du niveau de profondeur (chaque erreur l'incrémente de 1) : il ne faut pas rester en mode *debug* et il faut penser à en sortir.

Les commandes du mode *debug* sont des symboles réservés, tapés comme pour évaluer une variable, qui ne sont pas évalués (mais uniquement dans ce mode). Les principales sont :

- la commande `help` liste les commandes disponibles ;
- `abort` permet de sortir d'un niveau de *debug* ;
- `C-d` (contrôle-d) a un effet similaire à `abort`, sauf qu'il a un effet de *reprise* de l'évaluation dans certains cas (après un *break* ou `C-c` par exemple) : en cas de boucle infinie, il est impératif de sortir par `abort` ;
  attention, trop de `C-d` font sortir de l'environnement CLISP ;
- `where` affiche l'expression dans laquelle a eu lieu l'erreur ;
- `up` permet de remonter jusqu'à l'endroit où on peut identifier le contexte et analyser la valeurs des variables de l'environnement courant ;
- `down` permet de redescendre.

Le bon usage du mode *debug* consiste à rechercher la cause de l'erreur, en commençant par `where` puis en remontant (`up`) jusqu'à arriver au niveau où la cause est identifiable. A tout moment, les variables de l'environnement courant peuvent être inspectées. Lorsque l'erreur est identifiée, il faut sortir du mode *debug* par `abort` ou `C-d`.

Le mode *debug* ne marche que pour du code interprété : lorsque l'erreur survient dans une fonction compilée, l'erreur est localisée à l'interface de l'interprété et du compilé, lorsque l'appelant est interprété et l'appelé compilé.

5.6 Fichiers de définitions

Bien qu'il soit possible de définir les fonctions au *toplevel* de l'interprète, il est plus commode de les définir dans des fichiers et de charger ces fichiers par la fonction suivante.

(LOAD `file`) $\rightarrow$ **`t`**
Charge le fichier de nom `file`, c'est-à-dire exécute sur ce fichier une boucle `READ-EVAL` jusqu'à la fin du fichier.

D'un point de vue méthodologique, on évitera de mélanger des définitions et des exécutions : d'une part, la fonction `load` ne fait pas de `print` et ne permet pas de vérifier les exécutions, d'autre part il faut pouvoir charger les définitions sans entraîner des erreurs dues à des tests incorrects. Donc si vous voulez faire des tests dans un fichier, faites le dans un fichier séparé.

Editeurs de texte En CLISP, les fichiers de définition de fonctions ont par défaut l'extension "`.lsp` " ou "`.lisp` ". Les "bons" éditeurs de texte (`emacs`, `vim`, `eclipse`) ont un mode LISP et reconnaissent ces extensions. Dans le mode LISP, ils mettent en valeur les « `(` `)` » et indentent automatiquement.

Pour `emacs`, il peut être nécessaire d'activer la mise en valeur des parenthèses (menu "Options", *Paren Match Highlighting*) et il faut alors sauvegarder les options (même menu, *Save Options*).

Des informations concernant SLIME, un mode pour `emacs` dédié à LISP, se trouvent à cette adresse : `http://common-lisp.net/project/slime/`.

Indentation En complément de la mise en valeur des parenthèses, il est indispensable de disposer de l'indentation automatique : c'est ce qui va permettre de reconnaître d'un seul coup d'œil, dans quelle expression est incluse une sous-expression. En général, l'indentation d'une ligne est automatique mais le passage à la ligne est manuel : s'inspirer des exemples de définition de fonctions dans les chapitres 2 et 3.

Sous `emacs`, l'indentation d'une ligne par rapport à la précédente s'obtient avec la touche `tab` (ou →). On peut indenter globalement une région (par exemple la définition d'une fonction ou la totalité d'un fichier) en sélectionnant cette région puis en faisant `M-x indent-region`. Enfin, `M-x indent-sexp` (ou `C-M-q`) indente à partir de la ligne courante.

Sous `vim`, mettre `set autoindent` dans le fichier `.vimrc` : le curseur se positionne automatiquement à chaque passage à la ligne.

5.7 La compilation

LISP est un langage interprété muni d'un compilateur. Deux fonctions permettent de compiler une fonction particulière, ou un fichier, c'est-à-dire toutes les fonctions d'un fichier. On ne compilera bien sûr que les fichiers de définitions, jamais les fichiers de tests. Voir aussi la section 2.8.

(COMPILE `fn`) →
: Compile la fonction de nom `fn`.

(COMPILE-FILE `file`) →
: Compile le fichier source de nom `file` pour produire un fichier compilé : attention, il faut charger ensuite ce fichier par la fonction `load` pour pouvoir évaluer le code compilé.

  Diverses options sont disponibles (cf. manuel [CLI97b]).

En CLISP, les fichiers de code compilé, produits par la fonction `compile-file`, ont l'extension "`.fas`".

6

Recueil d'exercices

Une bonne λ-abstraction vaut toujours mieux qu'un mauvais copier-coller.

Ce chapitre regroupe des problèmes (sans les solutions) qui ont été donnés en sujet d'examen ou de TD ces dernières années. Inutile de dire qu'ils constituent de bons exercices. De très nombreux autres exercices sont disponibles dans les nombreux manuels et sur le Web.

6.1 Fonctions de tri

On suppose donné un prédicat de comparaison, λ-expression à deux arguments, qui retourne non `nil` ou `nil` suivant que le test de comparaison est vérifié ou pas.

Il s'agit de définir l'ensemble des fonctions de tri suivantes, en cherchant un compromis simplicité / efficacité. Si les algorithmes optimaux ne sont pas recherchés, il s'agit de se servir des propriétés des arguments, en particulier du fait qu'une liste est déjà ordonnée ou pas.

(BEST `fn ll`) $\rightarrow$ ***élément***
: Recherche dans la liste `ll` non ordonnée le meilleur élément d'après `fn`.

(INSERT `fn x ll`) $\rightarrow$ ***liste***
: Insère dans la liste `ll` déja ordonnée d'après `fn` l'élément `x`.

(MERGE `fn l1 l2`) $\rightarrow$ ***liste***
: Fusionne les listes `l1` et `l2` déja ordonnées d'après `fn`.

(SORT `fn ll`) $\rightarrow$ ***liste***
: Trie la liste `ll` d'après `fn`.

Mis à part la première, ces trois fonctions peuvent se faire en modification physique (`ninsert`, `nmerge` et `nsort`) ou en partage (`insert`, `merge` et `sort`).

Les *linéarisations* utilisées en programmation par objets pour l'héritage multiple offrent des variations intéressantes autour de la fonction `merge` : voir [Duc13], Section 4.5.

6.2 Recherche dans un graphe

On suppose un graphe donné par une fonction successeur, et une recherche donnée par un prédicat de succès. Une fonction générique de recherche dans un graphe prend alors comme argument une liste de sommets, les fonctions successeur et succès, et une fonction de combinaison des sommets restants avec les successeurs du sommet courant.

Exemple : un arbre binaire virtuel (et virtuellement infini) dont la racine est `1` et la fonction successeur `(lambda (n) (list (* 2 n) (1+ (* 2 n))))`.

(SEARCH-GRAPH `nodes succ goalp combiner`) $\rightarrow$
: C'est la fonction générique de recherche, où `nodes` est une liste de sommets, `succ` la fonction successeur, `goalp` le prédicat de succès et `combiner` la fonction de combinaison.

(DEPTH-FIRST-SEARCH `nodes succ goalp`) →
Consiste en un appel particulier à la fonction `search-graph`, avec une fonction de combinaison spécifique pour un parcours *en profondeur*.

(BREADTH-FIRST-SEARCH `nodes succ goalp`) →
Consiste en un appel particulier à la fonction `search-graph`, avec une fonction de combinaison spécifique pour un parcours *en largeur*.

(BEST-FIRST-SEARCH `nodes succ goalp cost`) →
Consiste en un appel particulier à la fonction `search-graph`, avec une fonction de combinaison spécifique pour une recherche suivant le meilleur coût, suivant la fonction `cost`, tous les sommets étant exploré dans l'ordre des coûts croissants. Attention de ne pas trier deux fois ce qui l'est déjà : il faut choisir la fonction de tri la plus appropriée. Attention aussi aux problèmes de laison (dynamique ou fermeture lexicale) qui se posent.

(FOCUS-FIRST-SEARCH `nodes succ goalp cost focus-max`) →
Cas particulier de la fonction précédente dans lequel on ne s'intéresse qu'aux `focus-max` meilleurs sommets.

(GREEDY-SEARCH `nodes succ goalp cost`) →
Cas particulier de la fonction précédente où l'on ne s'intéresse qu'à la meilleur solution. C'est une stratégie *gloutonne* (*greedy*) appelée classiquement *hill climbing*. Attention : il ne suffit bien sûr pas d'appeler la fonction précédente en lui passant une valeur de 1 pour `focus-max` : il faut définir une version plus efficace qui tienne compte de cette spécificité.

6.3 Déstructuration et appariement

6.3.1 Déstructuration

La déstructration est le mécanisme de liaison paramètres-arguments de l'évaluateur de LELISP qui n'est utilisé en COMMON LISP que par la forme spéciale `destructuring-let`, ainsi que pour les macros. Une *liste d'arguments* est déstructurée par un *arbre de paramètres* suivant le principe suivant :

paramètre	argument	action
`cons`	`cons`	double récursion
`nil`	`nil`	
variable	quelconque	liaison
autre		échec

Dans le cas d'une variable, la variable ne doit pas être rencontrée deux fois dans l'arbre des paramètres. La déstructuration retourne un environnement, c'est-à-dire un ensemble de liaisons variable-valeur, représentée par une liste d'association (*A-liste*).

Il est possible de généraliser la destructuration pour autoriser dans l'arbre de paramètres, non seulement des occurrences multiples de la même variable (qui doivent alors s'apparier avec le même argument) mais aussi des constantes atomiques, qui doivent alors s'apparier avec elles-mêmes.

(DESTRUCT `param arg alist`) → ***`a-liste ou`*** **`:fail`**
Déstructure les arguments `arg` d'après l'arbre de paramètres `param` dans l'environnement `alist`. Retourne `:fail` si la déstructuration échoue, l'environnement, éventuellement augmenté, si elle réussit.

6.3.2 Appariement

L'appariement (*pattern-matching*) d'un *motif* (*pattern*) et d'une expression, est une généralisation de la déstructuration dans laquelle, le motif, qui généralise l'arbre de paramètre peut contenir :

— des constantes,
— plusieurs occurrences de la même variable ;
— différents types de variables.

Du fait de ces généralisations, il est nécessaire d'avoir une représentation particulière pour les variables : on utilisera une cellule dont le `car` contient un marqueur spécifique indiquant le type de la variable (par exemple ?, ?∗, ?+, etc.) et le `cdr` le nom de la variable, c'est-à-dire un symbole LISP.

Si l'on ne considère que des variables « simples », qui s'apparient avec l'élément correspondant de l'expression, le principe de l'appariement est le suivant :

motif	expr	action
`cons`	`cons`	double récursion
constante	égalité	
variable (1ère)	quelconque	liaison
variable (2ième)	égalité	
autre		échec

(PM-VARIABLE-P `expr type`) → **`t / nil`**
Prédicat qui vérifie si son argument `expr` est une variable du type `type`, c'est-à-dire une cellule dont le `car` est égal à `type`.

(MATCH-VARIABLE `var expr alist`) → ***a-liste ou*** **:fail**
Apparie la variable `var` et l'expression `expr`, en vérifiant si c'est la première occurrence de la variable ou pas. Retourne `fail` si l'appariement échoue, l'environnement, éventuellement augmenté, s'il réussit.

(PAT-MATCH `pattern expr alist`) → ***a-liste ou*** **:fail**
Apparie l'expression `expr` avec le motif `pattern` dans l'environnement `alist`. Utilise `match-variable` dans le cas où le motif est une variable du pattern-matching. Retourne `fail` si l'appariement échoue, l'environnement, éventuellement augmenté, s'il réussit.

On étend ensuite le pattern-matching pour traiter des variables d'autres types :
— variable segment qui peuvent s'apparier avec une sous-liste de l'expression : de type ?∗ pour s'apparier avec une liste quelconque, de type ?+ pour s'apprier avec une liste non vide.
— variable simple qui ne peuvent s'apparier qu'avec des expressions d'un certain type : par exemple, les variables de type ?% ne peuvent s'apparier qu'avec des nombres.

(MATCH-SEGMENT `pattern expr alist init`) → ***a-liste ou*** **:fail**
Apparie le motif `pattern` qui commence par une variable segment et l'expression `expr`, en en prenant au moins `init` éléments. On ne traite pas l'indéterminisme (pas de backtrack) et on peut chercher à optimiser en traitant le cas où la variable est déjà appariée, et sinon en se servant des éléments qui suivent dans le motif.

(PAT-MATCH `pattern expr alist`) → ***a-liste ou*** **:fail**
Étend la définition précédente en traitant le cas des diverses variables segments et de la variable simple à contrainte numérique.

6.3.3 Filtrage

L'appariement a deux utilisations naturelles. L'une est bien connue et conduit à la réécriture aux systèmes experts et aux règles d'inférences (Cf. Section 6.3.5). Une autre utilisation conduit à la programmation par filtrage [Que90] : le corps d'une fonction est constitué de plusieurs clauses (similaires aux clauses d'un `cond`), ayant chacune leur λ-liste. On évalue le corps de la première clause dont la λ-liste s'apparie avec la liste d'argument.

La macro suivante est une forme analogue au `cond` qui réalise ce filtrage. On peut la mettre en œuvre avec n'importe quelle forme d'appariement, ici la déstructuration la plus générale, avec constantes et occurrences multiples de variables.

(*DESTRUCTURING-CASE* `expr` (`pat`$_1$ `e`$_{11}$ . . `e`$_{1k_1}$) . . (`pat`$_n$ `e`$_{n1}$ . . `e`$_{nk_n}$)) → **`e`**$_{jk_j}$ **`/ nil`**
Évalue l'expression `expr`, puis apparie son résultat avec chacun des motifs `pat`$_i$ jusqu'à ce qu'un de ces appariements marche : le corps de la clause correspondante (expressions `e`$_{i1}$...`e`$_{ik_i}$) est alors évalué comme un `progn`.

6.3.4 Unification

L'*unification* de Prolog est une extension de l'appariement dans laquelle la donnée peut, elle aussi, contenir des variables. Il n'y a donc plus de distinction entre la donnée et le motif, mais 2 motifs dont les ensembles de variables ne sont pas forcément disjoints. L'unification de 2 variables X et Y peut produire aussi bien la paire $(X\ .\ Y)$ que $(Y\ .\ X)$ si les deux variables ne sont pas déjà appariées, ou $(X\ .\ a)$ si Y est déjà apparié à a et si X n'est pas encore apparié. Exemple :

motif	motif	résultat
(a X b)	(a c Y)	((X . c)(Y . b))
(a X Y)	(a Y b)	((X . Y)(Y . b)) ou ((X . b)(Y . b))
(a X X)	(a Y Y)	((X . Y)) ou ((Y . X))

Question 6.3.4.1

On se place d'abord dans le cas de l'unification de 2 listes plates.

1. La fonction `assoc` est donc insuffisante pour rechercher la constante associée à une variable dans une liste d'association produite par l'unification. Ecrire la fonction `unify-assoc` qui retourne la constante à laquelle est associée une variable, si elle existe, `nil` si la variable n'est unifiée qu'à une autre variable, elle-même unifiée à aucune constante, et `fail` si la variable n'est unifiée à rien.
2. Ecrire la fonction `unify-variable` qui unifie une variable avec autre chose (constante ou variable).
3. Ecrire la fonction `unify-list` qui unifie 2 listes ;
4. que donne votre fonction sur les exemples suivants :

(a X X)	(a Y Y)	?
(a X Y)	(a Y X)	?
(a X X X)	(a Y Y Y)	?

5. Avant de passer à l'unification d'arbres, examinez les cas suivants : que doit retourner leur unification ?

(a X X)	(a (a Y) (a b))	?
(a X X)	(a (a Y) Y)	?

6. A la lumière de ces 2 cas, comment faut-il modifier la fonction `unify-variable` pour l'unification d'arbres ?
7. Ecrire la fonction `unify-tree` qui unifie 2 arbres ;

6.3.5 Règles de réécriture

Une règle de récriture est constituée d'un motif et d'une production qui sont tous deux des arbres de paramètres. Le principe consiste à apparier le motif avec une donnée : si l'appariement réussit, on remplace dans la production chaque paramètre par la donnée à laquelle elle est associée.

En pratique, on dispose d'un ensemble ordonné de règles de réécriture et on applique à la donnée la première règle qui s'applique, jusqu'à ce qu'il n'y en ait plus. L'arrêt dépend bien entendu de la structure des règles. Des règles "simplificatrices", où la taille de la production est toujours strictement inférieure à celle du motif, s'arrêteront toujours.

La réécriture peut être indéterministe, en particulier avec les variables segments.

EXERCICE 6.1. Définir la fonction `rewrite-1` qui prend en entrée une donnée, un motif et une production et réécrit la donnée suivant la règles de réécriture donnée par le motif et la production. Retourne `:fail` si l'appariement ne réussit pas. □

EXERCICE 6.2. Définir la fonction `rewrite` qui prend en entrée une donnée et une liste de régles de réécritures et réécrit la donnée tant qu'une règle s'applique. □

6.4 Tables de hachage

Une table de hachage est une structure d'association clé-valeur similaire aux listes d'association (A-liste) mais avec une implémentation plus sophistiquée, à base de vecteurs et de fonction de hachage pour indexer les clés. Il en existe des versions différentes suivant que des entrées de la table peuvent être supprimées ou pas. Pour une présentation plus approfondie, voir par exemple [Knu73].

6.4.1 Table de hachage générale

Cette sorte de table de hachage est constituée d'un vecteur de longueur n (par exemple 256) dont chaque champ pointe sur une A-liste (éventuellement nil). C'est la technique dite de *separate chaining*.

La fonction de hachage `hash` sert à associer à chaque clé l'indice du champ de la table de hachage à utiliser :

(HASH `x`) → ***`integer`***
Retourne l'indice où doit être mis l'objet x. Le hachage est indépendant de la longueur de la table et il faut donc obtenir l'entrée exacte en prenant la valeur de hachage `modulo` la taille de la table. *La fonction `hash` était fournie par* LELISP, *elle ne l'est plus par* ILOGTALK *et* COMMON LISP *: il faut donc bricoler une fonction de hachage..*

Une table de hachage a donc des fonctions très proches de celles d'une liste d'association mais elle permet de traiter des quantités de données beaucoup plus importantes, sans être pénalisée par un accès séquentiel : il suffit que la table soit assez longue ou même qu'elle ajuste sa taille suivant son contenu, ce que nous ne ferons pas ici, pour que sous certaines conditions d'utilisation, l'accès puisse se faire en temps constant (mais constant en moyenne, pas dans le pire des cas).

(MAKE-HASH-TABLE `n`) → ***`hashtable`***
Crée une table de hachage de taille `n`.

(GETHASH `key tab`) →
Retourne la valeur associée à la clé `key` dans la table `tab`.

(PUTHASH `key tab val`) → **`val`**
Affecte la valeur `val` à la clé `key` dans la table `tab`.

(REMHASH `key tab`) → **`t / nil`**
Enlève la clé `key` de la table `tab`.

(MAP-HASH-TABLE `fn tab`) → **`nil`**
Applique la fonction `fn` à toutes les paires de la table `tab`. Deux versions, en fonction ou macro.

(CLEAR-HASH-TABLE `tab`) → **`tab`**
Enleve toutes les clés de la table `tab`.

(COPY-HASH-TABLE `tab n`) → ***`hashtable`***
Fait une copie de la table `tab` en une table de taille différente, `n`.

On peut ensuite étendre la table de hachage pour qu'elle inclut un compteur d'entrées et qu'elle se retaille lorsque ce compteur dépasse un certain seuil.

6.4.2 Table de hachage en ajout seul

Ces tables sont des vecteurs, de taille constante donc, dont le principe d'accès est le suivant : on place la nouvelle entrée à la première place libre après sa valeur de hachage. lorsque l'on arrive à la fin de la table, on revient au début. C'est la technique dite de *linear probing*.

— Définir les mêmes fonctions que dans l'exercice précédent.
— Variante : les *hash-sets* : aucune valeur n'est associée à la clé et la table se comporte comme un ensemble ;
— Malgré le titre de la section, réfléchir au moyen de procéder au retrait d'une clé.

6.5 Fonctions sur les ensembles

6.5.1 Ensembles représentés comme des listes

La structure de donnée `set` n'existe pas en tant que telle : on utilise la structure de donnée de liste. La différence entre une liste et un ensemble est que, dans un ensemble (c'est-à-dire dans une liste considérée comme un ensemble), l'ordre n'est pas significatif et les éléments sont tous 2 à 2 différents.

(ADJOIN `x e`)

(SET-EQUAL `e1 e2`)

(SUBSETP `e1 e2`)

(UNION `e1 e2`)

(INTERSECTION `e1 e2`)

(DIFFERENCE `e1 e2`)

(REMOVE-DUPLICATES `l`)

(PRODUCT `e1 e2`) $\rightarrow$

Les fonctions `union`, `intersection` et `product` sont à faire en version binaire et n-aire.

Nombreuses variantes : version naïve, avec marquage, sur des ensembles ordonnés triés (à coupler avec les fonctions de tri), avec une structure de données abstraite [Rey75] ou avec des techniques de hachage.

6.5.2 Ensembles ordonnés représentés comme des listes

Refaire les mêmes fonctions, en se basant sur le fait que la liste est ordonnée pour rendre les traitements plus efficaces.

6.5.3 Le type de données "union d'intervalles"

On se propose de définir le type de données pour une union d'intervalles et de définir la bibliothèque des fonctions nécessaires à leur manipulation.

Une *union d'intervalles* représente l'union ensembliste des ensembles représentés par chaque intervalle :

$$\bigcup_{i \in [1\ n]} [x_i\ y_i]$$

On supposera que les intervalles sont *fermés*, c'est-à-dire que les bornes appartiennent à l'intervalle : l'union de deux intervalles consécutifs (la borne inf de l'un est borne sup de l'autre) est donc un intervalle, et un intervalle dont les bornes sont identiques n'est pas vide. On a donc toujours $x_i \leq y_i$.

Une union d'intervalles sera représentée par la liste des bornes de ses intervalles :

$$(\mathtt{x_1}\ \mathtt{y_1}\ \mathtt{x_2}\ \ldots\ \mathtt{x_n}\ \mathtt{y_n})$$

Il n'y a donc pas de structure de donnée particulière pour les intervalles, qui sont un cas particulier d'union d'intervalles avec $n = 1$. Une union d'intervalles peut éventuellement être vide : cas où $n = 0$, l'union d'intervalles est `()`.

Des unions d'intervalles différentes peuvent représenter le même ensemble de points. Il faut donc les normaliser. Une union d'intervalles sera normalisée, de façon à ce que sa représentation soit unique de la façon suivante :

— 2 intervalles de l'union sont disjoints ;

— les intervalles sont ordonnés (en ordre croissant).

Dans la suite du problème, on appellera *liste d'intervalles*, une liste $(\mathtt{x_1}\ \mathtt{y_1}\ \mathtt{x_2}\ \ldots\ \mathtt{x_n}\ \mathtt{y_n}), n \geq 0$ ou chaque $(\mathtt{x_i}\ \mathtt{y_i})$ est un intervalle, et *union d'intervalles* ne sera utilisé que pour une liste d'intervalles normalisée.

Toute liste d'intervalles vérifie donc $\forall i \in [1\ n], u_i \leq v_i$. Une union d'intervalles vérifiera en plus : $\forall i \in [2\ n], v_{i-1} < u_i$.

Par chance, toutes les opérations usuelles sur les ensembles s'appliquent aux unions d'intervalles en retournant des unions d'intervalles.

Question 6.5.3.2

Il est possible de faire des unions d'intervalles sur n'importe quel type de donnée totalement ordonné (chaîne, nombre entier, réel — mais pas complexe). Dans cette première question, on supposera qu'il s'agit d'intervalles numériques (type `number`), en considérant implicitement qu'il s'agit de flottants (cf. question 2).

De façon générale, on vérifiera de façon appropriée que tous les arguments passés aux fonctions sont bien du bon type. Ecrire les fonctions suivantes :

1. le prédicat `union-interval-p` qui teste si son argument est bien du type *union d'intervalles*, tel qu'il est spécifié plus haut ;
2. le prédicat `in` qui teste si un nombre (premier argument) appartient à une union d'intervalles (second argument) ;
3. la fonction `add-interval` à deux arguments, qui rajoute un intervalle (c'est-à-dire une union d'un intervalle) à une union d'intervalles et retourne l'union d'intervalles correspondant à cette union (NB C'est la fonction principale) ;
4. la fonction `union-intervals` qui fait l'union de deux unions d'intervalles pour retourner l'union d'intervalles "union".
5. la fonction `intersect-intervals` qui fait l'intersection de deux unions d'intervalles, pour retourner l'union d'intervalles "intersection" ;
6. le prédicat `sub-intervals-p` qui teste si son premier argument est bien inclus dans le second, les deux étant des unions d'intervalles. NB. On évitera de se servir de la fonction précédente.
7. la fonction `norm-intervals` qui prend en argument une liste d'intervalles et la normalise pour retourner l'union d'intervalles correspondant.

Pour les deux dernières fonctions, et sur le modèle des fonctions `add-interval` et `union-intervals`, on commencera par la fonction analogue qui traite un intervalle unique relativement à une union d'intervalles : on passera ensuite à la fonction qui traite deux unions d'intervalles.

Question 6.5.3.3

Dans cette question, on va essayer d'étendre les fonctions précédentes à un traitement polymorphe de tous les types totalement ordonnés.

1. D'un point de vue *mathématique* le type considéré a un effet sur les traitements à effectuer sur les bornes : lequel ? On considérera d'un côté le cas des entiers, de l'autre le cas rationels, des réels ou des chaînes : qu'est ce qui est incorrect dans le cas des entiers dans les fonctions qui précèdent ?
2. D'un point de vue *informatique* y a-t-il une réelle différence de traitement entre ces divers types ? Mais en pratique ?
3. Comment faudrait-il modifier les fonctions précédentes pour qu'elles puissent traiter des intervalles de n'importe quel type ? Considérez le cas des fonctions `in` et `union-interval-p`.
4. En toute généralité, de quelle interface fonctionnelle du type a-t-on besoin ?

Réponse : il y a une différence importante suivant qu'il s'agit d'un ordre *discret* muni d'une fonction *successeur* ou d'un ordre *dense* pour lequel entre deux nombres il en existe toujours un troisième. On n'a pas traité cette différence, en se plaçant dans le cas le plus simple, celui d'un ordre dense [1].

Question 6.5.3.4

Au lieu d'implémenter les unions d'intervalles comme une liste de longueur paire $(\mathtt{x_1}\ \mathtt{y_1}\ \mathtt{x_2}\ \ldots\ \mathtt{x_n}\ \mathtt{y_n})$, on pourrait l'implémenter comme une liste d'intervalles, chaque intervalle étant une cellule dont le `car` serait la borne inférieure et le `cdr` serait la borne supérieure : $((\mathtt{x_1}\ .\ \mathtt{y_1})\ (\mathtt{x_2}\ \ldots\ (\mathtt{x_n}\ .\ \mathtt{y_n}))$.

1. refaire les deux questions précédentes avec cette nouvelle représentation.
2. quels sont les avantages et les inconvénients des deux représentations ?

1. Dont les flottants ne sont qu'une approximation : la seule réalisation effective d'un ordre dense est donnée par les rationnels, pour les langages qui en ont une implémentation effective (COMMON LISP, par exemple).

6.6 Structures de données avec des doublets

La structure de doublets qui est à la base des listes LISP permet d'implémenter diverses structures de données comme les piles, files ou les listes doublement chaînées.

6.6.1 Structure de données de file

Une file est une espèce de liste dont on peut consommer (enlever) l'élément de tête et à laquelle on peut rajouter un élément en queue. Chacune de ces opérations doit se faire en $\mathcal{O}(1)$.

Pour accéder efficacement à la queue de la liste, il faut la mémoriser. On implémentera donc une file par une liste encapsulée par une cellule, dont le `car` pointe sur la tête de la liste et le `cdr` pointe sur la dernière cellule de la liste (si la file n'est pas vide, bien entendu, sur `nil` sinon).

Question 6.6.1.5
Dessinez la représentation par doublets d'une file vide et d'une file non vide.

Question 6.6.1.6
En se servant de cette implémentation pour que tous les accès soient faits en temps constant, définir les fonctions suivantes :

1. `new-queue` qui crée une file vide ;
2. `empty-queue?` qui teste si son argument est une file vide ;
3. `first-queue` qui retourne le premier élément de la file argument, sans l'enlever ;
4. `last-queue` qui retourne le dernier élément de la file argument.

Question 6.6.1.7
Compte-tenu de l'implémentation des files, les notions de copie, partage et modification physique concernent la cellule qui encapsule aussi bien que la liste qui est encapsulée.

1. Précisez le point précédent.
2. définir `rest-queue` qui retourne le « reste » de la file argument, sous la forme d'une file (en version partage) ;
3. définir `add-to-queue` (resp. `nadd-to-queue`) qui ajoute un nouvel élément en queue de file et retourne la file, en version partage (resp. modification physique) ;
4. définir `rem-from-queue` (resp. `nrem-from-queue`) qui enlève l'élément en tête de la file et le retourne, en version partage (resp. modification physique) ; pourquoi la version partage n'a-t-elle aucun sens ?

Question 6.6.1.8
Définir les fonctions suivantes :

1. `list-to-queue` qui « convertit » une simple liste en une file ;
2. `queue-to-list` qui « convertit » une file en une simple liste ;

Question 6.6.1.9
Pour définir les fonctions suivantes de manipulation de files, 3 solutions sont envisageables :
— écrire des fonctions récursives sur des files, analogues aux fonctions correspondantes sur les listes ;
— se servir des fonctions de conversion file-liste et des fonctions correspondantes sur les listes ;
— se servir de la connaissance que l'on a de l'implémentation des files.

1. comparer ces trois techniques suivant des critères de modularité ou d'efficacité ; on choisira la plus efficace pour la suite ;
2. définir `append-queue` (resp. `nappend-queue`) qui concatène deux files en version partage (resp. modification physique) ;
3. définir `reverse-queue` (resp. `nreverse-queue`) qui inverse une file en version copie (resp. modification physique).

6.6.2 Structure de données de liste doublement chaînée

6.6.3 Structure de données de pile

Une pile est une structure de donnée caractérisée par deux opérations : `push` pour empiler un élément et `pop` pour dépiler le sommet de la pile, et par deux invariants (à interpréter avec délicatesse) : `(pop (push x pile)) = x` et `(push (pop pile) pile) = pile`.

Question 6.6.3.10

À première vue, une pile s'implémente trivialement et efficacement (en $\mathcal{O}(1)$) avec des doublets à la LISP : il suffit de considérer que n'importe quelle liste *propre* (avec un `cdr` final à `nil`) est une pile.

1. Dans le corps d'une fonction ayant comme paramètre une variable `pile` dont la valeur est censée être une pile, donner une expression LISP ayant l'effet du `push` (resp. du `pop`).
2. Est-il possible de tirer des expressions précédentes une définition purement fonctionnelle (sans effet de bord) des fonctions `push` et `pop` ? La donner ou expliquer pourquoi ce n'est pas possible.
3. Est-il possible d'en tirer une définition des fonctions `push` et `pop`, même avec des effets de bord ? La donner ou expliquer pourquoi ce n'est pas possible.
4. Toujours dans le corps d'une fonction liant une variable `pile` dont la valeur est censée être une pile, donner une définition locale (avec `labels`) de `push` et `pop` sur cette variable `pile` (les définitions locales n'ont pas de paramètre pour la pile).

Question 6.6.3.11

Comme la définition fonctionnelle des piles est difficile *avec cette implémentation*, on veut fournir une interface pseudo-fonctionnelle des piles avec des macros `popf` et `pushf` qui vont combiner les actions correspondant à l'empilement-dépilement avec l'effet de bord sur l'entité (variable ou cellule) qui pointe sur la pile.

1. Définir, le plus simplement possible, les macros `popf` et `pushf`.
2. donnez l'expansion complète de `(popf (pushf x pile))` et de `(pushf (popf pile) pile)` : les invariants des piles sont-ils respectés ? dans quel sens de l'égalité ?
3. La définition de ces macros reste valable lorsque la pile est « pointée », non pas par une variable, mais par une cellule (`car` ou `cdr`) : faire une version plus complexe permettant d'éviter une double évaluation de *<expression-complexe>* dans le cas d'un `(popf (car <expression-complexe>))`.

NB. Les macros `pushf` et `popf` font partie des bibliothèques standard de la plupart des dialectes LISP, dont ILOGTALK et COMMON LISP.

Question 6.6.3.12

Le jeu bien connu des tours de Hanoï constitue un bon exemple de piles. A l'origine, le jeu est constitué de 3 piles — 1 pleine de disques de taille croissante, les 2 autres vides — qu'on modélisera par une liste de piles (c'est-à-dire de listes) :

```
(defglobal hanoi '((1 2 3 4 5) () ()))
```

Le jeu consiste à déplacer la première pile (la source) sur la seconde (la cible) en se servant de la troisième (l'annexe) en ne recourant qu'à des `push` et `pop` et en faisant en sorte que, sur chaque pile, les disques restent toujours ordonnés. La technique la plus simple est récursive et ne nécessite pas de considérer la contrainte de la taille des disques : pour déplacer une pile de taille n de la source vers la cible, on déplace d'abord les $n-1$ disques de la source vers l'annexe, puis le n-ième disque de la source vers la cible, pour finir par remettre la pile de taille $n-1$ de l'annexe sur la cible.

1. Écrire la fonction `hanoi` avec 4 paramètres : le nombre de disques à déplacer et les 3 piles source, cible et annexe. Pour pouvoir faire des effets de bord sur la liste de piles (valeur de la variable `hanoi`) sans modifier la liaison de cette variable, on passera en argument, non pas les piles, mais les cellules pointant sur les piles :

```
(hanoi 5 (global hanoi) (cdr (global hanoi)) (cddr (global hanoi)))
```

On se servira bien sûr de `popf` et `pushf`.

2. quelle est la complexité (en temps et en cellules) de cette fonction ?

Question 6.6.3.13

Pour donner une vraie interface fonctionnelle à des piles implémentées par des doublets, il suffit d'implémenter la pile avec un doublet pointant (par son `car`, le `cdr` ne sert pas) sur une liste qui représente la véritable pile. (NB Il s'agit là de ce que l'on a fait de façon groupée pour les tours de Hanoï.) Avec cette implémentation, définir les fonctions :

1. `stackp` qui teste si son argument est une pile, et `empty-stack-p` qui teste si son argument est une pile vide ;
2. `pop` et `push` ;
3. les invariants sont-ils respectés ?

6.7 Arithmétique des polynomes

Deux représentations différentes des polynômes, toutes les deux à base de listes, seront considérées successivement.

6.7.1 Polynômes denses

La première représentation consiste à prendre la liste des coefficients du polynôme dans l'ordre des **degrés croissants** :

$3x^4 + 2x - 1$ se représente alors par `(-1 2 0 0 3)`.

On ne tient pas compte de la variable dans la représentation. Cette représentation convient bien aux polynômes *denses*, dont la plupart des coefficients ne sont pas nuls.

Question 6.7.1.14

Définir les fonctions :

1. `polyp` qui teste si son argument est un polynôme ;
2. `norm-poly` qui simplifie (normalise) son polynôme argument en enlevant les coefficients nuls inutiles ; on pourra se contenter de faire une version avec modification physique.

   NB. Plusieurs des fonctions qui suivent réclament des arguments normalisés. Noter que la liste vide est le polynome 0.
3. `degre-poly` qui retourne le degré d'un polynôme ;
4. `val-poly`, fonction à 2 arguments, qui évalue un polynôme pour une valeur de la variable ;

   NB il est déraisonnable de faire des élévations à la puissance.
5. `+poly` qui fait la somme de ses deux polynômes arguments ;
6. `scal*poly` qui multiplie un polynôme par un scalaire (entier) ;
7. `xn*poly` qui multiplie un polynôme par une puissance n de la variable.
8. `-poly`, fonction à 1 ou 2 arguments qui retourne l'opposé de son argument ou la différence de ses arguments.
9. `*poly` qui fait le produit de 2 polynômes ;

   NB On se servira avec profit des fonctions précédentes.
10. `deriv-poly` qui calcule la dérivée (par rapport à la variable implicite) de son polynôme argument.

    Rappel : la dérivée de $\sum_{i=0}^{n} a_i x^i$ est $\sum_{i=1}^{n} i a_i x^{i-1}$
11. `/poly` qui divise 2 polynômes et retourne la paire constituée du quotient et du reste ;

    NB. Plus difficile, à garder pour la fin.

6.7.2 Polynômes peu denses

Lorsque la plupart des coefficients du polynôme sont nuls, comme dans $x^{100} - 1$, il est déraisonnable d'utiliser la représentation précédente, et on préfèrera une représentation par liste d'association degré-coefficient, par exemple `((100 . 1) (0 . -1))` pour l'exemple précédent.

Dans cette représentation, chaque degré ne peut apparaître qu'une fois, et on choisira une représentation normalisée avec des **degrés décroissants**.

Question 6.7.2.15

Définir toutes les fonctions de la question 6.7.1.14 pour cette nouvelle représentation. Pour différencier les noms des fonctions, ont suffixera les nouvelles par 2.

Pour la normalisation (fonction `norm-poly2`) on sommera les coefficients des termes de même degré, on enlèvera les termes de coefficient nul et on ordonnera les termes suivant les degrés croissants.

NB. On ne se servira dans la question 6.7.2.15 ni des fonctions de conversion de la question 6.7.2.16, ni des fonctions de la question 6.7.1.14.

Question 6.7.2.16

Faire les fonctions de conversion entre les deux représentations, `poly1->2` et `poly2->1`.

6.8 Dérivation et simplification

Il s'agit d'écrire les fonctions de dérivation et de simplification symbolique, en utilisant la technique de *programmation dirigée par les données* vue en cours. On essaiera de suivre les suggestions suivantes :

- Pour attacher à chaque opérateur les fonctions de dérivation et de simplification qui lui sont spécifiques, on définira les macros idoines.
- On considérera que les opérateurs arithmétiques $+$ et $\times$ sont d'arité variable ;
- On cherchera à traiter la dérivée de la composition de fonctions de façon générale ;
- Pour la simplification, on traitera les éléments neutres, les éléments absorbants (multiplication par 0), et les inverses, ainsi que les identités remarquables des fonctions.
- Pour la trigonométrie, on pourra introduire la constante `pi`.
- Lorsqu'une expression ne contient pas de variable, simplifier signifie calculer.
- On traitera les 4 opérateurs arithmétiques, ainsi que toutes les fonctions usuelles : trigonométrie, trigonométrie hyperbolique, logarithme et exponentielle.

Au total, l'interface fonctionnelle à réaliser est la suivante :

(DERIVATION `expr var`) $\rightarrow$ ***`expression`***

Dérive l'expression `expr` (en notation polonaise préfixée) relativement à la variable `var`. Simplifie le résultat.

(SIMPLIFY `expr`) $\rightarrow$ ***`expression`***

Simplifie l'expression `expr`.

(*DEFDERIV* `op params` $\mathbf{expr}_1 \ldots \mathbf{expr}_n$) $\rightarrow$

Cette macro définit la fonction de dérivation associée à l'opérateur `op`.

(*DEFSIMPLIFY* `op params` $\mathbf{expr}_1 \ldots \mathbf{expr}_n$) $\rightarrow$

Cette macro définit la fonction de simplification associée à l'opérateur `op`.

(GETDERIV `op`) $\rightarrow$ ***`fonction`***

Retourne la fonction de dérivation associée à l'opérateur `op`.

(GETSIMPLIFY `op`) $\rightarrow$ ***`fonction`***

Retourne la fonction de simplification associée à l'opérateur `op`.

Des fonctions auxiliaires peuvent bien entendu être introduites.

Pour la simplification, on peut aussi utiliser les règles de réécriture (Section 6.3.5).

7

Glossaire

arbre : cf. **liste**.

atome : ce qui ne peut pas se couper, au sens de `car` et de `cdr`, donc tout ce qui n'est pas une **cellule**. La liste vide est un atome.

booléen : comme il n'y a pas de type booléen en LISP, ce n'est qu'affaire d'interprétation. Voir **vrai** et **faux**.

cellule (ou *cons*) : la brique de base de construction des **listes** ; c'est le contraire d'un **atome**.

chirurgie : voir **modification physique**.

continuation : c'est une fonction passée en argument d'une autre fonction au résultat de laquelle la continuation est appliquée.

copie : se dit d'une fonction sur les listes qui produit une liste dont les cellules sont fraîchement allouées et n'appartiennent pas aux listes arguments. La copie reste **fonctionnelle**.

doublet : synonyme de **cellule**.

effet de bord : dans le sens le plus large, ce qui modifie l'état de la mémoire. Il est clair que tout calcul modifie au moins l'état de la pile, ce n'est donc pas très discriminant. Dans les langages fonctionnels, un effet de bord se définit comme une modification des **liaisons** ou des pointeurs `car` et `cdr` des **cellules**. Les effets de bord se font par `setf`. L'allocation d'une nouvelle **cellule** (par `cons`) n'est pas un effet de bord (bien que l'état de la mémoire soit modifié).

égalité : on peut distinguer plusieurs sortes d'égalité : l'égalité *numérique*, modulo des conversions de type ; l'égalité *physique*, qui correspond à l'*identité* d'objets (même adresse) ; l'égalité *logique* qui correspond à une identité de type et à une identité de la forme imprimée. La seconde implique les deux autres. L'égalité logique de deux nombres implique leur égalité numérique, mais pas forcément leur égalité physique.

COMMON LISP propose 3 fonctions générales d'égalité — du physique au logique : `eq`, `eql` et `equal` — et une fonction d'égalité par type de donnée — `=`, `string-equal`, etc.

environnement : ensemble de **liaisons**. COMMON LISP distingue l'environnement lexical et l'environnement dynamique.

environnement courant : l'environnement dans lequel l'**évaluateur** cherche la **valeur** associée a une **variable**.

évaluateur : c'est le processus qui évalue les **expressions** LISP et qui est assimilable à la fonction `eval`. L'intérêt de LISP est que cette fonction peut s'exprimer en LISP, ce qui permet de définir des *méta-évaluateurs*.

expression : la base des programmes LISP : une expression *évaluable* est soit un **atome** (constante ou **variable**), soit une **liste** plate, propre et non vide, dont le `car` est une **fonction**, une **forme spéciale** ou une **macro**. Toute expression est une **valeur** mais toute valeur n'est pas une expression *évaluable*.

faux : interprétation booléenne de la liste vide, ou `nil`.

fermeture (ou *closure*) : désigne une *fonction locale* en ce qu'elle capture l'environnement lexical courant de sa définition : la fermeture peut être passée en argument à une autre fonction qui l'activera ; l'accès à l'environnement capturé est alors possible — aussi bien en écriture qu'en lecture — en dehors de l'espace lexical de cet environnement, mais aussi en dehors de la durée de vie « normale » de l'environnement.

fonction : l'abstraction procédurale à la base des langages fonctionnels : le passage des arguments se fait par valeur, la fonction retourne une valeur. Enfin, une fonction est une valeur.

fonction globale : définie par `defun` et connue dans tout l'espace du programme.

fonction locale : fonction anonyme (λ-expression ou `let`) ou définie par `labels` ou `fletn` et connue uniquement dans le corps du `labels` (ou `fletn`).

fonctionnel : style de programmation par **fonctions**, sans **effets de bord**.

fonctionnelle : une **fonction** qui prend en argument et applique des **fonctions**. Exemple : `apply`, les fonctions de *mapping*.

forme spéciale : une « fausse fonction » en position fonctionnelle dans les **expressions** mais pour lequel l'évaluateur n'évalue pas forcément les arguments. Exemple : `if` ou `quote`. On ne peut pas définir de formes spéciales.

forme syntaxique : cf. **forme spéciale**.

garbage collector : le mécanisme de ramassage des **cellules** non utilisées.

keyword : cf. **mot-clé**.

lambda ou λ : constructeur issu du λ-calcul qui introduit une variable et crée une fonction, dite λ-fonction, dont l'application est dite une λ-expression.

liaison : le résultat de l'association d'une **variable** et d'une **valeur**, lors de l'**application** d'une **fonction** à des arguments. Une liaison se caractérise dans l'espace, par sa *portée* c'est-à-dire l'ensemble du programme où elle est connue, et, dans le temps, par sa durée de vie. Un ensemble de liaisons forme un **environnement**.

liaison lexicale : dans la liaison lexicale, l'application d'une **fonction globale** se fait dans un environnement vide et l'application d'une **fonction locale** se fait dans l'environnement lexical courant. La portée de la liaison lexicale est limitée mais son extension est indéfinie.

liaison dynamique : dans la liaison dynamique, l'application d'une **fonction globale** se fait dans l'environnement dynamique courant. La portée d'une liaison dynamique est illimitée, mais son extension réduite à la durée d'activation de la fonction qui introduit la variable.

Ce n'est possible en COMMON LISP qu'avec des variables déclarées dynamiques avec `defvar`.

liaison globale : la liaison globale est une liaison établie au *toplevel* de l'interprète. Ce n'est possible en COMMON LISP qu'avec `defvar` et `defconstant`.

liste : tout ce qui s'écrit entre (...). Une liste est soit *la* liste vide `( )` ou `nil`, soit une liste non vide, c'est-à-dire une **cellule**. Le terme de *liste* est ambigu : ce peut être une liste *plate* ou une liste *récursive*, c'est-à-dire un *arbre n-aire*. On peut les voir aussi comme l'*arbre binaire* des cellules. Enfin, une liste *plate* peut être *propre*, au sens où son dernier `cdr` est `nil`.

Une liste plate a des éléments (ce qui est pointé par ses `car` successifs). Un arbre a des feuilles.

liste d'association : structure de donnée consistant en une **liste** de **paire** clés-valeurs, qui permet de constituer un dictionnaire où des valeurs sont associées à des clés.

macro : une « fausse fonction » pour lesquelles l'évaluateur évalue deux fois les expressions : la première fois, la macro est appliquée (comme une fonction normale) aux arguments non évalués ; le résultat doit être une expression LISP qui est à nouveau évaluée.

mapping : procédé qui consiste à appliquer une **fonction** aux éléments successifs d'une liste.

modification physique (ou **chirurgie**) : se dit d'une fonction sur les listes qui réutilise les listes arguments, par *effets de bord*, en détruisant les arguments eux-mêmes — on parle aussi de fonctions destructrices.

mot-clé : symbole constant, auto-valué (il est inutile de le `quote` r), utilisé dans les formes syntaxiques. En COMMON LISP, un mot-clé est n'importe quel **symbole** commençant par un " :".

notation pointée : désigne la syntaxe LISP dans laquelle une liste s'écrit en reproduisant l'arbre binaire des cellules, chaque cellule s'écrivant comme une paire parenthésée séparée par un point.

package : un *package* définit un espace de noms qui permet d'éviter des conflits de noms entre différents modules d'un gros programme. C'est une notion indispensable à qui veut développer une grosse application en COMMON LISP. Les *packages* se notent syntaxiquement comme des préfixes des symboles, séparés par "« : » " ou "« :: » ". La spécification des *packages* est un peu confuse en COMMON LISP : pour plus de détails, voir le chapitre de [Ste90] qui leur est consacré.

paire : synonyme de **cellule**. S'utilise surtout pour parler de *paire pointée* dans la **notation pointée** des **listes**.

partage : se dit d'une fonction sur les listes qui produit une liste dont une partie des cellules appartiennent aux listes arguments qui ne sont pas elles-mêmes modifiées. Le partage reste **fonctionnel**.

prédicat : se dit d'une fonction dont la valeur est interprétée comme **booléenne**, en particulier des fonctions unaires de vérification de **type**.

récursion enveloppée : se dit d'un appel récursif dont le résultat est passé comme argument à une fonction avant que la fonction récursive appelante ne termine. Une récursion enveloppée consomme forcément de la pile d'exécution.

récursion terminale : se dit d'un appel récursif dont le résultat est retourné immédiatement par la fonction appelante. Bien compilée, une récursion terminale ne consomme pas de pile.

symbole : désigne ce qui est appelé *identificateur* dans les langages « usuels » ; la différence vient du fait qu'en LISP, un symbole est une structure de donnée accessible à l'exécution, en particulier grâce à `quote`. Un symbole peut être constant : `nil`, `t` et les **mots-clés**.

toplevel : la boucle d'interaction `read-eval-print` de l'interprète.

type : LISP est un langage « non typé », ce qui signifie qu'il est typé *dynamiquement* : les valeurs ont un type, mais pas les expressions du langage. Le programmeur doit dont vérifier le type des arguments des fonctions à l'aide des **prédicats** adéquats, en particulier en utilisant la fonction `assert`.

valeur : tout « objet » LISP bien formé et manipulable comme tel par le langage.

variable : un **symbole** (non constant) considéré comme une « variable », c'est-à-dire dans la portée d'une **liaison**.

vrai : interprétation booléenne de toute valeur qui n'est pas `nil`, en particulier le symbole constant `t`.

Index

Bibliographie

[ASS89] H. Abelson, G.J. Sussman, and J. Sussman. *Structure et interprétation des programmes informatiques*. InterÉditions, Paris, 1989.

[Cha93] J. Chailloux. *LeLisp version 15.25 : reference manual*. Ilog, 1993.

[CLI97a] CLISP manual page, 1997.

[CLI97b] Implementation notes for CLISP, 1997.

[DL13] R. Ducournau and M. Lafourcade. Compilation et interprétation des langages. Université Montpellier 2, polycopié de Master Informatique, 86 pages, 2013.

[Duc13] R. Ducournau. Programmation par Objets : les concepts fondamentaux. Université Montpellier 2, polycopié de Master Informatique, 215 pages, 2013.

[GM94] C. A. Gunter and J. C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming : Types, Semantics, and Language Design*. The MIT Press, 1994.

[ILO95] ILOG, Gentilly. ILOG TALK *reference manual, Version 3.2*, 1995.

[Knu73] D. E. Knuth. *The art of computer programming, Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[Nor92] P. Norvig. *Paradigms of artificial intelligence programming*. Morgan-Kaufmann, 1992.

[Que90] Ch. Queinnec. *Le filtrage : une application de (et pour) Lisp*. InterÉditions, Paris, 1990.

[Que94] Ch. Queinnec. *Les langages Lisp*. InterÉditions, Paris, 1994.

[Rey75] J.C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In A. Schuman, editor, *New directions in algorithmic languages*. IFIP-INRIA, 1975. (réédité dans [GM94]).

[SJ93] E. Saint-James. *La programmation applicative : de* LISP *à la machine en passant par le lambda-calcul*. Hermès, 1993.

[Ste90] G. L. Steele. *Common Lisp, the Language*. Digital Press, second edition, 1990.

[WH84] P.H. Winston and B.K.P. Horn. LISP. Addison-Wesley, 1984.

Table des matières