19. SOLUTIONS AUX EXERCICES

19.1. CHAPITRE 1

1. Détermination du type :

123	atome numérique
(EIN (SCHO%NES) BUCH)	liste
(AHA (ETONNANT . . . . !))	objet impossible à cause des 'points'
(((((1) 2) 3) 4) 5)	liste
-3Aiii	atome alphanumérique
T	atome alphanumérique
(((((ARRRGH)))))	liste

2. Détermination du nombre d'éléments :

(EIN (SCHO%NES) BUCH)	3 éléments : **EIN, (SCHO%NES), BUCH**
(((((1) 2) 3) 4) 5)	2 éléments : **((((1) 2) 3) 4), 5**
(((((ARRRGH)))))	1 élément : **((((ARRRGH))))**

(UNIX (IS A) TRADEMARK (OF) BELL LABS)
6 éléments : **UNIX, (IS A), TRADEMARK, (OF), BELL, LABS**

3. Détermination du nombre de la profondeur maximum :

$(_1$**EIN** $(_2$**SCHO%NES**$)_2$ **BUCH**$)_1$	**SCHO%NES** à la profondeur 2
$(_1(_2(_3(_4(_5$**1**$)_5$ **2**$)_4$ **3**$)_3$ **4**$)_2$ **5**$)_1$	**1** à la profondeur 5
$(_1(_2(_3(_4(_5$**ARRRGH**$)_5)_4)_3)_2)_1$	**ARRRGH** à la profondeur 5
$(_1$**UNIX** $(_2$**IS A**$)_2$ **TRADEMARK** $(_2$**OF**$)_2$ **BELL LABS**$)_1$	**IS, A** et **OF** à la profondeur 2

19.2. CHAPITRE 2

1. Les combinaisons de **CAR**, **CDR** et **CONS** :

a. **(A (B C))**
b. **(D (E F))**
c. **(B C)**
d. **(E F)**
e. **(NOBODY IS PERFECT)**
f. **((CAR A) (CONS A B))**

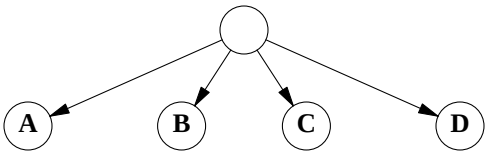
19.3. CHAPITRE 3

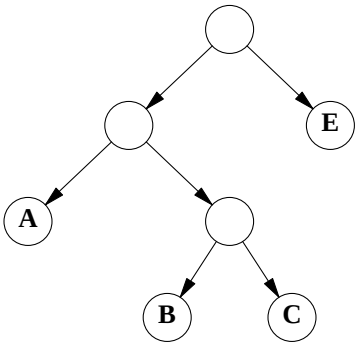
1. Combinaisons de **CAR** et **CDR** :

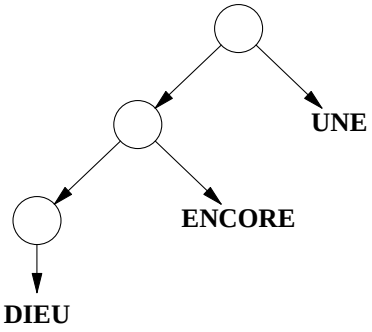
(CAR (CDR (CDR (CDR '(A B C D)))))	→	**D**
(CAR (CDR (CADR (CAR '((A (B C)) E)))))	→	**C**
(CAR (CAR (CAR '(((DIEU) ENCORE) UNE))))	→	**DIEU**

(CADR (CAR '(((DIEU) ENCORE) UNE))) → **ENCORE**

2. Traduction en forme arborescente :



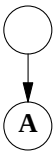




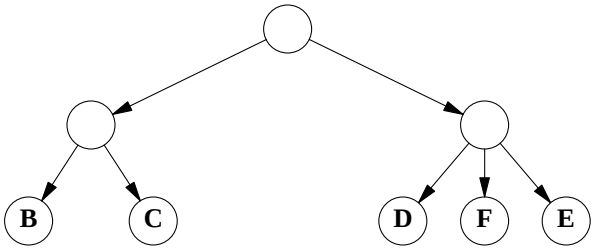
3. **SOL, ((C D)), ((HELLO) HOW ARE YOU), (JE JE JE BALBUTIE), (C EST SIMPLE)**

4. Calcul d'opérations sur des arbres :

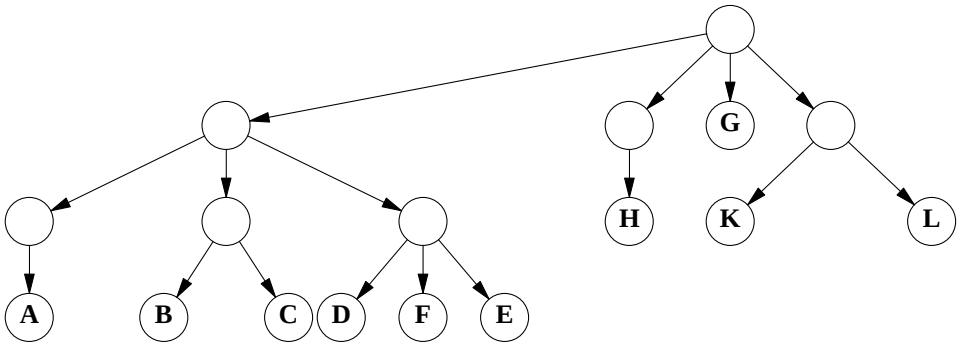
(A)



((B C) (D F E))



et finalement **(((A) (B C) (D F E)) (H) G (K L))**



5. Traduction des appels de fonction de l'exercice précédent sous forme de liste :

(CAR '((A) (B C) (D F E)))
(CDR '((A) (B C) (D F E)))
(CONS '((A) (B C) (D F E)) '((H) G (K L)))

19.4. CHAPITRE 4

1. Résultats des appels de la fonction **TRUC1** :

   ((A B C) 1 2 3)
   ((JA ES GEHT) OUI CA VA)
   ((A CAR) UNE VOITURE)

2. Combinaisons de fonctions utilisateurs :

   (MI MI MI)
   ((UN BRAVO LA TABLE) (SUR BRAVO BRAVO) CUBE ROUGE)

3. Une fonction de salutation :

   (DE BONJOUR ()
   '(BONJOUR))

4. Quatre répétitions du même élément :

```
(DE 4FOIS (ARGUMENT)
   (CONS ARGUMENT
      (CONS ARGUMENT
         (CONS ARGUMENT
            (CONS ARGUMENT ())))))
```

5. Une fonction qui inverse une liste de 3 éléments :

```
(DE REVERSE3 (ARG1 ARG2 ARG3)
   (CONS ARG3 (CONS ARG2 (CONS ARG1 NIL))))
```

19.5. CHAPITRE 5

19.5.1. chapitre 5.1.1

1. Une fonction qui teste si le deuxième élément est un nombre :

```
(DE NUMBERP-CADR (L)
    (NUMBERP (CADR L)))
```

2. Une fonction qui teste si le premier élément est une liste :

```
(DE LISTE-CAR? (L)
   (CONS 'EST (CONS (CONSP (CAR L)) ())))
```

ou :

```
(DE LISTE-CAR? (L)
   (CONS 'EST (CONS (NULL (ATOM (CAR L))) NIL)))
```

3. Telle que la fonction **NUMBERP-CADR** est définie, elle teste si le *premier* élément est un nombre. Les résultats des appels sont donc : **T**, **NIL**, **T**, **T**

19.5.2. chapitre 5.2.1

1. Une fonction qui teste si les 3 premiers éléments sont des nombres :

```
(DE 3NOMBRES (L)
   (IF (NUMBERP (CAR L))
      (IF (NUMBERP (CADR L))
         (IF (NUMBERP (CADDR L)) 'BRAVO 'PERDANT)
         'PERDANT)
      'PERDANT))
```

2. Une fonction qui inverse une liste de 1, 2 ou 3 éléments :

```
(DE REV (L)
   (IF (NULL (CDR L)) L
      (IF (NULL (CDR (CDR L))) (CONS (CADR L) (CONS (CAR L) ()))
         (CONS (CADDR L) (CONS (CADR L) (CONS (CAR L) NIL))))))
```

3. Voici les résultats des appels de la fonction **BIZARRE** :

```
(ALORS, BEN, QUOI)
(1 2 3)
(1 2 3)
(TIENS C-EST-TOI)
(NOMBRE)
(DO N EST PAS UNE LISTE, MSIEUR)
```

19.6. CHAPITRE 6

1. La nouvelle fonction **DELETE**, qui enlève les occurrences du premier argument à des profondeurs quelconques, se différencie de celle examinée par l'appel récursif sur le **CAR** de la liste (dans le cas où le premier élément est une liste) et par l'utilisation de la fonction auxiliaire **COLLE**.

   Cette fonction **COLLE** sert quand une sous-liste ne contient que des éléments à éliminer, comme dans l'appel

   (DELETE 'A '(1 (A A) 2 3))

   Le résultat devrait être : **(1 2 3)**, donc une liste où toute cette sous-liste a disparu. Si nous avions utilisé la fonction **CONS** au lieu de cette nouvelle fonction, le résultat aurait été **(1 NIL 2 3)**, ce qui est moins élégant !

```
(DE DELETE (ELE LISTE)
   (IF (NULL LISTE) ()
     (IF (ATOM LISTE)
         (IF (EQ ELE LISTE) () LISTE)
         (IF (CONSP (CAR LISTE))
            (COLLE (DELETE ELE (CAR LISTE)) (DELETE ELE (CDR LISTE)))
            (IF (EQ (CAR LISTE) ELE) (DELETE ELE (CDR LISTE))
               (CONS (CAR L) (DELETE ELE (CDR L))))))))
```

   avec la fonction **COLLE** que voici :

```
(DE COLLE (X LISTE)
      (IF (NULL X) LISTE
         (CONS X LISTE)))
```

2. Une fonction qui double tous les éléments d'une liste :

```
(DE DOUBLE (L)
  (IF (NULL L) ()
     (CONS (CAR L) (CONS (CAR L) (DOUBLE (CDR L))))))
```

3. Et voici une fonction qui double tous les atomes d'une liste :

```
(DE DOUBLE (L)
  (IF (NULL L) ()
     (IF (CONSP (CAR L))
        (CONS (DOUBLE (CAR L)) (DOUBLE (CDR L)))
         (CONS (CAR L) (CONS (CAR L) (DOUBLE (CDR L)))))))
```

4. Voici la fonction **MEMQ** qui cherche l'occurrence d'un élément à l'intérieur d'une liste :

```
(DE MEMQ (ELE LISTE) (COND
     ((NULL LISTE) NIL)
     ((EQ (CAR LISTE) ELE) LISTE)
     (T (MEMQ ELE (CDR LISTE)))))
```

5. La fonction **MEMBER** se distingue de la fonction **MEMQ** par

   • la recherche d'un élément de type quelconque : la modification nécessaire sera de remplacer l'appel de la fonction **EQ** par un appel de la fonction **EQUAL**

   • la recherche de l'élément à un niveau arbitraire : comme préalablement avec les fonctions **DELETE** et **DOUBLE**, la solution sera d'appliquer la fonction non seulement sur les **CDR**s successifs, mais également sur les **CAR**, chaque fois que le **CAR** est une liste.

Voici alors cette nouvelle fonction :

```
(DE MEMBER (ELE LISTE) (COND
    ((ATOM LISTE) NIL)
    ((EQUAL (CAR LISTE) ELE) LISTE)
    (T (MEMB1 ELE (MEMBER ELE (CAR LISTE)) (CDR LISTE)))))

(DE MEMB1 (ELE AUX LISTE)
     (IF AUX AUX (MEMBER ELE (CDR LISTE))))
```

6. Une fonction qui groupe les éléments successifs de deux listes :

```
(DE GR (L1 L2)
   (IF (NULL L1) L2
      (IF (NULL L2) L1
        (CONS
          (CONS (CAR L1) (CONS (CAR L2) ()))
          (GR (CDR L1) (CDR L2))))))
```

7. D'abord la fonction auxiliaire **FOOBAR** : elle construit une liste avec autant d'occurrences du premier argument qu'il y a d'éléments dans la liste deuxième argument. Voici quelques exemples d'appels :

```
(FOOBAR '(X Y Z) '(1 2 3))      →    ((X Y Z) (X Y Z) (X Y Z))
(FOOBAR '(1 2 3) '(X Y Z))      →    ((1 2 3) (1 2 3) (1 2 3))
(FOOBAR 1 '(A A A A A A))       →    (1 1 1 1 1 1)
```

Ensuite la fonction **FOO** (à prononcer comme *fouh*) : elle construit une liste contenant les résultats des appels de **FOOBAR** de la liste donnée en argument, et de tous les **CDR**s successifs de cette liste.

Si l'on livre à **FOO** une liste de *x* éléments, la liste résultat sera donc une liste commençant par *x* répétitions de la liste même, suivie par *x-1* occurrences du **CDR** de cette liste, suivie par *x-2* occurrences du **CDR** du **CDR** de cette liste, et ainsi de suite, jusqu'à une unique occurrence de la liste constituée du dernier élément de la liste donnée en argument.

Par exemple, si l'on appelle **FOO** comme :

```
(FOO '(X Y Z))
```

le résultat sera :

```
((X Y Z) (X Y Z) (X Y Z) (Y Z) (Y Z) (Z))
```

8. Le charme de la fonction **F** se situe dans la permutation des arguments à chaque appel récursif !

Cette fonction construit une liste où les éléments du premier argument et ceux du deuxième argument sont, en préservant l'ordre, mixés. Voici trois appels de cette fonction :

```
(F '(A B C D) '(1 2 3 4))     →    (A 1 B 2 C 3 D 4)
(F '(A B C D) '(1 2))         →    (A 1 B 2 C D)
(F '(A B C) '(1 2 3 4 5))     →    (A 1 B 2 C 3 4 5)
```

9. Cette surprenante fonction **BAR** (quatre appels récursifs !) calcule l'inverse de la liste donnée en argument. Dans l'effet, cette fonction est donc identique à la fonction **REVERSE** avec le deuxième argument égal à **NIL**.

Afin de bien comprendre comment elle fonctionne, faites-la donc tourner à la main - mais ne prenez pas une liste trop longue !

19.7. CHAPITRE 7

1. Pour la fonction **CNTH** nous supposons que l'argument numérique donné sera ≥ 1, autrement la fonction n'aura pas beaucoup de sens : quel sera le '-3ième´ élément d'une liste ?

```
(DE CNTH (NOMBRE LISTE) (COND
   ((NULL LISTE) NIL)
   ((LE N 1) (CAR LISTE))
   (T (CNTH (1- NOMBRE) (CDR LISTE)))))
```

2. L'écriture d'une fonction transformant des nombres décimaux en nombres octaux et hexadécimaux sera grandement facilitée si nous divisons cette taˆche en plusieurs sous-taˆches : d'abord nous allons écrire une fonction **DEC-OCT** qui traduit des nombres décimaux en nombres octaux, ensuite nous alllons écrire une fonction **DEC-HEX**, traduisant les nombres décimaux en nombres hexadécimaux, et, finalement, nous allons écrire la fonction principale **DEC-OCT-HEX** qui ne fait rien d'autre que construire une liste avec les résultats des appels des deux fonctions précédentes.

```
(DE DEC-OCT (N)
 (IF (> N 0)
    (APPEND (DEC-OCT (/ N 8)) (APPEND (REM N 8) NIL)) NIL))

(DE DEC-HEX (N)
 (IF (> N 0)
    (APPEND (TRAD (DEC-HEXA (/ N 16))) (APPEND (REM N 16) NIL)) NIL))
```

La fonction **TRAD** sert à traduire les valeurs entre 10 et 15 dans leurs représentations correspondantes hexadécimales (A à F).

```
(DE TRAD (L)
 (IF (NULL L) ()
     (CONS
      (CNTH (1+ (CAR L)) '(0 1 2 3 4 5 6 7 8 9 A B C D E F)))
      (TRAD (CDR L)))))

(DE DEC-OCT-HEX (NOMBRE)
  (CONS (DEC-HEX NOMBRE) (CONS (DEC-OCT NOMBRE) ())))
```

3. La fonction **OCT-DEC** fait la conversion des nombres octaux en nombres décimaux, la fonction **HEX-DEC** celle des nombres hexadécimaux en nombres décimaux.

```
(DE OCT-DEC (N)
   (LET ((N N)(RES 0))
     (IF N (SELF (CDR N) (+ (* RES 8) (CAR N))) RES)))

(DE HEX-DEC (N)
   (LET ((N N)(RES 0))
     (IF N (SELF (CDR N) (+ (* RES 16)
                    (IF (NUMBERP (CAR N)) (CAR N)
                      (LET ((M (CAR N))) (COND
                        ((EQ M 'A) 10)
                        ((EQ M 'B) 11)
                        ((EQ M 'C) 12)
                        ((EQ M 'D) 13)
                        ((EQ M 'E) 14)
                        ((EQ M 'F) 15))))))
        RES)))
```

4. Et voici la fonction **FIBONACCI** :

```
(DE FIBONACCI (N M)
  (IF (LE N 1) 1
      (+ (FIBONACCI (1- N)) (FIBONACCI (- N 2)))))
```

Ci-dessous une version récursive terminale de cette même fonction :

```
(DE FIBONACCI (N)
   (IF (LE N 1) 1 (FIB 2 '(1 1) N)))

(DE FIB (COMPTEUR LAST N)
   (IF (= COMPTEUR N) (+ (CAR LAST)(CADR LAST))
       (FIB (1+ COMPTEUR)
            (CONS (CADR LAST)(CONS (+ (CAR LAST) (CADR LAST)) NIL))
            N)))
```

5. La fonction **QUOI** 'inverse´ un nombre, c'est-à-dire, si l'on appelle **QUOI** avec un nombre de la forme $c_1c_2...c_n$, le résultat sera un nombre de la forme $c_nc_{n-1}...c_1$. Par exemple, le résultat de l'appel **(QUOI 1234)** sera le nombre **4321**.
6. Voici la fonction de Monsieur Ackermann :

```
(DE ACK (M N) (COND
   ((ZEROP M) (1+ N))
   ((ZEROP N) (ACK (1- M) 1))
   (T (ACK (1- M) (ACK M (1- N))))))
```

7. Voici deux versions d'une fonction qui calcule la longueur d'une liste. Seulement la première version est récursive terminale.

```
(DE LENGTH (L)
   (LET ((L L) (N 0))
        (IF (NULL L) N
            (SELF (CDR L) (1+ N)))))

(DE LENGTH (L)
    (IF (NULL L) 0 (1+ (LENGTH (CDR L)))))
```

8. La fonction **NBATOM** est identique à la fonction **LENGTH**, sauf qu'elle entre dans toutes les sous-listes. Là aussi, nous allons donner deux versions :

```
(DE NBATOM (L)
   (LET ((L L) (N 0))
        (IF (NULL L) N
          (IF (ATOM (CAR L)) (SELF (CDR L) (1+ N))
              (SELF (CDR L) (SELF (CAR L) N))))))

(DE NBATOM (L)
    (IF (NULL L) 0
       (IF (ATOM L) 1
           (+ (NBATOM (CAR L)) (NBATOM (CDR L))))))
```

19.8. CHAPITRE 8

1. Bien évidemment, afin de représenter ces diverses relations, il suffit de prendre les relations comme indicateurs sur les P-listes :

a. **(PUT ’GUY ’PERE ’PIERRE)**
   ou **(PUT ’PIERRE ’ENFANT ’(GUY))**
b. **(PUT ’MARIE ’PERE ’PIERRE)**
   ou **(PUT ’PIERRE ’ENFANT (CONS ’MARIE (GET ’PIERRE ENFANT)))**
c. **(PUT ’JACQUES ’PERE ’PIERRE)**
   ou **(PUT ’PIERRE ’ENFANT (CONS ’JACQUES (GET ’PIERRE ENFANT)))**
d. **(PUT ’PIERRE ’SEXE ’MASCULIN)**
e. **(PUT ’MARIE ’SEXE ’FEMININ)**
f. **(PUT ’GUY ’SEXE ’MASCULIN) (PUT ’JACQUES ’SEXE ’MASCULIN)**
g. **(PUT ’JUDY ’SEXE (GET ’MARIE ’SEXE))**
h. **(PUT ’JUDY ’AGE 22)**
i. **(PUT ’ANNE ’AGE 40)**
j. **(PUT ’SARAH ’AGE (+ (GET ’JUDY ’AGE) (GET ’ANNE ’AGE)))**

2. Ceci est notre premier exercice d’écriture d’un *vrai* programme : le premier exercice qui ne se réduit pas à l’écriture d’une seule fonction, mais qui suppose une analyse préalable du problème, la décomposition du problème en sous-problèmes, la reconnaissance de sous-problèmes communs à différentes taˆches, etc.

Nous ne donnerons pas la solution complète de cet exercice : seules des *pistes* qu’il faudrait poursuivre et développer sont énoncées :

Chaque livre sera représenté par un atome dont le nom correspond au code de ce livre. Ainsi, par exemple, **A1**, **A2**, **A3** etc seront des livres. La bibliothèque entière sera donnée par une fonction, nommons-la **BIBLIO**, qui sera définie comme :

(DE BIBLIO () ’(A1 A2 A3 . . . An))

Ainsi un appel de

(BIBLIO)

nous livrera la liste :

(A1 A2 A3 . . . An)

Ensuite, nous devons à chaque livre attacher des propriétés : son titre, son auteur et la liste des mots clefs. Ces propriétés ont leur place naturelle sur la P-liste des atomes-livres. Par exemple. pour les deux livres de Karl Kraus, nous aurons :

(PUT ’A1 ’AUTEUR ’(KARL KRAUS))
(PUT ’A1 ’TITRE ’(DIE LETZTEN TAGE DER MENSCHHEIT))
(PUT ’A2 ’AUTEUR ’(KARL KRAUS))
(PUT ’A2 ’TITRE ’(VON PEST UND PRESSE))

Ce qui reste à faire ensuite, après avoir mis à jour la bibliothèque, c’est d’écrire un ensemble de fonctions permettant de la consulter.

Construisons d’abord une fonction auxiliaire qui cherche à l’intérieur de la bibliothèque toutes les occurrences d’une certaine propriété :

```
(DE CHERCHEALL (PROP VAL) (CHERCHE PROP (BIBLIO)))

(DE CHERCHE (PROP BASE) (COND
    ((NULL BASE) NIL)
    ((EQUAL (GET (CAR BASE) PROP) VAL)
            (CONS (CAR BASE) (CHERCHE PROP (CDR BASE))))
    ((MEMBER VAL (GET (CAR BASE) PROP))
            (CONS (CAR BASE) (CHERCHE PROP (CDR BASE))))
    (T (CHERCHE PROP (CDR BASE)))))
```

Cette fonction permet, par exemple, de trouver tous les livres d'un certain auteur. Ainsi, si les livres **A1**, **A2** et **A23** sont des livres de Karl Kraus, et le livre **A456**, un livre de Karl Kraus et Ludwig Wittgenstein, l'appel

```
(CHERCHEALL 'AUTEUR '(KARL KRAUS))
```

nous donne la liste **(A1 A2 A23 A456)**.

Nous avons besoin également d'une fonction, nommons-la **FORALL**, qui nous donne pour chacun de ces livres la propriété cherchée. Là voici :

```
(DE FORALL (LISTE PROPRIETE)
    (IF (NULL LISTE) NIL
        (LET ((AUX (GET (CAR LISTE) PROPRIETE)))
          (IF AUX (CONS AUX (FORALL (CDR LISTE) PROPRIETE))
                  (FORALL (CDR LISTE) PROPRIETE)))))
```

Maintenant, nous pouvons commencer à écrire les fonctions d'interrogation de cette base de données. D'abord une fonction permettant de trouver, à partir du titre d'un livre, son auteur :

```
(DE AUTEUR (TITRE)
        (FORALL (CHERCHALL 'TITRE TITRE) 'AUTEUR))
```

Remarquez que cette fonction prévoit même le cas où deux auteurs ont écrit un livre de même titre.

Ensuite une fonction qui donne tous les livres d'un auteur :

```
(DE ALLTITRE (AUTEUR)
        (FORALL (CHERCHALL 'AUTEUR AUTEUR) 'TITRE))
```

Pour terminer, voici une fonction qui nous ramène tous les livres se référant à un sujet donné dans la liste de mots-clefs, qui se trouve sous la propriété **MOT-CLEF**.

```
(DE ALLLIVRE (MOT-CLEF)
        (FORALL (CHERCHALL 'MOT-CLEF MOT-CLEF) 'TITRE))
```

Si l'on veut connaître tous les auteurs qui ont écrit sur un sujet donné, il suffit de remplacer, dans cette dernière fonction, la propriété **TITRE** par la propriété **AUTEUR**.

Après ces quelques idées, arrêtons le développement d'un programme d'interrogation d'une base de données bibliographique (simple). A vous de le continuer !

3. Donnons d’abord les bonnes propriétés aux divers atomes :

```
(PUT ’ALBERT ’MERE ’MARTINE)     (PUT ’ALBERT ’PERE ’CLAUDE)
(PUT ’PIERRE ’MERE ’MARTINE)     (PUT ’PIERRE ’PERE ’JEAN)
(PUT ’KLAUS ’MERE ’EVE)          (PUT ’KLAUS ’PERE ’ALBERT)
(PUT ’GERARD ’MERE ’JULIE)       (PUT ’GERARD ’PERE ’PIERRE)
(PUT ’GUY ’MERE ’JULIE)          (PUT ’GUY ’PERE ’PIERRE)
(PUT ’MARIE ’MERE ’JULIE)        (PUT ’MARIE ’PERE ’PIERRE)
(PUT ’JACQUES ’MERE ’JULIE)      (PUT ’JACQUES ’PERE ’PIERRE)
(PUT ’ALAIN ’MERE ’ALICE)        (PUT ’ALAIN ’PERE ’KLAUS)
(PUT ’BRIAN ’MERE ’CATHERINE)    (PUT ’BRIAN ’PERE ’ALAIN)
(PUT ’ANNE ’MERE ’SARAH)         (PUT ’ANNE ’PERE ’GERARD)
(PUT ’CATHERINE ’MERE ’JANE)     (PUT ’CATHERINE ’PERE ’JACQUES)
(PUT ’JUDY ’MERE ’ANNE)          (PUT ’JUDY ’PERE ’PAUL)
```

et le sexe :

```
(DE PUT-SEXE (SEXE LISTE)
   (IF (NULL LISTE) NIL
      (PUT (CAR LISTE) ’SEXE SEXE)
      (PUT-SEXE SEXE (CDR LISTE))))

(PUT-SEXE ’MASCULIN
      ’(CLAUDE JEAN ALBERT PIERRE KLAUS GERARD GUY JACQUES
        ALAIN BRIAN PAUL))

(PUT-SEXE ’FEMININ
      ’(MARTINE EVE JULIE ALICE SARAH MARIE JANE ANNE
        CATHERINE JUDY))
```

Comme dans l’exercice ci-dessus, construisons une fonction qui nous calcule la liste de toutes les personnes :

```
(DE PERSONNES ()
     ’(CLAUDE MARTINE JEAN ALBERT EVE PIERRE JULIE ALICE
       KLAUS SARAH GERARD GUY MARIE JACQUES JANE ALAIN ANNE
       CATHERINE PAUL BRIAN JUDY))
```

Ensuite, il ne reste qu’à écrire des fonctions qui déduisent de cette base de données des relations familiales. Commençons par une fonction qui trouve des frères :

```
(DE FRERES (X)
   (LET ((PERE (GET X ’PERE)) (MERE (GET X ’MERE)))
      (FINDALL ’MERE MERE
         (FINDALL ’PERE PERE
            (FINDALL ’SEXE ’MASCULIN (PERSONNES))))))
```

Naturellement, la fonction **FINDALL** n’existe pas encore ! Comment suis-je tombé là-dessus ? Regardez : sachant que des frères doivent tous être de sexe masculin, qu’ils doivent tous avoir le même père et la même mère, je me suis dit qu’afin de trouver tous les frères de quelqu’un, il suffit de chercher à l’intérieur de la liste de toutes les personnes masculines connues, les personnes ayant le même père et qu’à l’intérieur de cet ensemble de personnes il faut trouver celles qui ont la même mère.

C;a fait donc trois fois que je dois chercher à l’intérieur d’une liste tous les éléments qui ont une certaine propriété identique. Au lieu de trois fois écrire la même chose, j’ai décidé d’écrire une fonction auxiliaire

qui devra faire cela. Voici la petite fonction **FINDALL** :

```
(DE FINDALL (INDICATEUR VALEUR LISTE) (COND
    ((NULL LISTE) ())
    ((EQUAL (GET (CAR LISTE) INDICATEUR) VALEUR)
        (CONS (CAR LISTE) (FINDALL INDICATEUR VALEUR (CDR LISTE))))
    (T (FINDALL INDICATEUR VALEUR (CDR LISTE)))))
```

Regardez cette fonction : elle contient deux appels récursifs. Dans ces deux appels seulement *un* argument, **LISTE**, est modifié. Les deux autres arguments restent constants. Sachant que chaque appel d'une fonction implique une nouvelle liaison des variables paramètres, nous pouvons légèrement améliorer cette fonction en sortant les arguments invariants de la boucle. Voici donc la deuxième version de la fonction **FINDALL** :

```
(DE FINDALL (INDICATEUR VALEUR LISTE)
  (LET ((LISTE LISTE)) (COND
      ((NULL LISTE) ())
      ((EQUAL (GET (CAR LISTE) INDICATEUR) VALEUR)
          (CONS (CAR LISTE) (SELF (CDR LISTE))))
      (T (SELF (CDR LISTE))))))
```

Mais continuons : une fonction qui trouve les soe;urs d'une personne ne sera pas difficile à écrire : il suffit de remplacer, dans la fonction **FRERES**, la propriété **MASCULIN** par la propriété **FEMININ**, ce qui donne :

```
(DE SOEURS (X)
  (LET ((PERE (GET X 'PERE)) (MERE (GET X 'MERE)))
    (FINDALL 'MERE MERE
      (FINDALL 'PERE PERE
        (FINDALL 'SEXE 'FEMININ (PERSONNES))))))
```

Maintenant, nous avons deux fonctions quasiment identiques : la fonction **FRERES** et la fonction **SOEURS**. Chaque fois que vous rencontrerez dans un programme des fonctions presque identiques, vous pouvez être suˆr qu'il y a des optimisations à faire en construisant des fonctions auxiliaires qui captent la similarité. Ecrivons donc une fonction auxiliaire que nous nommerons **FINDALL-TROIS**, puisqu'elle doit trouver tous les éléments qui ont trois propriétés identiques :

```
(DE FINDALL-TROIS (X PROP1 PROP2 PROP3 VALEUR3)
    (FINDALL PROP1 (GET X PROP1)
      (FINDALL PROP2 (GET X PROP2)
        (FINDALL PROP3 VALEUR3 (PERSONNES)))))
```

et les fonctions **FRERES** et **SOEURS** se simplifient en :

```
(DE FRERES (X)
  (DELETE X (FINDALL-TROIS X 'PERE 'MERE 'SEXE 'MASCULIN)))

(DE SOEURS (X)
  (DELETE X (FINDALL-TROIS X 'PERE 'MERE 'SEXE 'FEMININ)))
```

et pour avoir à la fois les frères et les soe;urs, il suffit d'écrire :

```
(DE SIBLINGS (X) (DELETE X (APPEND (FRERES X) (SOEURS X))))
```

A vous de compléter l'exercice en écrivant les fonctions nécessaires pour trouver les cousins, cousines, tantes, oncles, etc. Toutes ces fonctions devraient maintenant être faciles à écrire !

Pour conclure cet exercice, voici la fonction qui vous ramène tous les ancêtres d'une personne :

```
(DE ANCETRES (X)
   (LET ((MERE (GET X 'MERE)) (PERE (GET X 'PERE)))
        (IF (NULL MERE)
          (IF (NULL PERE) () (CONS PERE (ANCETRES PERE)))
          (IF (NULL PERE)
              (CONS MERE (ANCETRES MERE))
              (CONS MERE (CONS PERE
                         (APPEND (ANCETRES PERE)
                                 (ANCETRES MERE))))))))
```

19.9. CHAPITRE 9

1. Voici la fonction **FACTORIELLE** en tant que mémo-fonction :

```
(DE FACTORIELLE (N) (COND
    ((LE N 0) 1)
    ((GET 'FACTORIELLE N))
    (T (PUT 'FACTORIELLE N (* N (FACTORIELLE (1- N))))
       (GET 'FACTORIELLE N))))
```

2. Sachant qu'un **COND** évalue un test après l'autre, nous pouvons réécrire ce programme comme :

```
(DE FOO (X) (COND
    ((< X 0) (- 0 X))
    ((ZEROP X) 0)
    (T (* X -1))))
```

   Bien évidemment, ce programme peut encore se simplifier en :

```
(DE INVERSE (X) (- 0 X))
```

3. Voici un premier petit simplificateur algébrique :

```
(DE SIMP (X)(COND
    ((ATOM X) X)
    ((EQ (CAR X) '+) (SIMPLIFY-PLUS (SIMP (CADR X)) (SIMP (CADDR X))))
    ((EQ (CAR X) '*) (SIMPLIFY-TIMES (SIMP (CADR X)) (SIMP (CADDR X))))
    (T X)))

(DE SIMPLIFY-PLUS (X Y) (COND
    ((AND (NUMBERP X) (NUMBERP Y)) (+ X Y))
    ((ZEROP X) Y)
    ((ZEROP Y) X)
    (T (CONS '+ (CONS X (CONS Y ()))))))

(DE SIMPLIFY-TIMES (X Y) (COND
    ((AND (NUMBERP X) (NUMBERP Y)) (* X Y))
    ((OR (ZEROP X) (ZEROP Y)) 0)
    ((= X 1) Y)
    ((= Y 1) X)
    (T (CONS '* (CONS X (CONS Y ()))))))
```

   Dans ce programme nous avons utilisé deux nouvelles fonctions : **AND** et **OR**.

   La fonction **AND** est le *et* logique. Elle prend un nombre quelconque d'arguments et ramène la valeur du dernier argument, si *tous* les arguments se sont évalués à des valeurs différentes de **NIL**, et ramène **NIL** dans les autres cas.

La fonction **OR** est le *ou* logique. Elle aussi prend un nombre quelconque d'arguments et ramène la valeur du *premier* argument qui s'évalue à une valeur différente de **NIL**. Si *tous* les arguments s'évaluent à **NIL**, la valeur de la fonction **OR** est **NIL**.

Bien entendu, nous aurions pu écrire le programme sans ces deux fonctions standard nouvelles. Mais l'écriture avec **AND** et **OR** est plus lisible. Jugez vous même : voici l'écriture de **SIMPLIFY-TIMES** sans ces nouvelles fonctions :

```
(DE SIMPLIFY-TIMES (X Y) (COND
    ((ZEROP X) 0)
    ((ZEROP Y) 0)
    ((= X 1) Y)
    ((= Y 1) X)
    ((NUMBERP X)
        (IF (NUMBERP Y) (* X Y)
            (CONS '* (CONS X (CONS Y ()))))))
    (T (CONS '* (CONS X (CONS Y ()))))))
```

19.10. CHAPITRE 10

1. Si nous voulons généraliser ce petit programme de manière telle qu'il puisse conjuguer des verbes non seulement au présent, mais également en d'autres temps, la première chose à résoudre est : comment représenter les terminaisons supplémentaires ?

Actuellement, les terminaisons se touvent en plein milieu de la fonction **CONJ1**. Cacher des données à l'intérieur du code n'est que très rarement une bonne méthode. Souvent il est préférable (et plus lisible) que les données soient séparées des algorithmes. C'est cela que nous allons faire en créant deux atomes **ER** et **IR**. Sur les P-listes de ces deux atomes nous mettrons, sous des indicateurs correspondant au temps les listes de terminaisons. Ce qui donne :

```
(PUT    'IR
        'présent
          '(is is it it issons issez issent issent)
        'imparfait
          '(issais issais issait issait issions issiez issaient issaient)
        'passé-simple
          '(is is it it îmes îtes irent irent)
        'futur
          '(irai iras ira ira irons irez iront iront)
        'conditionnel
          '(irais irais irait irait irions iriez iraient iraient)
        'subjonctif-présent
          '(isse isses isse isse issions issiez issent issent)
        'subjonctif-imparfait
          '(isse isse ît ît issions issiez issent issent))
```

```
(PUT    'ER
        'présent
          '(e es e e ons ez ent ent)
        'imparfait
          '(ais ais ait ait ions iez aient aient)
        'passé-simple
          '(ai as a a aˆmes aˆtes èrent èrent)
        'futur
          '(erai eras era era erons erez eront eront)
        'conditionnel
          '(erais erais erait erait erions eriez eraient eraient)
        'subjonctif-présent
          '(e es e e ions iez ent ent)
        'subjonctif-imparfait
          '(asse asses aˆt aˆt assions assiez assent assent))
```

Maintenant nous devons également changer la fonction **CONJ1** pour enlever les listes de terminaisons qui s'y trouvent et pour adapter le programme aux changements : nous allons donc remplacer les lignes :

```
(LET ((TERMINAISONS (COND
        ((EQ TYP 'ER) '(E ES E E ONS EZ ENT ENT))
        ((EQ TYP 'RE) '(S S T T ONS EZ ENT ENT))
        (T '(IS IS IT IT ISSONS ISSEZ ISSENT ISSENT)))))
```

par l'unique ligne :

```
(LET ((TERMINAISONS (GET TYP TEMPS)))
```

en supposant que la variable **TEMPS** ait comme valeur le temps de conjugaison demandé. **TEMPS** sera donc le nouveau deuxième argument de la fonction **CONJUGUE** :

```
(DE CONJUGUE (VERBE TEMPS) . . .
```

Ensuite nous remplaçons l'appel de la fonction **PRINT**, à l'intérieur de **CONJ1**, par un appel de la fonction **VERBPRINT** que voici :

```
(DE VERBPRINT (PRONOM VERBE)
  (LET ((AUX (MEMQ TEMPS '(subjonctif-présent subjonctif-imparfait))))
    (IF AUX
       (IF (MEMQ PRONOM '(il elle ils elles)) (PRIN1 "qu'") (PRIN1 "que")))
    (PRINT PRONOM VERBE)))
```

et le tour est joué !

Néanmoins, comme vous avez suˆrement constaté, il n'y encore rien de prévu pour des conjugaisons dans des temps composés. Afin de pouvoir conjuguer dans ces temps, il faut absolument connaître les quelques conjugaisons nécessaires du verbe *avoir*. Mettons-les sur la P-liste de l'atome **AVOIR** :

```
(PUT    'AVOIR
        'passé-composé
          '(ai as a a avons avez ont ont)
        'plus-que-parfait
          '(avais avais avait avait avions aviez avaient avaient)
        'passé-antérieur
          '(eus eus eut eut euˆmes euˆtes eurent eurent)
        'futur-antérieur
          '(aurai auras aura aura aurons aurez auront auront)
        'conditionnel-passé1
          '(aurais aurais aurait aurait aurions auriez auraient auraient)
        'conditionnel-passé2
          '(eusse eusses euˆt euˆt eussions eussiez eussent eussent))
```

Ensuite construisons, comme précédemment pour le subjonctif, une fonction spéciale d'impression des temps composés :

```
(DE PR-COMPOS (PRON PARTICIPE)
  (LET ((AUX (GET 'AVOIR TEMPS))
        (PARTICIPE (CREE-MOT RACINE PARTICIPE))
        (PRON PRON))
    (IF (NULL AUX)("voila, c'est tout")
      (PRINT (IF (EQ (CAR PRON) 'je) "j'" (CAR PRON))
             (CAR AUX) PARTICIPE)
      (SELF (CDR AUX) PARTICIPE (CDR PRON)))))
```

Il ne reste qu'à changer la fonction **CONJ1** de manière à introduire une possibilité de construire des temps composés. Rappelons-nous qu'actuellement nous avons une fonction qui sait conjuguer normalement (des temps non composés) et une qui sait faire des temps composés. Donc il faut introduire quelque part un test permettant d'appeler soit l'une soit l'autre.

La fonction **CONJ1** ci-dessous résout le problème en restant à l'intérieur de **CONJ1** si l'évaluation de **(GET TYP TEMPS)** était différent de **NIL**, et de lancer la fonction **PR-COMPOS** dans le cas contraire, c'est-à-dire quand il n'y a pas, sur la P-liste de **TYP** (donc **ER** ou **IR**), une liste de terminaisons associées au temps demandé :

```
(DE CONJ1 (RACINE TYP PRONOMS)
  (LET ((TERMINAISONS (GET TYP TEMPS)))
    (LET ((PRONOMS PRONOMS)(TERMINAISONS TERMINAISONS))
      (COND
        ((NULL PRONOMS) "voila, c'est tout")
        ((ATOM TERMINAISONS)
          (PR-COMPOS PRONOMS (IF (EQ TYP 'IR) 'i 'é)))
        (T (VERBPRINT (CAR PRONOMS)
                      (CREE-MOT RACINE (CAR TERMINAISONS)))
           (SELF (CDR PRONOMS)(CDR TERMINAISONS)))))))
```

2. Cet exercice est laissé entièrement à vos soins ! Notez toutefois, qu'après l'exercice précédent, il ne devrait poser aucun problème.

3. Entre temps il doit être évident qu'un problème peut être résolu (et donc programmé) de beaucoup de façons différentes. Voici *une* manière de résoudre le problème de la tour de Hanoi avec, initialement, des disques sur chacune des aiguilles :

```
(DE HANOI (N D A I AID)
   (WHEN (> N 0)
      (IF AID (HANOI (1- N) D I A) (HANOI N I A D N))
      (PRINT 'DISQUE N 'DE D 'VERS A)
      (IF (= N AID) (HAN (1- N) I D A) (HAN (1- N) I A D))))

(DE HAN (N D A I)
   (WHEN (> N 0)
      (HAN (1- N) D I A)
      (LET ((C 3))
        (WHEN (> C 0) (PRINT 'DISQUE N 'DE D 'VERS A) (SELF (1- C))))
      (HAN (1- N) I A D)))
```

Etudiez bien cette fonction ! Elle a l'air très difficile à comprendre !

Notez que j'ai utilisé la fonction **WHEN**. Cette fonction est similaire à la fonction **IF** : le premier argument est un *test*, et si ce test s'évalue à quelque chose différent de **NIL** la suite d'instructions qui compose le reste des arguments du **WHEN** est éxécutée et la valeur de la dernière expression est celle du **WHEN**, sinon, si l'évaluation du test donne **NIL**, on sort du **WHEN** en ramenant la valeur **NIL**.

Voici deux appels de cette fonction (si vous voulez voir le résultat avec plus de disques, implémentez, améliorez et testez vous-mêmes le programme) :

```
? (HANOI 1 'depart 'arrivee 'intermediaire)
  DISQUE 1 DE intermediaire VERS arrivee
  DISQUE 1 DE depart VERS arrivee
= NIL
? (HANOI 2 'depart 'arrivee 'intermediaire)
  DISQUE 1 DE arrivee VERS depart
  DISQUE 1 DE intermediaire VERS depart
  DISQUE 2 DE intermediaire VERS arrivee
  DISQUE 1 DE depart VERS intermediaire
  DISQUE 1 DE depart VERS intermediaire
  DISQUE 1 DE depart VERS intermediaire
  DISQUE 2 DE depart VERS arrivee
  DISQUE 1 DE intermediaire VERS arrivee
  DISQUE 1 DE intermediaire VERS arrivee
  DISQUE 1 DE intermediaire VERS arrivee
= NIL
```

19.11. CHAPITRE 11

1. La fonction **TIMES**, qui multiplie une suite de nombres, peut être définie comme suit :

```
(DE TIMES L (TTIMES L))

(DE TTIMES (L)
   (IF (NULL (CDR L)) (CAR L)
      (* (CAR L) (TTIMES (CDR L)))))
```

2. Voici une fonction qui construit une *liste des valeurs* des éléments de rang pair :

```
(DE SLIST L (SSLIST (CDR L)))

(DE SSLIST (L)
   (IF (NULL L) ()
       (CONS (CAR L) (SSLIST (CDDR L)))))
```

3. Et voici la fonction qui construit une *liste des éléments* de rang pair :

```
(DF QSLIST (L) (SSLIST (CDR L)))

(DE SSLIST (L)
   (IF (NULL L) ()
       (CONS (CAR L) (SSLIST (CDDR L)))))
```

   Visiblement, les deux fonctions **SSLIST** sont identiques. La différence se situe exclusivement au niveau de la définition de la fonction **SLIST**, une NEXPR, et **QSLIST**, une FEXPR.

4. Voici la fonction **MAX** qui doit trouver le maximum dans une suite de nombres :

```
(DE MAX L (IF L (MAXIMUM (CDR L) (CAR L))))
```

   La fonction auxiliaire **MAXIMUM** prendra deux arguments : une liste de nombres et le maximum déjà trouvé. Naturellement, rien n'exclut que je considère le premier argument comme un maximum temporaire, je cherche donc dans la liste restante un nombre plus grand, et ainsi de suite. La voici :

```
(DE MAXIMUM (L N)
    (IF (NULL L) N
        (MAXIMUM (CDR L) (IF (< N (CAR L)) (CAR L) N))))
```

   La fonction **MIN** sera évidemment très similaire : l'unique différence sera qu'au lieu de faire un test **(< N (CAR L))** nous testerons si **(> N (CAR L))**. Voici donc les deux fonctions nécessaires :

```
(DE MIN L (IF L (MINIMUM (CDR L) (CAR L))))

(DE MINIMUM (L N)
    (IF (NULL L) N
        (MINIMUM (CDR L) (IF (> N (CAR L)) (CAR L) N))))
```

19.12. CHAPITRE 12

1. La nouvelle version de la fonction **TIMES** :

```
(DE TIMES L
   (IF (NULL L) 1
       (* (CAR L) (APPLY 'TIMES (CDR L)))))
```

2. La nouvelle version de la fonction **SLIST** :

```
(DE SLIST L
   (IF (NULL (CDR L)) ()
       (CONS (CADR L) (APPLY 'SLIST (CDDR L)))))
```

3. La nouvelle version de la fonction **QSLIST** :

```
(DF QSLIST (L)
   (IF (NULL (CDR L)) ()
       (CONS (CADR L) (EVAL (CONS 'QSLIST (CDDR L))))))
```

   Ici nous ne pouvons pas utiliser la fonction **APPLY** : elle n'admet pas de fonction de type FEXPR en position fonctionnelle.

4. Afin que notre petit traducteur mot à mot fonctionne au moins pour les quelques phrases exemples, nous avons besoin du vocabulaire suivant :

```
(DE CHAT () 'CAT)
(DE LE () 'THE)
(DE MANGE () 'EATS)
(DE SOURIS () 'MOUSE)
(DE A () 'HAS)
(DE VOLE () 'STOLEN)
(DE FROMAGE () 'CHEESE)
```

Ensuite nous avons besoin d'une fonction qui prend une suite quelconque de mots - ça sera donc une FEXPR - transforme chacun des mots en un appel de fonction et ramène en valeur la suite des résultats de chacune des fonctions-mots. Allons-y :

```
(DF TRADUIRE (PHRASE)
   (IF (NULL PHRASE) ()
      (CONS
         (APPLY (CAR PHRASE) ())      ; pour traduire un mot ;
         (EVAL (CONS 'TRADUIRE        ; pour traduire le reste ;
                  (CDR PHRASE))))))   ; de la phrase ;
```

19.13. CHAPITRE 13

1. Voici le macro-caractère '^' qui vous permet de charger un fichier simplement en écrivant

^nom-de-fichier

```
(DMC "^" () ['LIB (READ)])
```

Remarquez bien qu'une fois que nous avons défini un macro-caractère, nous pouvons l'utiliser dans la définition d'autres macro-caractères. Ici, nous avons, par exemple, utilisé les macro-caractères '[' et ']' définis dans ce chapitre.

2. Le problème dans l'écriture de ces deux macro-caractères est qu'ils se composent de plus d'un seul caractère. C'est donc à l'intérieur de la définition des macro-caractères '>' et '<' que nous devons tester qu'ils sont bien suivis du caractère '='. Voici les définitions des deux macro-caractères :

```
(DMC ">" ()
  (LET ((X (PEEKCH)))
       (IF (NULL (EQUAL X "=")) '> (READCH) 'GE)))

(DMC "<" ()
  (LET ((X (PEEKCH)))
       (IF (NULL (EQUAL X "=")) '< (READCH) 'LE)))
```

3. D'abord le programme écrivant les nombres pairs de 2 à **N** dans le fichier **pair.nb** :

```
(DE PAIR (N)
   (OUTPUT "pair.nb")
   (LET ((X 2))(COND
         ((<= X N) (PRINT X) (SELF (+ X 2)))
          (T (PRINT "FIN") (OUTPUT) N))))
```

Ensuite faisons la même chose pour les nombres impairs :

```
(DE IMPAIR (N)
   (OUTPUT "impair.nb")
   (LET ((X 1))(COND
        ((<= X N) (PRINT X) (SELF (+ X 2)))
        (T (PRINT "FIN") (OUTPUT) N))))
```

Ensuite, il ne reste qu'à calculer la somme :

```
(DE SOMME ()
      (INPUT "impair.nb")
      (LET ((X (READ)) (IMPAIR))
            (IF (EQUAL X "FIN") (SOMM1 (REVERSE IMPAIR))
                (SELF (READ) (CONS X IMPAIR)))))

(DE SOMM1 (IMPAIR)
      (INPUT "pair.nb")
      (LET ((X (READ)) (PAIR))
            (IF (NULL (EQUAL X "FIN")) (SELF (READ) (CONS X PAIR))
                (INPUT)
                (SOMM2 IMPAIR (REVERSE PAIR)))))

(DE SOMM2 (IMPAIR PAIR)
   (OUTPUT "somme.nb")
   (LET ((IMPAIR IMPAIR) (PAIR PAIR))
   (COND
      ((AND (NULL IMPAIR) (NULL PAIR)) (OUTPUT))
      ((NULL IMPAIR) (PRINT (CAR PAIR)) (SELF NIL (CDR PAIR)))
      ((NULL PAIR) (PRINT (CAR IMPAIR)) (SELF (CDR IMPAIR) NIL))
      (T (PRINT (+ (CAR IMPAIR) (CAR PAIR)))
           (SELF (CDR IMPAIR) (CDR PAIR))))))
```

19.14. CHAPITRE 14

1. D'après ce que nous savons, tous les appels récursifs qui ne sont pas arguments d'une autre fonction, i.e.: dont le résultat n'est pas utilisé par une autre fonction, sont des appels récursifs terminaux. Les autres sont des appels récursifs normaux. Ci-dessous, nous redonnons la fonction **MATCH** avec les appels récursifs terminaux écrits en *italique* :

```
(DE MATCH (FILTRE DONNEE) (COND
        ((ATOM FILTRE) (EQ FILTRE DONNEE))
        ((ATOM DONNEE) NIL)
        ((EQ (CAR FILTRE) '$)
             (MATCH (CDR FILTRE) (CDR DONNEE)))
        ((EQ (CAR FILTRE) '&) (COND
             ((MATCH (CDR FILTRE) DONNEE))
             (DONNEE (MATCH FILTRE (CDR DONNEE)))))
        ((MATCH (CAR FILTRE) (CAR DONNEE))
             (MATCH (CDR FILTRE) (CDR DONNEE)))))
```

2. Afin de permettre le signe '**%**' en plus, nous devons introduire une clause supplémentaire dans la fonction **MATCH**. Evidemment, cette clause doit tester si le filtre commence par une liste et si cette liste commence par ce signe '**%**' particulier :

```
((AND (CONSP (CAR FILTRE)) (EQ (CAAR FILTRE) '%)) . . . )
```

Ensuite, similaire au traitement du filtrage des *séquences*, nous allons récursivement appeler **MATCH** en testant succesivement un élément après l'autre de cette liste de candidats. Précisément : nous allons construire des nouveaux *filtres*, un filtre particulier pour chaque candidat, regarder si ce filtre correspond au premier élément de la donnée, si oui, nous avons trouver une bonne donnée et il ne reste qu'à filtrer le reste du filtre avec le reste de la donnée, si non, il faut alors tester avec le filtre constitué du candidat suivant. Seulement si aucun des filtres ainsi construits ne correspond au premier élément de la donnée nous devons signaler un échec.

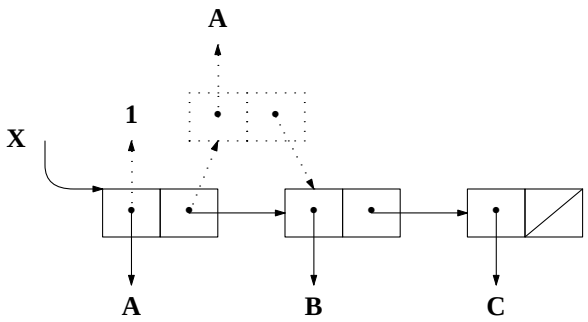
Voici la clause en entier :

```
((AND (CONSP (CAR FILTRE)) (EQ (CAAR FILTRE) '%))
      (LET ((AUX (CDR (CAR FILTRE)))) (COND
            ((NULL AUX) NIL)
            ((MATCH (CAR AUX) (CAR DONNEE))
                  (MATCH (CDR FILTRE) (CDR DONNEE)))
            (T (SELF (CDR AUX))))))
```

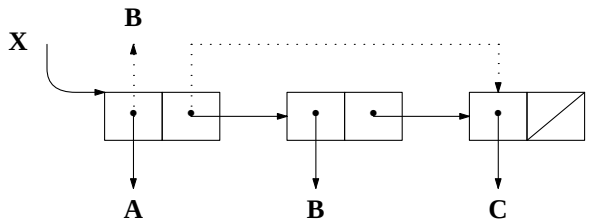
19.15. CHAPITRE 15

1. Dans les représentations graphiques suivantes, les listes originales sont dessinées avec des traits pleins, les modifications avec des traits en pointillés.

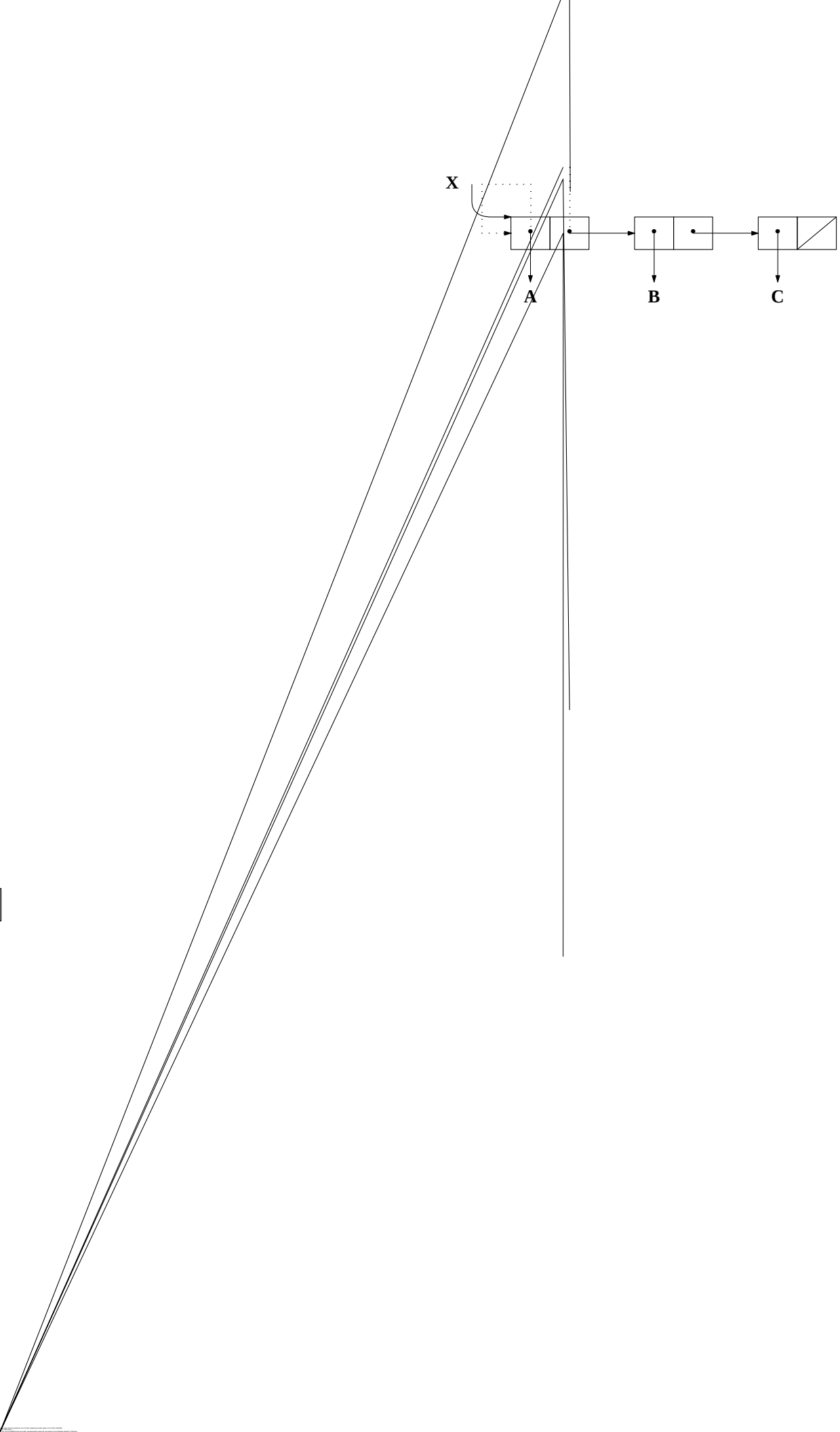
a. **(SET 'X '(A B C)) (ATTACH 1 X)**



b. **(LET ((X '(A B C))) (SMASH X))**



c. **(SETQ X '(A B C)) (RPLACA (RPLACD X X) X)**


X
A
B
C

```
(DE RPLACB (OBJET LISTE)
    (RPLACA (RPLACD OBJET (CDR LISTE)) (CAR LISTE)))
```

Cette fonction s'appelle **DISPLACE** en LE_LISP.

19.16. CHAPITRE 16

1. Evidemment, les macro-fonctions **INCR** et **DECR** sont très similaires. Les voici :

```
(DM INCR (L)
    (RPLACB L '(SETQ ,(CADR L) (1+  ,(CADR L)))))
```

et en utilisant la fonction **DMD** :

```
(DMD DECR L
    '(SETQ ,(CADR L) (1-  ,(CADR L))))
```

2. et voici la fonction **SETNTH** :

```
(DMD SETNTH L
    '(RPLACA (NTH ,(CADDR L) ,(CADR L)) ,(CADR (CDDR L))))
```

19.17. CHAPITRE 17

1. Voici la fonction **MAPS** qui applique une fonction à tous les sous-structures d'une liste :

```
(DE MAPS (L F)(COND
        ((NULL L)())
        ((ATOM L)(F L))
        (T (F L)(MAPS (CAR L) F)(MAPS (CDR L) F)))
```

2. Evidemment, l'unique différence entre la fonction **WHILE** et la fonction **UNTIL** est le sens du test : il suffit de l'inverser. Ce qui donne :

```
(DMD UNTIL CALL
        '(LET ()
            (IF ,(CADR CALL) ()
                ,@(CDDR CALL) (SELF)))))
```

et la deuxième version :

```
(DMD UNTIL CALL
        '(IF ,(CADR CALL) ()
            ,@(CDDR CALL) ,CALL))
```

3. La fonction **FACTORIELLE** avec l'itérateur **DO** :

```
(DE FACTORIELLE (N)
    (DO ((N N (1- N))
         (R 1 (* N R)))
        ((LE N 1) R)))
```

4. Et finalement la macro-fonction **DO** en utilisant **WHILE** :

```
(DM DO (CALL)
    (LET ((INIT (CADR CALL)) (STOPTEST (CAAR (CDDR CALL)))
        (RETURN-VALUE (CDAR (CDDR (CALL))))
        (CORPS (CDDDR CALL)))
    (RPLACB CALL
        '((LAMBDA ,(MAPCAR INIT 'CAR)
            (WHILE (NULL ,STOPTEST)
                ,@CORPS
                ,@(MAPCAR INIT
                    (LAMBDA (X)
                        '(SETQ ,(CAR X) ,(CADDR X)))))
            ,@RETURN-VALUE)
          ,@(MAPCAR INIT 'CADR)))))
```

fonctions standard

autres exercices

bibliographie