CEYX-Version 15.II : programmer en CEYX

J.M. Hullot

INRIA

CENTRE DE ROCQUENCOURT

Rapports Techniques

N° 45

CEYX - Version 15

II : PROGRAMMER EN CEYX

Jean-Marie HULLOT

Février 1985

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France

Tél : (1) 39 63 55 11

INRIA
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex

CEYX - Version 15

II: Programmer en CEYX

Jean-Marie Hullot

Eté 1984

Résumé: *CEYX est un ensemble d'outils permettant de faciliter la tâche des programmeurs LISP pour définir et manipuler des structures arbitraires. Aux structures définies en CEYX sont associées des espaces sémantiques, organisés hiérarchiquement, dans lesquels sont rangées les fonctions de manipulation propres à ces structures. Un style de programmation orientée objets à la SmallTalk est ainsi rendu possible.*

Ce document est le manuel programmeur CEYX. Toutes les fonctions du sytème y sont documentées et de nombreux programmes sont présentés en annexe.

Abstract: *CEYX is a set of tools allowing to define and manipulate arbitrary data structures using the LISP programming language. Semantical properties are associated to CEYX structures: they are the basic actions that can be performed on such structures. Moreover, structures are arranged by families in a hierarchical manner.*

This report is the CEYX Reference Manual.

Mots Clefs: *LISP, LE_LISP.*

Vouloir tenir une coupe et l'emplir plus qu'à ras bord,
c'est peine perdue.
Vouloir manier un outil et sans cesse l'affiler,
cela ne saurait durer longtemps.

Lao Tseu

Avertissement

Ce document constitue le manuel programmeur CEYX. Si vous êtes débutant en CEYX, consultez d'abord le manuel d'initiation à CEYX, disponible comme rapport INRIA:

- *CEYX: Une Initiation*, (J.-M. Hullot).

Programmer en CEYX, c'est d'abord programmer en LISP. CEYX est écrit en LE_LISP, il serait donc bon aussi que vous ayez sous la main le manuel LE_LISP, également disponible comme rapport INRIA:

- *LE_LISP de l'INRIA, Le Manuel de Référence*, (J. Chailloux).

Enfin, vous trouverez dans les rapports suivants la documentation de processeurs spécialisés CEYX:

- *VPRINT, Le Composeur CEYX*, (G. Berry, J.-M. Hullot),
- *CXYACC et LEX-KIT*, (G. Berry, B.Serlet).

CHAPITRE 1

Les Modèles

Jean-Marie Hullot

1.1 Introduction

Les structures de données fournies par une incarnation du langage LISP telle que LE_LISP peuvent être groupées en deux familles:

- les structures atomiques: symbol, fix, float, string;
- les structures composites: cons, vector.

Le but premier de CEYX est de:

- créer de nouvelles structures de données par composition de ces structures primitives, et de leur associer un nom;
- définir automatiquement des fonctions LISP de création d'instances de ces structures,
- définir automatiquement des fonctions d'accès en lecture et en écriture aux composantes des instances.

Associer à un nom struct une définition de modèle consiste essentiellement à associer à ce nom un *modèle de structure*, c'est à dire une information telle que:

```
(defmodel struct (Cons symbol (Vector fix fix cons)))
```

pour dire que toutes les instances de ce modèle sont une cellule cons dont le car est un symbole et le cdr un vecteur de longueur 3 dont la première composante est un petit entier (fixnum), la seconde un petit entier, et la troisième une cellule cons.

Le problème qui apparait rapidement lorsqu'on utilise ainsi des structures de données bâties à partir de structures primitives est un problème d'accès aux différentes composantes des instances de ces structures. Au fur et à mesure que la complexité des structures croît, la longueur des chaînes d'accès en car/cdr/vref croît et la lisibilité décroît. Pour résoudre ce problème CEYX donne la possibilité de nommer les composantes auquelles on désire accéder lors de la définition du modèle:

```
(defmodel struct
     (Cons (Field nom symbol)
           (Vector (Field x fix)
                   (Field y fix)
                   (Field obj cons))))
```

Une fonction LISP de nom 'mkstruct' à 3 arguments x, y et obj permettant d'engendrer des instances du modèle struct dont les champs x, y et obj ont des valeurs données est définie par:

```
(defmake struct mkstruct (x y obj))
```

Les fonctions d'accès en lecture/écriture aux diverses composantes des instances sont définies par

```
(defaccess struct)
```

Ces fonctions portent les noms des divers champs du modèle mais sont internées dans l'espace de noms (package) associé au symbole struct

```
#:struct:nom
#:struct:x
#:struct:y
#:struct:obj
```

Ces fonctions prennent toutes un ou deux arguments; dans le cas d'un seul argument elles permettent de consulter la valeur du champ voulu de l'objet passé en argument; dans le cas de deux elles permettent de remplacer la valeur stockée dans le champ voulu de l'objet premier argument par la valeur passée en deuxième argument.

Exemples

```
? (setq x (mkstruct 1 2 '(a . b)))
= (() . #[1 2 (a . b)])
? (#:struct:nom x 'toto)
= (toto . #[1 2 (a . b)])
? (#:struct:nom x)
= toto
? (#:struct:x x)
= 1
? (#:struct:y x)
= 2
```

1.2 Le Langage de Description de Modèles

1.2.1 Syntaxe

Nous dirons qu'un symbole <symbol> *a une valeur de modèle* si et seulement si <symbol> a été déclaré par:

```
(defmodel <symbol> <model>)
```

Nous appelons *modèle* toute s-sexpression de la forme:

```
<model> =   <modelname>
          |<field>
          |(Predicate <symbol> <sexpr>)
          |(List <model>)
          |(Cons <model>1 <model>2)
          |(Vector <model>1 ... <model>n)
          |<modelx>

<modelname> = <symbol> ayant une valeur de modèle

<field> =  (Field <symbol> <model> <sexpr>)
         |(Field <symbol> <model>)
         |(Field <symbol>)
```

où <modelx> désigne les macromodèles qui seront définis plus loin et <sexpr> les s-expressions LISP définies par:

```
<sexpr> =  (<sexpr>1 . <sexpr>2)
         |#[<sexpr>1 ... <sexpr>n]
         |<symbol>
         |<string>
         |<fix>
         |<float>
```

Exemples

```
? (defmodel symbol   (Predicate symbolp 'nil))
= symbol
? (defmodel cons     (Cons (Field car) (Field cdr)))
= cons
? (defmodel number   (Predicate numberp 0))
= number
? (defmodel lnumber  (List number))
= lnumber
? (defmodel point    (Cons (Field x number 0) (Field y number 0)))
= point
? (defmodel segment  (Vector
                       (Field org
```

```
                    (Cons (Field xorg number 0)
                          (Field yorg number 0)))
          (Field ext
                    (Cons (Field xext number 0)
                          (Field yext number 0))))
= segment
```

1.2.2 Sémantique

1.2.2.1 Discrimination

Les modèles sont d'abord des *schémas de filtrage* de <sexpr>. Ainsi nous dirons que le modèle

```
(Predicate symbolp ())
```

filtre toutes les <sexpr> telles que

```
(symbolp <sexpr>) ≠ ()
```

et que le modèle

```
(Cons symbol (Vector cons symbol))
```

filtre toutes les <sexpr> de la forme

```
(<symbol> . #[(<sexpr> . <sexpr>) <symbol>])
```

Plus généralement, nous dirons qu'un modèle model *filtre* une s-expression sexpr ou, de manière équivalente, que sexpr *est une instance* de model si et seulement si:

model *filtre* sexpr	
model	condition
modelname	la valeur de modèle de modelname *filtre* sexpr
(Field symbol)	
(Field symbol model1 ...)	model1 *filtre* sexpr
(Predicate fun init)	(funcall fun sexpr) ≠ ()
(List model1)	sexpr = (sexpr1 ... sexprn) model1 *filtre* sexpri
(Cons model1 model2)	sexpr = (sexpr1 . sexpr2) model1 *filtre* sexpr1 model2 *filtre* sexpr2
(Vector model1 ... modeln)	sexpr = #[sexpr1 ... sexprn] modeli *filtre* sexpri

Exemples

```
symbol   filtre  car
cons     filtre  (1 . 2)
lnumber  filtre  (1 2 3 4)
point    filtre  (1 . 2)
segment  filtre  #[(0 . 0) (10 . 90)]
```

A cette relation de filtrage, nous associons une construction LISP:

(omatchq <modelname> <sexpr>)

Cette fonction, qui n'évalue pas son premier argument, renvoie une valeur différente de () si et seulement si <modelname> *filtre* <sexpr>.

Exemples

```
? (omatchq symbol 'car)
= t
? (omatchq number 'car)
= ()
```

```
? (omatchq number 1)
= 1
? (omatchq lnumber '(1 2 3 4))
= t
? (omatchq lnumber '(1 2 a 3))
= ()
? (omatchq point '(1 . 2))
= 2
? (omatchq point 'a)
= ()
? (omatchq segment '#[(0 . 0) (10 . 90)])
= t
? (omatchq segment '#[(0 . 1) (a . 0)])
= ()
? (de always-true (x) t)
= always-true
? (defmodel * (Predicate always-true ()))
= *
? (omatchq * '(1 #[1 (2 . 3)]))
= t
```

1.2.2.2 Déstructuration

Les modèles sont ensuite des *schémas de déstructuration* de <sexpr>. Nous dirons que symbol *est un champ* de model si et seulement si:

symbol *est un champ* de model	
model	condition
modelname	symbol *est un champ* de la valeur de modèle de modelname
(Field symbol1)	(eq symbol symbol1)
(Field symbol1 model1 ...)	ou (eq symbol symbol1) ou symbol *est un champ* de model1†
(Cons model1 model2)	ou symbol *est un champ* de model1† ou symbol *est un champ* de model2†
(Vector model1 ... modeln)	ou symbol *est un champ* de model1† ... ou symbol *est un champ* de modeln†

† Dans les cas où modeli est un nom de modèle modelname, on ne descend pas dans la valeur de modèle de modelname.

Exemples

car *est un champ* de cons
x *est un champ* de point
org *est un champ* de segment
xext *est un champ* de segment

Dans le cas où une s-expression <sexpr> *est instance* du modèle <model> et où le symbole <symbol> *est un champ* du modèle <model>, nous définissons la valeur du champ <symbol> de l'instance <sexpr>:

valeur d'un champ = f (model symbol sexpr)		
model	condition	valeur
modelname		f (valeur de modèle de model1 symbol sexpr)
(Field symbol1 ...)	(eq symbol symbol1)	sexpr
(Field symbol1 model1 ...)	(neq symbol symbol1)	f (model1 symbol sexpr)
(Cons model1 model2)	symbol *est un champ* de model1	f (model1 symbol (car sexpr)))
(Cons model1 model2)	symbol *est un champ* de model2	f (model2 symbol (cdr sexpr))
(Vector model1 ... modeln)	symbol *est un champ* de modeli	f (modeli symbol (vref i sexpr))

où car,cdr et vref sont les constructions LISP permettant de déstructurer les cellules

cons et les cellules vectors.

Nous donnons des constructions LISP permettant de consulter ou de modifier des valeurs de champs.

(ogetq <modelname> <fieldname> <sexpr>)

Les deux premiers arguments ne sont pas évalués, <sexpr> ***est instance*** de <modelname> et <fieldname> ***est un champ*** de <modelname>. Cette construction ramène en valeur la valeur du champ <fieldname> de l'instance <sexpr>.

(oputq <modelname> <fieldname> <sexpr> <val>)

Les deux premiers arguments ne sont pas évalués, <sexpr> ***est instance*** de <modelname> et <fieldname> ***est un champ*** de <modelname>. Cette construction met dans la valeur du champ <fieldname> de l'instance <sexpr> la s-expression <val>. Ceci est réalisé par les fonctions idoines de modification physique rplaca, rplacd et vset.

Exemples

```
? (setq p '(1 . 2))
= (1 . 2)
? (ogetq point x p)
= 1
? (oputq point y p 25)
= (1 . 25)
? (ogetq point y p)
= 25
? (setq p1 '(10 . 10))
= (10 . 10)
? (setq seg (vector p p1))
= #[(1 . 25) (10 . 10)]
? (ogetq segment org seg)
= (1 . 25)
? (ogetq segment xorg seg)
= 1
? (oputq segment xorg seg 22)
= (22 . 25)
? seg
= #[(22 . 25) (10 . 10)]
? (ogetq segment ext seg)
= (10 . 10)
```

1.2.2.3 Instantiation

Les modèles sont enfin des ***schémas de construction*** des sexpr. Nous définissons la valeur par défaut d'un modèle:

défaut (model)	
model	valeur
modelname	défaut (valeur de modèle de modelname)
(Field symbol)	()
(Field symbol model1)	défaut (model1)†
(Field symbol model1 sexpr)	(eval sexpr)
(Predicate fun init)	(eval init)
(List model1)	()
(Cons model1 model2)	(cons défaut (model1) défaut (model2))†
(Vector model1 ... modeln)	(vector défaut (model1) ... défaut (modeln))†

† sauf dans le cas où modeli est un nom de modèle, auquel cas cette valeur par défaut vaut ().

où cons et vector sont les fonctions LISP de création de cellules cons et de vectors.

Exemples

```
défaut (symbol)    =  nil
défaut (cons)      =  (())
défaut (number)    =  0
défaut (lnumber)   =  ()
défaut (point)     =  (0 . 0)
défaut (vector)    =  #[(0 . 0) . (0 . 0)]
```

Nous définissons une construction LISP permettant de créer des instances de modèles.

(omakeq <modelname> <fieldname>1 <sexpr>1 ... <fieldname>n <sexpr>n)

Ni <modelname>, ni les <fieldname>i ne sont évalués. Cette fonction ramène en valeur une instance de <modelname> qui est une copie de défaut (<modelname>) dont les champs <fieldname>i ont été mis aux valeurs <sexpr>i.

Nous définissons également une version restreinte qui évalue son argument:

(omake <modelname>)

Cette fonction ramène en valeur une instance de <modelname> qui est une copie de défaut (<modelname>).

Exemples

```
? (omakeq symbol)
= nil
? (omakeq number)
= 0
? (omakeq segment)
= #[(0 . 0) (0 . 0)]
? (omake 'segment)
= #[(0 . 0) (0 . 0)]
? (setq seg (omakeq segment xorg 10 yorg 20 xext 30 yext 40))
= #[(10 . 20) (30 . 40)]
? (ogetq segment org seg)
= (10 . 20)
? (setq seg1 (omakeq segment org (omakeq point x 0 y 0)
?                                xext 3 yext 4)))
= #[(0 . 0) (3 . 4)]
```

1.2.3 Les Macro Modèles

Nous définissons des nouvelles expressions de notre langage de modèles: les macro modèles <modelx>.

```
<modelx> =  (Alter <model>
                <fieldname>1 <model>1
                ...
                <fieldname>n <model>n)
            (Include <modelname>)
            (Extend <model>
                <fieldname>1 (Vector <model>11 ... <model>p1)
                ...
                <fieldname>n (Vector <model>1n ... <model>pn))
```

Ces modèles portent le nom de *macro* modèles car, avant de les interpréter, CEYX leur applique une expansion, c'est à dire une transformation de ces expressions dans le langage de modèles de base. Leur sémantique sera donc parfaitement définie lorsque nous aurons décrit la règle d'expansion.

Dans ce qui suit, les <fieldname>i *sont des champs* du modèle <model>.

Expansion des Macro Modèles	
Modèle	Expansion
(Alter <model> <fieldname>1 <model>1 ... <fieldname>n <model>n)	<model> où (Field <fieldname>i ...) est remplacé par <model>i
(Include <modelname>)	valeur de modèle de <modelname>
(Extend <model> <fieldname>1 (Vector <model>11 ... <model>p1) ... <fieldname>n (Vector <model>1n ... <model>qn))	<model> où (Field <fieldname>i (Vector <model>1i' ... <model>ri') ...) est remplacé par (Field <fieldname>i (Vector <model>1i' ... <model>ri' <model>1i ... <model>mi))

On notera que pour la construction Extend, l'expansion ne peut se faire que si les champs à étendre sont déjà des vectors.

On comprendra ces constructions sans difficultés sur des exemples.

Exemples

```
? (defmodel bipoint
?           (Alter point
?                  x (Field org
?                           (Alter point
?                                  x (Field xorg number 0)
?                                  y (Field yorg number 0)))
?                  y (Field ext
?                           (Alter point
?                                  x (Field xext number 0)
?                                  y (Field yext number 0)))))
= bipoint
? (setq bipoint (omakeq bipoint xorg 1 yorg 2 ext (omakeq point)))
= ((1 . 2) 0 . 0)
? (oputq bipoint xext bipoint 4)
= (4 . 0)
? (ogetq bipoint org bipoint)
= (1 . 2)
? (defmodel named-point
?           (Cons (Field name) (Field point (Include point))))
= named-point
? (omakeq named-point name 'p x 1 y 2)
= (p 1 . 2)
? (defmodel A (Field attributs (Vector)))
= A
? (omakeq A)
= #[]
? (defmodel A1
?           (Extend A
?                   attributs (Vector (Field a1))))
= A1
? (omakeq A1 a1 1)
= #[1]
? (omatchq A (omakeq A1))
= t
? (omatchq A1 (omakeq A))
= ()
? (defmodel A3
?           (Extend A1
?                   attributs (Vector (Field a2) (Field a3))))
= A3
? (setq a (omakeq A3 a1 1 a2 2 a3 3))
= #[1 2 3]
? (ogetq A3 a1 a)
```

```
= 1
? (omatchq A1 a)
= t
? (ogetq A1 a1 a)
= 1
```

1.3 Définition de Modèles

Nous définissons dans cette partie trois constructions LISP fondamentales pour le programmeur CEYX. Il s'agit des constructions:

- defmodel, qui associe à un symbole LISP une définition de modèle;
- defaccess, qui définit des fonctions LISP permettant d'accéder en lecture/écriture aux valeurs des champs des instances d'un modèle. Pour des commodités d'écriture, il est préférable d'utiliser ces fonctions que les constructions générales ogetq et oputq;
- defmake, qui définit une fonction LISP de création d'instances d'un modèle, prenant comme argument un sous-ensemble sélectionné de ses champs. Encore une fois il est préférable d'utiliser des fonctions de création ainsi définies que la construction générale omakeq.

1.3.1 La Construction defmodel

Nous avons déjà rencontré et utilisé la construction defmodel, qui est en quelque sorte la primitive d'affectation dans le langage de modèles. Nous la définissons ici en tant que construction LISP.

(defmodel <name> <model>)

Cette fonction, qui n'évalue aucun de ses arguments, associe au symbole <name> une valeur de modèle <model>.
De manière interne une certaine représentation de <model> est placée dans le champ objval de <name>.

Nous avons déjà donné de nombreux exemples d'utilisation de cette construction dans les sections précédentes.

Exemples

```
? (defmodel fiche
?           (Cons
?              (Cons (Field nom symbol)
?                    (Field prenom symbol))
?              (Cons (Field adresse string)
?                    (Field attributs (Vector)))))
= fiche
? (setq fiche-monsieur (omakeq fiche nom 'Jacquemart prenom 'Marcel))
= ((Jacquemart . Marcel) () . #[])
```

1.3.2 La Construction defaccess

(defaccess <modelname> <fieldname>1 ... <fieldname>n)

Aucun argument n'est évalué. Cette construction amène à la définition des n fonctions d'accès de nom #:<modelname>:<fieldname>i, i.e. <fieldname>i dans l'espace de noms (package) <modelname>. Ces fonctions prennent un (lecture) ou deux (écriture) arguments, elles permettent d'accéder en lecture/écriture aux valeurs des champs de l'instance de <modelname> passée en argument.
Dans le cas où n=0 (pas de nom de champ passé en argument), les fonctions d'accès à tous les champs de <modelname> sont engendrées.

Exemples

```
? fiche-monsieur
= ((Jacquemart . Marcel) () . #[])
? (defaccess fiche nom prenom adresse)
= fiche
? ; Les fonctions définies par cet appel sont:
? ;
? ; #:fiche:nom
? ; #:fiche:prenom
? ; #:fiche:adresse
? ({fiche}:nom fiche-monsieur)
= Jacquemart
? ({fiche}:prenom fiche-monsieur)
= Marcel
? ({fiche}:adresse fiche-monsieur "Sans domicile fixe")
= (Sans domicile fixe . #[])
? ({fiche}:adresse fiche-monsieur)
= Sans domicile fixe
? fiche-monsieur
= ((Jacquemart . Marcel) Sans domicile fixe . #[])
? (defmodel point (Cons (Field x number 0) (Field y number 0)))
= point
? (defaccess point)
= point
? ; Les fonctions définies par cet appel sont:
? ;
? ; #:point:x
? ; #:point:y
```

1.3.3 La Construction defmake

(defmake <modelname> <mkname> (<fieldname>1 ... <fieldname>n))
(defmake <modelname> <mkname>)

Aucun argument n'est évalué. Dans le premier cas, cette construction définit une fonction de nom <mkname> prenant n arguments <arg>i et renvoyant en valeur une instance de <modelname> dont les champs <fieldname>i ont pour valeurs <arg>i.
De même que pour la fonction de, il est possible de passer les arguments sous forme d'arbre.
Dans le second cas, aucune fonction n'est engendrée, mais la fonction de nom <mkname> est déclarée comme fonction de construction du modèle <modelname>. Cette information est utilisée par la construction describe.

Exemples

```
? (defmake fiche fiche (nom prenom))
= fiche
? (pretty fiche)
(de fiche (nom prenom) (cons (cons nom prenom) (copy '(() . #[]))))
= ()
? (setq fiche-madame (fiche 'Dugland 'Yvette))
= ((Dugland . Yvette) () . #[])
? (defmake point mkpoint (x y))
= mkpoint
? (pretty mkpoint)
(de mkpoint (x y) (cons x y))
= ()
? (mkpoint 1 2)
= (1 . 2)
? (defmodel liste-nommee
?           (Cons (Field nom symbol)
```

```
?                (Field liste (List *))))
= liste-nommee
? (defmake liste-nommee liste-nommee (nom . liste))
= liste-nommee
? (pretty liste-nommee)
(de liste-nommee (nom . liste) (cons nom liste))
= ()
? (liste-nommee 'toto 'a 'b 'c)
= (toto a b c)
? (defmake liste-nommee faire-liste-nommee liste)
= faire-liste-nommee
? (pretty faire-liste-nommee)
(de faire-liste-nommee liste (cons () liste))
= ()
? (faire-liste-nommee 1 2 3)
= (() 1 2 3)
```

1.3.4 Les Modèles Prédéfinis

Un certain nombre de modèles sont définis à l'initialisation de CEYX, nous les décrivons ci-dessous. Tout d'abord le modèle *, qui filtre toutes les s-expressions LISP

```
(defmodel * (Predicate always-true ()))
(de always-true (x) t)
```

puis les types primitifs LISP:

```
(defmodel null (Predicate null ()))
(defmodel symbol (Predicate symbolp 'nil))
(defmodel fix (Predicate fixp 0))
(defmodel float (Predicate floatp 0.))
(defmodel number (Predicate numberp 0))
(defmodel string (Predicate stringp ""))
(defmodel atom (Predicate atomp nil))
(defmodel cons (Cons (Field car) (Field cdr)))
(defmodel vector (Predicate vectorp #[]))
```

1.4 Les Espaces Sémantiques

1.4.1 Les Espaces de Noms

L'espace des symboles en LE_LISP est organisé de manière arborescente. Ainsi quand le lecteur LISP voit

```
stop
```

il s'agit pour lui du symbole de *pname* "stop" dans l'espace de nom (package) global ||. Quand il voit

```
#:compilateur:stop
```

il sagit du symbole de *pname* "stop" dans l'espace de nom appelé compilateur. Hiérarchiser ainsi l'espace des symboles permet classiquement de se préserver des conflits de noms entre deux programmes. Ainsi stop est-il le symbole stop habituel et #:compilateur:stop est-il le symbole stop du compilateur.

Nous définissons une construction permettant de déterminer si un symbole est dans l'espace de nom d'un autre symbole.

(<=p <symbol>1 <symbol>2)

Cette fonction ramène une valeur différente de (), si et seulement si si <symbol>1 posséde dans ses espaces de noms parents <symbol>2.
Cette fonction peut être définie en LISP par:

```
(de <=p (pkg1 pkg2)
    (if (eq pkg1 pkg2) t
        (if (eq pkg1 '||) nil
            (<=p (packagecell pkg1) pkg2))))
```

Exemples

```
? (<=p '#:toto:tata:titi:tutu '#:toto:tata)
= t
```

Le problème qui se pose très vite quand on veut utiliser de manière intensive la hiérarchie des packages est un problème d'écriture: il devient vite pénible d'écrire de trop longues chaînes d'accès

```
#:toto:tata:titi:tutu:tete:tyty:mafonction
```

Pour aider à résoudre ce problème, nous introduisons la notion d'abbréviation.

(defabbrev <full-name> <abbrev>)

Cette fonction qui n'évalue pas ses arguments met dans la pliste de <abbrev> le symbole <full-name>, information qui est accessible à l'utilisateur par la fonction plink. Le symbole <abbrev> est appelé une abbréviation de <full-name>.

Les abbréviations sont utilisées par le lecteur LISP en conjonction avec les accolades {} qui sont donc de ce fait des caractères réservés: {<abbrev>} est lu comme <full-name> si <abbrev> est une abbréviation de <full-name> et comme <abbrev> dans le cas contraire.

(plink <abbrev>)

L'argument <abbrev> est évalué, sa valeur doit être un symbole. Si la valeur de <abbrev> a été définie comme une abbréviation de <full-name>, ce dernier est renvoyé en valeur, sinon la valeur de <abbrev> est renvoyée.

Exemples

```
? '{toto}
= toto
? '{toto}:a
= #:toto:a
? (defabbrev #:t:o:t:o toto)
= #:t:o:t:o
? '{toto}
= #:t:o:t:o
? (plink 'toto)
= #:t:o:t:o
? '{toto}:ah:la:la
= #:t:o:t:o:ah:la:la
```

Nous étendons la construction defmodel de manière à générer une abbréviation à la demande.

(defmodel (<name> <abbrev>) <model>)

Cette construction est équivalente à:

```
(defmodel <name> <model>)
(defabbrev <name> <abbrev>)
```

Exemples

```
? (defmodel (#:toto:tata:titi titi)
```

```
?          (Cons (Field a) (Field b)))
= #:toto:tata:titi
? '{titi}
= #:toto:tata:titi
? (defmake {titi} titi (a b))
= titi
? (setq x (titi 1 2))
= (1 . 2)
? '({titi}:a x)
= (#:toto:tata:titi:a x)
? ({titi}:a x)
** eval : fonction indefinie : #:toto:tata:titi:a
   [stack 3]    (lock ...)
   [stack 2]    (tag #:system:error-tag ...)
   [stack 1]    (itsoft ...)
? (defaccess {titi})
= #:toto:tata:titi
? ({titi}:a x)
= 1
? ; une utilisation pour faire du franglais
? (defmodel (Book Livre)
?           (Vector
?              (Field chapters (Field chapitres))
?              (Field title (Field titre))))
= Book
? (defaccess Book)
= Book
? (defmake Book Book (chapters title))
= Book
? (defmake {Livre} Livre (chapitres titre))
= Livre
? (setq b (Book '(1 2 3) "Paris by Night"))
= #[(1 2 3) Paris by Night]
? ({Livre}:titre b)
= Paris by Night
? ({Livre}:chapters b)
= (1 2 3)
```

1.4.2 Les Propriétés Sémantiques des Modèles

Nous avons déja vu comment la construction defaccess produit des fonctions d'accès aux instances d'un modèle, dont les noms sont des symboles internés dans l'espace de noms canoniquement associé au modèle. Ainsi, pour le modèle fiche, les fonctions d'accès

```
#:fiche:nom
#:fiche:prenom
#:fiche:adresse
```

sont définies.

Nous généralisons cette situation en définissant les ***propriétés sémantiques*** d'un modèle comme étant l'ensemble des fonctions définies dans l'espace de noms du modèle. Ainsi définir une propriété sémantique du modèle model revient-il à définir une fonction LISP classique dans l'espace de nom du modèle.

Exemples

```
? (de #:fiche:imprime (fiche)
?     (print "Nom: " (#:fiche:nom fiche))
?     (print "Prenom: " (#:fiche:prenom fiche))
?     (when (#:fiche:adresse fiche)
?           (print "Adresse: " (#:fiche:adresse fiche)))
?     (#:fiche:nom fiche))
= #:fiche:imprime
? (#:fiche:imprime fiche-monsieur)
Nom: Jacquemart
```

```
Prenom: Marcel
Adresse: Sans domicile fixe
= Jacquemart
? (#:fiche:imprime fiche-madame)
Nom: Dugland
Prenom: Yvette
= Dugland
```

Les *propriétés sémantiques* d'un modèle constituent en quelque sorte son *mode d'emploi*: comment accèder aux champs des instances, comment imprimer ces instances, comment les éditer, etc...

Il est de bon ton en CEYX de considérer que le premier argument d'une propriété sémantique est une instance du modèle sur lequel on définit cette propriété sémantique. La raison en apparaîtra clairement lors de l'introduction de la construction send au chapitre suivant.

1.4.3 Récupération Fonctionnelle Hiérarchique

Nous avons vu plus haut que les espaces de noms LE_LISP sont organisés de manière arborescente. Ceci se traduit en CEYX par une organisation hiérarchique des espaces sémantiques associés aux modèles. Reprenons l'exemple du modèle fiche défini par

```
? (defmodel fiche
?           (Cons
?              (Cons (Field nom symbol)
?                    (Field prenom symbol))
?              (Cons (Field adresse string)
?                    (Field attributs (Vector)))))
= fiche
```

et définissons maintenant une fiche spéciale qui est celle que voudrait utiliser un cordonnier:

```
? (defmodel {fiche}:fiche-cordonnier
?           (Extend fiche
?                   attributs (Vector (Field pointure fix))))
= #:fiche:fiche-cordonnier
```

Toutes les instances de ce nouveau modèle possèdent les champs nom, prénom, adresse, attributs du modèle fiche et les fonctions d'accès définies sur fiche par la construction defaccess sont applicables aux instances du modèle #:fiche:fiche-cordonnier. Plus généralement, toutes les propriétés sémantiques de fiche sont applicables aux instances du modèle #:fiche:fiche-cordonnier. Nous dirons que ce nouveau modèle *hérite* des propriétés sémantiques du modèle fiche.

La fonction LE_LISP getfn permet de déterminer si une propriété sémantique est définie dans un espace de noms ou bien dans l'un de ses ancêtres:

(getfn <pkg> <sym> <lastpkg>)
(getfn <pkg> <sym>)

> Cette fonction cherche si le symbole de nom <sym> possède une définition de fonction dans le package <pkg> et, s'il n'en existe pas, dans ses packages pères jusqu'au package <lastpkg> exclu. Dans le cas où <lastpkg> n'est pas spécifié, on remonte jusqu'au package global || inclus. Cette construction ramène le nom de la fonction si elle existe et () autrement.

En CEYX, nous utiliserons une version spéciale de la fonction getfn:

({CEYX}:getfn <pkg> <sym>)

> Cette fonction a un comportement analogue à getfn, sinon que la recherche est effectuée dans un premier temps jusq'au package || exclu puis en cas d'échec dans le package * (le modèle * ayant été défini plus haut) et enfin dans le package ||. Cette fonction peut être décrite en LISP par:

```
(de #:CEYX:getfn (package fun)
```

```
(or (getfn package fun '||)
    (getfn '* fun)))
```

Pour faciliter l'usage de cette construction, nous définissons les constructions semcall et hfuncall.

(hfuncall <pkg> <sym> <arg>1 ... <arg>n)

Tous les arguments sont évalués, cette construction peut être définie en LISP par:

```
(de hfuncall (pkg sym . args)
    (apply ({CEYX}:getfn pkg sym) args))
```

(semcall <pkg> <sym> <arg>1 ... <arg>n)

Analogue à la précédente sinon qu'on passe par le plink de <pkg>.

```
(de semcall (pkg sym . args)
    (apply ({CEYX}:getfn (plink pkg) sym) args))
```

La récupération fonctionnelle hiérarchique est surtout utilisée en conjugaison avec les objets auto-typés définis au chapitre suivant.

1.5 Description de Modèles

Nous donnons ici des fonctions permettant de décrire interactivement un modèle, c'est à dire tant sa structure que son espace sémantique.

(mdescribe <modelname> [<prof>])

Cette fonction, qui n'évalue pas ses arguments imprime sur le terminal la description du modèle <modelname> et de tous ses sous-modèles jusqu'à la profondeur <prof>. Si l'argument optionnel <prof> n'est pas donné on ne décrit que le modèle lui-même.

(tbl-describe <modelname>)

L'argument n'est pas évalué. Une description du modèle <modelname> est imprimée sur le terminal sous la forme d'une spécification pour troff/tbl. Très utile pour les documentations quand on travaille sous Unix.

(apropos <package>)

Liste toutes les fonctions de l'espace de nom package.

(describe <object> [<modelname>])

Les arguments sont évalués et une description de <object> est imprimée sur le terminal. Dans le cas où l'argument optionnel n'est pas donné, l'objet est décrit suivant son type (voir chapitre suivant), sinon il est décrit en supposant qu'il est une instance de <modelname>.

Exemples

```
: (defmodel toto (Cons (Field a symbol) (Field b number)))
= toto
: (mdescribe toto)

        Modele: toto

Champs:
    a ~ symbol
    b ~ number

= ()
: (defaccess toto)
= toto
: (mdescribe toto)

        Modele: toto
```

```
Champs:
     a ~ symbol
     b ~ number

Proprietes Semantiques:
     a obj
     b obj

= ()
: (defmake toto mktoto (a b))
= mktoto
: (mdescribe toto)
          Modele: toto
Fonctions de Création:
     mktoto (a b)

Champs:
     a ~ symbol
     b ~ number

Proprietes Semantiques:
     a obj
     b obj

= ()
: (de {toto}:print (x)
:     (print "a: " ({toto}:a x))
:     (print "b " ({toto}:b x)))
= #:toto:print
: (mdescribe toto)
          Modele: toto
Fonctions de Création:
     mktoto (a b)

Champs:
     a ~ symbol
     b ~ number

Proprietes Semantiques:
     a obj
     b obj
     print (x)

= ()
: (defmodel ({toto}:tata tata)
:           (Alter toto a (Cons (Field a1) (Field a2))))
= #:toto:tata
: (defaccess {tata})
= #:toto:tata
: (mdescribe toto)
          Modele: toto
Fonctions de Création:
     mktoto (a b)

Champs:
     a ~ symbol
     b ~ number

Proprietes Semantiques:
     a obj
     b obj
     print (x)

Sous Modeles:
```

```
      tata

= ()
: (mdescribe {tata})
          Modele: #:toto:tata
Abreviations:
      tata

Champs:
      a ~ (Cons ...)
      a1 ~ *
      a2 ~ *
      b ~ number

Proprietes Semantiques:
      a obj
      a1 obj
      a2 obj
      b obj

= ()
: (tbl-describe toto)
```

Le Modèle: toto	
Fonction de Création	
mktoto	(a b)
Champs	
a	symbol
b	number
Propriétés Sémantiques	
a	obj
b	obj
print	(x)
Sous Modèle	
tata	

```
: (describe (mktoto 1 2) 'toto)
(1 . 2) est une instance du modele toto
Ses différents champs valent:
a: 1
b: 2
= t
```

CHAPITRE 2

Les Types

Jean-Marie Hullot

CEYX permet un style de programmation orientée objets à la SmallTalk. On peut résumer le principe de fonctionnement d'un tel langage de la manière suivante:

- à chaque type est associé un espace sémantique, c'est à dire un ensemble de fonctions de manipulations spécifiques aux instances de ce type,
- les types sont organisés de manière hiérarchique, un sous type héritant des espaces sémantiques de ses parents,
- tous les objets manipulés sont auto-typés,
- pour appliquer une fonction de manipulation à une instance, on lui envoie en message le nom de la fonction et ses arguments. L'objet, qui connait son type, n'a donc plus qu'à chercher dans l'espace sémantique associé à ce type la fonction voulue.

Le chapitre précédent a montré comment définir des espaces sémantiques hiérarchiques, nous introduisons maintenant les notions de type, d'objets auto-typés et d'envoi de message.

On définit un type en CEYX de la même manière qu'on définit un modèle, mais en utilisant la construction deftype en lieu et place de defmodel. Nous pouvons définir le type Point par:

```
? (deftype Point (Cons (Field x fix 0) (Field y fix 0)))
= Point
```

La seule et unique différence entre modèles et types est que les instances des seconds sont étiquettées par le nom du type qui les a engendrées. Cet étiquettage est réalisée en utilisant des cellules cons spéciales (tcons) qui s'imprime sous la forme #(.) au lieu de (.).

```
? (defmake Point mkpoint (x y))
= mkpoint
? (setq p (mkpoint 5 6))
= #(Point 5 . 6)
```

Le type d'un objet est soit son type LISP (symbol, cons, vector ...), s'il n'est pas une cellule étiquettée, soit le car de cette cellule étiquettée:

```
? (type 1)
= fix
? (type '(1 . 2))
= cons
? (type p)
= Point
```

Un des avantages essentiels des objets auto-typés est qu'à partir du nom du type qui les a engendrés, ils ont accès à l'espace sémantique associé à ce nom. Ainsi si on envoie à une instance de Point le *message* translate avec les arguments dx et dy, cette instance est à même de découvrir elle-même qu'il faut appliquer la fonction #:Point:translate, si toutefois elle existe. C'est l'objet de la construction send.

```
? (defaccess Point)
= Point
? (de {Point}:translate (point dx dy)
?     (send 'x point (+ dx (send 'x point)))
?     (send 'y point (+ dy (send 'y point)))
?     point)
= #:Point:translate
```

```
? (send 'translate p 3 4)
= #(Point 8 . 10)
```

2.1 Les Types LISP

Les types LISP de base correspondent aux diverses formes que peuvent avoir les s-expressions. LISP est à même de reconnaître les types:

- symbol,
- fix,
- float,
- string,
- cons,
- vector,

à l'aide des prédicats symbolp, fixp, floatp, stringp, consp et vectorp. Dans la suite nous considérons que () est de type null, reconnu par le prédicat null.

Nous avons vu au chapitre précédent comment construire des modèles (les types de base LISP ayant eux-mêmes été définis comme modèles). Par exemple nous avons défini le modèle Coord par:

```
(defmodel Coord
          (Cons (Field x fix 0) (Field y fix 0)))
```

et une fonction de création de ce modèle par:

```
(defmake Coord Coord (x y))
```

Si nous construisons maintenant une instance du modèle Coord:

```
? (setq c (Coord 3 4))
= (3 . 4)
```

Cette instance c est de type cons, et nous n'avons aucune information sur le fait que c est une instance du modèle Coord. Notre but dans ce chapitre est de donner une extension au langage de modèles permettant de définir des objets LISP portant leur marque de fabrique, c'est à dire gardant trace du modèle dont ils sont instances.

2.2 Les Types CEYX

2.2.1 Les Objets

Nous définissons les "objets" par extension du langage de s-expressions:

```
<object> =  <sexpr>
            |#(<symbol> . <object>)
```

Nous définissons une fonction de construction LISP associée:

(tcons <symbol> <sexpr>)

Cette fonction est en tout point analogue au cons, sinon que les paires pointées construites sont étiquettées, i.e. sont reconnaissables par le prédicat tconsp.

```
? (tcons 'a 'b)
= #(a . b)
? (cons 'a 'b)
= (a . b)
```

(tconsp <obj>)

Cette fonction renvoie une valeur différente de () si et seulement si <obj> est une cellule étiquettée.

```
? (tconsp (tcons 'a 'b))
= #(a . b)
? (tconsp (cons 'a 'b))
= ()
```

Dans une incarnation du langage LISP telle que LE_LISP, la construction tcons a été introduite dans le noyau du langage pour les besoins de CEYX. La construction tcons peut être implémentée de manière efficace sur ZetaLISP, en utilisant une zone d'allocation spéciale pour les cellules étiquettées. Dans d'autres sytèmes LISP (MacLISP, FranzLISP, ...), une bonne implémentation est difficile à réaliser.

2.2.2 La Contruction deftype

Parallèlement à cette extension du langage de s-expressions, nous étendons le langage de modèles en introduisant le modèle Tcons:

<model> = ... | (Tcons <symbol> <object>)

Ce nouveau modèle joue un rôle très particulier en CEYX, c'est pourquoi il n'est introduit dans le langage de modèles que de manière interne et n'est utilisable en standard que par l'intermédiaire de la construction deftype. Le modèle Tcons est utilisé pour décrire des modèles particuliers que nous appelons ***types***, dont la particularité est que les instances sont des cellules tcons, dont la partie gauche contient le nom du modèle dont l'objet est instance et dont la partie droite contient la représentation de cette instance.

Nous introduisons la construction deftype qui permet de définir des types CEYX ou modèles auto-typés.

(deftype <name> <model>)
(deftype (<name> <abbrev>) <model>)

Cette fonction, qui n'évalue aucun de ses arguments, associe au symbole <name> une valeur de modèle obtenue de la façon suivante:

```
si <model> = (Tcons <symbol> <model>1)
   alors (Tcons <name> <model>1)
   sinon (Tcons <name> <model>)
```

Comme pour defmodel, on peut donner une abbréviation <abbrev> pour le type <name>.

Exemples

```
? (deftype Art (Cons (Field x fix 0) (Field y fix 0)))
= Art
? (defmake Art Art (x y))
= Art
? (setq x (Art 1 2))
= #(Art 1 . 2)
```

La sémantique du Modèle Tcons sera parfaitement définie quand nous aurons dit comment il réagit à la construction omatchq:

(omatchq <type> <object>) ≠ () ssi (tconsp <object>)
et (<=p (tcar <object>) <type>)

c'est à dire qu'un objet <object> est instance d'un type <type> si et seulement si <object> est un tcons dont le car est un symbole inférieur au sens de la hiérarchie des packages à <type>.

Exemples

```
? (deftype fiche
?         (Cons
?            (Cons (Field nom) (Field prenom))
```

```
?              (Field caracteristiques (Vector))))
= fiche
? (defaccess fiche)
= fiche
? (defmake fiche fiche (nom prenom))
= fiche
? (deftype ({fiche}:fiche-cordonnier fiche-cordonnier)
?          (Extend fiche
?                  caracteristiques
?                     (Vector (Field adresse)
?                             (Field pointure))))
= fiche-cordonnier
? (defaccess {fiche-cordonnier} adresse pointure)
= #:fiche:fiche-cordonnier
? (defmake {fiche-cordonnier} fiche-cordonnier (nom prenom adresse pointure))
= fiche-cordonnier
? (setq x (fiche-cordonnier 'Jacquemart 'Marcel "Inconnue" 48))
= #(#:fiche:fiche-cordonnier (Jacquemart . Marcel) . #[Inconnue 48])
? (ormatchq fiche-cordonnier x)
= t
? (ormatchq fiche x)
= t
? (ormatchq fiche-cordonnier (fiche 'Dugland 'Yvette))
= ()
```

2.2.3 La Fonction type

Nous introduisons une fonction ramenant en valeur le type au sens CEYX de n'importe quel objet.

(type <object>)

Ramène en valeur le type de l'argument <object>, qui est évalué. Cette fonction peut être définie en LISP par:

```
(de type (object)
    (cond
      ((null object) 'null)
      ((tconsp object) (car object))
      ((symbolp object) 'symbol)
      ((fixp object) 'fix)
      ((floatp object) 'float)
      ((stringp object) 'string)
      ((consp object) 'cons)
      ((vectorp object) 'vector))) t
```

2.3 La Construction send

(send <msg> <obj> <arg>1 ... <arg>n)

Tous les arguments sont évalués. Cette construction recherche si le symbole <msg> existe et possède une définition de fonction dans l'espace de noms associé à (type <obj>) d'abord et dans ses parents ensuite, ceci jusqu'à l'espace de noms racine || exclu. Si la recherche aboutit on applique cette fonction à la liste d'arguments (<obj> <arg>1 ... <arg>n). Sinon on effectue la même recherche dans l'espace de noms associé au symbole * puis dans l'espace de nom racine et on continue comme précédemment en cas de réussite, sinon une erreur est déclenchée.

En LE_LISP, la construction send a été introduite dans le noyau du langage pour les besoins de CEYX. Dans d'autres sytèmes, elle peur être définie en LISP par:

```
(de send (msg . args)
```

```
    (apply (#:CEYX:getfn (type (car args)) msg) args))

(de #:CEYX:getfn (package fun)
    (or (getfn package fun '|)
        (getfn '* fun)))
```

(sendq <msg> <obj> <arg>1 ... <arg>n)

Cette construction est équivalente à la précédente, sinon qu'elle n'évalue pas son premier argument <msg>.

Exemples

```
? (deftype Point (Field coord (Include Coord)))
= Point
? (defmake Point mkpoint (x y))
= mkpoint
? (defaccess Point)
= Point
? (setq p (Point 1 2))
** eval : fonction indefinie : Point
   [stack 3]    (lock ...)
   [stack 2]    (tag #:system:error-tag ...)
   [stack 1]    (itsoft ...)
? (setq p (mkpoint 1 2))
= #(Point 1 . 2)
? (send 'x p)
= 1
? (send 'y p 10)
= (1 . 10)
? p
= #(Point 1 . 10)
? (deftype Vector (Field vect (Include Vect)))
= Vector
? (defaccess Vector)
= Vector
? (defmake Vector mkvector (xorg yorg xext yext))
= mkvector
? (setq v (mkvector 0 0 10 25))
= #(Vector (0 . 0) 10 . 25)
? (de {Vector}:translate (vector dx dy)
?     ({Vector}:xorg vector (+ dx ({Vector}:xorg vector)))
?     ({Vector}:yorg vector (+ dy ({Vector}:yorg vector)))
?     ({Vector}:xext vector (+ dx ({Vector}:xext vector)))
?     ({Vector}:yext vector (+ dy ({Vector}:yext vector)))
?     vector)
= #:Vector:translate
? (sendq translate v 5 5)
= #(Vector (5 . 5) 15 . 30)
? (de {fiche}:imprime (fiche)
?     (print "Nom: " (sendq nom fiche))
?     (print "Prenom: " (sendq prenom fiche))
?     t)
= #:fiche:imprime
? x
= #(#:fiche:fiche-cordonnier (Jacquemart . Marcel) . #[Inconnue 48])
? (sendq imprime x)
Nom: Jacquemart
Prenom: Marcel
= t
? (de {fiche-cordonnier}:imprime (fiche)
?     ({fiche}:imprime fiche)
?     (print "Adresse: " (sendq adresse fiche))
?     (print "Pointure: " (sendq pointure fiche))
?     t)
= #:fiche:fiche-cordonnier:imprime
? (sendq imprime x)
Nom: Jacquemart
```

```
Prenom: Marcel
Adresse: Inconnue
Pointure: 48
= t
```

CHAPITRE 3

Le Précompilateur

Bertrand Serlet

3.1 Objectifs

Tous les programmes CEYX sont compilables par le compilateur standard LE_LISP, mais pour obtenir une efficacité maximale lors de l'exécution il est préférable de leur faire subir un prétraitement. Ce prétraitement se fait grâce au compilateur CEYX, appelé **cxcp**, conçu comme une couche au dessus du compilateur LE_LISP. **Cxcp** permet aussi le découplage avec le compilateur LE_LISP, d'où une répartition des tâches.

Ce prétraitemant consiste actuellement:
- à compiler de façon spéciale les accès aux champs des objets CEYX;
- à expanser à la demande certaines fonctions comme s'il s'agissait de macros;
- à optimiser la compilation de certains send.

3.1.1 Compilation des accès aux champs

Contrairement aux versions précédentes de CEYX, les fonctions d'accès aux champs sont maintenant définies comme des fonctions standards (par *de*) et non plus comme des macros. L'avantage énorme de ce changement est d'autoriser l'utilisation de la construction send pour accéder aux champs. Cela permet aussi de rendre la mise au point plus facile puisque l'on peut pister ou profiler (par *trace* ou *timetrace*) les appels. Il est aussi possible de faire dynamiquement des vérifications de type, lors de la mise au point, en modifiant la définition de ces fonctions. Malheureusement ces nouvelles possibilités ont un prix: on perd un appel fonctionnel (en fait deux) et une construction de *cons* à chaque appel (car les fonctions d'accès ont un nombre variable d'arguments). Cette perte d'efficacité, supportable lors de la mise au point, est intolérable en compilé. **Cxcp** remplace tous les appels explicites à des fonctions d'accès par la chaîne d'accès elle-même. Une fois le code compilé, tout se passe donc comme dans les versions antérieures, et les programmes compilés restent aussi efficaces.

```
: (defmodel test (Record (Field a) (Field b)))
= test
: (defaccess test)
= test
: (de testfun (x) (+ (#:test:a x) (#:test:b x))))
= testfun
: (pretty #:test:a)
(de #:test:a obj (#:CEYX:system:g138 obj))

= ()
: (compile testfun () t)
testfun se compile.
(de testfun (x) (+ (#:test:a x) (#:test:b x)))

((fentry testfun subr1)
 (mov '(testfun x) a4)
 (call $cbind1)
 (mov nil a2)
 (call cons)
 (call #:test:a)
 (push a1)
```

```
  (mov (cvalq x) a1)
  (mov nil a2)
  (call cons)
  (call #:test:b)
  (mov a1 a2)
  (pop a1)
  (plus a2 a1)
  (return))
= ((2 . -12430))
: (de testfun (x) (+ (#:test:a x) (#:test:b x)))
** de : fonction redefinie : testfun
= testfun
: (cxcp testfun () t)
testfun se compile.
(de testfun (x) (+ (#:test:a x) (#:test:b x)))
((fentry testfun subr1)
 (mov '(testfun x) a4)
 (call $cbind1)
 (mov (car a1) a1)
 (mov (cvalq x) a2)
 (cdr a2)
 (plus a2 a1)
 (return))
= ()
```

3.1.2 Expansion à la demande de fonctions

Il est parfois nécessaire pour des sections de programmes critiques en temps de transformer de petites fonctions en macros, afin d'y gagner un appel fonctionnel. La méthode habituelle consiste à écrire d'abord la fonction sous forme de *de*, puis à la saupoudrer de quelques caractères bizarres (' , .. ,@ !).

```
(de test (x y) (+ x y 1)) -> (dmd test (x y) '(+ ,x ,y 1))
ou, si l'on n'aime pas la backquote:
(dmd test (x y) (list '+ x y 1))
```

Mais **attention**, la transformation en macro cache le piège de double évaluation des variables:

```
(de test (x) (+ x x 1)) -> (dmd test (x) '(+ ,x ,x 1))
est faux, par exemple sur l'appel:
(test (next1 1))
```

Dans cet exemple, une protection correcte en double évaluation peut se faire par:

```
(dmd test (x)
     (if (atomp x)
         '(+ ,x ,x 1)
         (let ((var (gensym))) '(let ((,var ,x)) (+ ,var ,var 1)))))
```

Cxcp permet d'éviter la transformation en macro à la main. L'utilisateur choisit explicitement les fonctions expansées, et précise, toujours explicitement, les variables à protéger en double évaluation.

3.1.3 Compilation des send

Send est la fonction de base pour le style de programmation orienté objet. Malheureusement cette fonction coûte cher, moins cher en LE_LISP (où elle fait partie de l'interprète) que sur d'autres dialectes LISP, mais elle est toujours plus coûteuse qu'un simple appel fonctionnel. Or il apparaît à l'usage que pour bon nombre de sémantiques, les appels à send peuvent être optimisés en un *selectq* aiguillant suivant le type de l'objet sur un appel fonctionnel direct. Pour que **cxcp** sache quels sont les types d'objets les plus fréquents, l'utilisateur fournit au compilateur une expression LISP à évaluer, en comptant (conceptuellement) lors de cette phase d'évaluation, la fréquence des apparitions. Lors de la phase de compilation **cxcp** utilise cette information pour générer des selectq optimaux.

3.2 Utilisation du pré-compilateur

3.2.1 Appel du Compilateur

(cxcp)

De même que la fonction LE_LISP compile-all-in-core, cette fonction compile toutes les fonctions de l'oblist en leur faisant préalablement subir les traitements propres à **cxcp**.

(cxcp <fun> [ind1 [ind2 [ind3]]])
(cxcp (<fun>1 ... <fun>k) [ind1 [ind2 [ind3]]])

Les <fun>i ne sont pas évaluées. Compile une fonction ou une liste de fonctions en leur faisant subir préalablement les traitements propres à cxcp. Les indicateurs optionnels ind1, ind2 et ind3 ont la même signification que dans la fonction compiler du compilateur LE_LISP.

(cxcp-package <pckg> [ind1 [ind2 [ind3]]])
(cxcp-package (<pkg>1 ... <pkg>n) [ind1 [ind2 [ind3]]])

La même chose que la précédente sur toutes les fonctions d'un package ou d'une liste de packages.

3.3 Expansion à la Demande

L'expansion à la demande de fonctions LISP est réalisée par la construction cxcp-inline.

(cxcp-inline <fundescr>1 ... <fundescr>n)

Les <fundescri>, qui ne sont pas évalués sont:
- soit des noms de fonctions,
- soit des listes (<fun> <arg>1 ... <argn>), où <fun> est un nom de fonction et les <arg>j le nom des arguments de <fun> qu'on veut protéger contre le phénomène de double évaluation.

Remarque: le compilateur refuse de transformer des fonctions à nombre variable d'arguments en émettant un message d'avertissement.

Attention: Aucune vérification n'est faite sur la validité de l'expansion inline. Des cas triviaux où l'expansion inline simple peut ne pas être suffisante sont:
- les cas de double évaluation
- les cas d'appels avec effets de bords.
- les cas d'appel de FSUBR.

Un exemple typique est la fonction xcons:

```
(de xcons (a b) (cons b a))
```

Pour cette fonction, l'expansion inline est fausse si l'ordre des effets de bord des arguments d'appel importe.

Un autre cas difficile:

```
(de foo () (ommakeq foo foo))
```

qui devrait engendrer:

```
(dmd foo () `(ommakeq foo ,foo))
```

Ce cas difficile est en fait résolu par l'application du macro-expanseur avant tout traitement.

3.4 Expansion des send guidée par une évaluation

La construction présentée dans cette partie est un premier pas vers la compilation personnalisée de type "execute and compile", puisqu'elle est guidée par une première exécution sur le programme de l'utilisateur.

(ecxcp <expr>)
(ecxcp <expr> <fun> [ind1 [ind2 [ind3]]])
(ecxcp <expr> (<fun>1 ... <fun>k) [ind1 [ind2 [ind3]]])

Les <fun>i ne sont pas évalués. Avant d'appeler la fonction cxcp standard, avec les <fun>i (non évalués comme pour cxcp) et éventuellement les indicateurs, l'expression <expr> est évaluée dans un environnement où tous les appels à send dans les fonctions à compiler sont dynamiquement étendus en des selectq sur tous les types des objets successifs auxquels ce send envoie des messages. La dernière clause du selectq est de la forme:

```
(t (send ...))
```

Cet appel à send garantit l'extensibilité. Dès que le nombre de clauses dépasse la valeur de la variable #:cxcp:ecxcp, qui vaut 5 par défaut, l'expansion est annulée au profit d'un send classique.

CHAPITRE 4

La Bibliothèque Initiale

Jean-Marie Hullot

Nous présentons dans ce chapitre un certain nombre de constructions qui ont été déclarées d'utilité publique au cours du temps.

4.1 Les Records

Il s'agit ici de définir des modèles qui sont des arbres binaires équilibrés de Cons dont les feuilles sont des Fields, à partir de la liste de ces Fields. Pour cela, nous avons introduit un nouveau macromodèle de nom Record dans le langage de modèles:

(Record <model>1 ... <model>n)

qui est interprété comme un arbre binaire équilibré de Cons:

(Cons (Cons ... (Cons <model>1 <model>2) ...)
(Cons ... (Cons <model>n-1 <model>n) ...))

Exemples

```
? (defmodel toto (Record (Field a) (Field b) (Field c) (Field d)))
= toto
? ;est équivalent à
? (defmodel toto (Cons (Cons (Field a) (Field b))
?                      (Cons (Field c) (Field d))))
= toto
```

Nous donnons une construction permettant de définir des records avec une syntaxe allégée. Pour cela nous définissons d'abord ce qu'est un descripteur de champ <fielddescr>:

<fielddescr> = <symbol>
|<symbol>~<model>
|(<symbol> <object>)
|(<symbol>~<model> <object>)

Dans le contexte des constructions defrecord, deftrecord, defclass et deftclass définies ci-après, les <fielddescr> seront respectivement interprétés comme:

(Field <symbol>)
(Field <symbol> <model>)
(Field <symbol> * <object>)
(Field <symbol> <model> <object>)

(defrecord <name> <fielddescr>1 ... <fielddescr>n)
(defrecord (<name> <abbrev>) ...)

Aucun argument n'est évalué. Dans le cas ou <abbrev> n'est pas fourni, on le prend égal au pname de <name> dans le package global LE_LISP ||. ***Attention, ceci n'est pas du tout le cas pour la construction defmodel.*** Cette construction est équivalente à:

```
(defmodel (<name> <abbrev>) (Record <fielddescr>1 ... <fielddescr>n))
(defaccess <name>)
```

Exemples

```
? (defrecord toto a (b~string "") c~fix)
= toto
```

```
? (defmake toto toto (a b c))
= toto
? (setq x (toto 1 "ahlala" 3))
= (1 "ahlala" . 3)
? ({toto}:a x)
= 1
? ; mais atteption à l'exportation par défaut dans ||
? (defrecord #:toto:tata a b c)
= #:toto:tata
? '{tata}
= #:toto:tata
? (defrecord Coord (x~fix 0) (y~fix 0))
= Coord
? (defmake Coord Coord (x y))
= Coord
```

(deftrecord <name> <fielddescr>1 ... <fielddescr>n)
(deftrecord (<name> <abbrev>) ...)

Cette construction est équivalente à la précédente, sinon qu'on engendre un type au lieu d'engendrer un modèle.

Exemples

```
? (deftrecord Point coord~(Include Coord)))
= Point
? (defmake Point mkpoint (x y))
= mkpoint
? (defmake Point Point coord)
= Point
? (mkpoint 1 2)
= #(Point 1 . 2)
? (Point (Coord 1 2))
= #(Point (1 . 2))
```

4.2 Les Classes

Il s'agit ici de donner des facilités pour décrire des modèles toujours extensibles: on les définit comme des Vectors de modèles qui sont donc toujours extensibles par la droite. Ces modèles sont appelés des classes et, pour eux, la hiérarchie structurelle coincide avec la hiérarchie des espaces sémantiques. En effet, le nom d'une sous-classe est toujours interné dans l'espace de noms de la classe mère.

Pour définir les classes, nous commençons par définir une classe initiale:

```
? (defmodel Class (Field class-attributes (Vector)))
= Class
```

et pour les classes qui sont des types, une Tclasse initiale:

```
? (deftype Tclass (Field class-attributes (Vector)))
= Tclass
```

Le principe de la construction des classes consiste à étendre répétitivement ces classes au champ class-attributes. Ceci est réalisé par les constructions defclass et deftclass.

(defclass <name> <fielddescr>1 ... <fielddescr>n)
(defclass (<name> <abbrev>) ...)

Aucun argument n'est évalué. Dans le cas ou <abbrev> n'est pas fourni, on prend pour abbréviation de <name> le symbole du package global || LE_LISP dont le pname est ||. ***Attention, ceci n'est pas du tout le cas pour la construction defmodel.*** Le symbole <name> doit avoir pour packagecell un symbole <supername> qui possède une définition de classe. Cette construction est alors équivalente à:

```
(defmodel (<name> <abbrev>)
          (Extend <supername>
```

```
                    class-attributes (Vector <fielddescr>1 ... <fielddescr>n))
(defaccess <name> <fieldname>1 ... <fieldname>i)
```

Attention que pour les classes, le defaccess n'est fait que sur les nouveaux champs.

(deftclass <name> <fielddescr>1 ... <fielddescr>n)
(deftclass (<name> <abbrev>) ...)

Cette construction est équivalente à la précédente, sinon qu'on engendre un type au lieu d'engendrer un modèle et que les Tclasses sont des extensions de la Tclasse initiale Tclass.

Exemples

```
? (deftclass fiche nom prenom)
= #:Tclass:fiche
? (defmake {fiche} fiche (nom prenom))
= fiche
? (setq fiche (fiche 'Jacquemart 'Marcel))
= #(#:Tclass:fiche . #[Jacquemart Marcel])
? (deftclass {fiche}:fiche-client adresse~string)
= #:Tclass:fiche:fiche-client
? (deftclass {fiche-client}:fiche-cordonnier pointure~fix)
= #:Tclass:fiche:fiche-client:fiche-cordonnier
? (defmake {fiche-cordonnier} fiche-cordonnier
?                             (nom prenom adresse pointure))
= fiche-cordonnier
? (setq fiche (fiche-cordonnier 'Jacquemart
?                               'Marcel
?                               "Inconnue"
?                               48))
= #(#:Tclass:fiche:fiche-client:fiche-cordonnier . #[Jacquemart Marcel
Inconnue 48])
? ; sacrée pointure!
? (deftclass {fiche}:fiche-police condamnations~string)
= #:Tclass:fiche:fiche-police
? (defmake {fiche-police} fiche-police (nom prenom condamnations))
= fiche-police
? (fiche-police 'Jacquemart 'Marcel "Vol de Chaussures")
= #(#:Tclass:fiche:fiche-police . #[Jacquemart Marcel Vol de Chaussures])
? ; pour une fois qu'il en avait trouvé à sa taille!
```

4.3 Les Arbres

Les arbres sont des types construits à partir d'un type arbre initial:

```
(deftype Tree (Cons (Field sons)
                    (Field tree-attributes (Vector))))
```

Par un principe tout à fait analogue à celui utilisé pour les classes, les extensions successives se font en étendant le Vector du champ tree-attributes. Ceci est réalisé par la construction deftree:

(deftree <name> <fielddescr>1 ... <fielddescrn>)
(deftree (<name> <abbrev>)) <fielddescr>1 ... <fielddescr>n)

Cette contruction est analogue à deftclass sinon qu'on étend à partir du type Tree au lieu du type Tclass.

La nouveauté des arbres par rapport aux classes réside dans la possibilité de définir des "constructeurs", par exemple les opérateurs du langage quand on utilise les arbres pour construire la syntaxe abstraite d'un langage de programmation. Il s'agit donc, étant donné un arbre possédant un certains nombres d'attributs, de spécifier comment sont structurés ses fils.

(defcons <name> <sonsdescr> <fielddescr>1 ... <fielddescr>n)
(defcons (<name> <abbrev>) <sonsdescr> ...)

Comme pour les classes et les arbres, <name> est un symbole dont le packagecell <supername> doit avoir une définition d'arbre et <abbrev> s'il est omis est pris égal à (symbol '|| <name>). Les <fielddescr>i sont interprétés comme pour le deftree, i.e. ce sont des champs qui sont ajoutés en queue de tree-attributes. <sonsdescr> est soit un fielddescr auquel cas le champ sons a pour modèle <fielddescr>, soit une liste de <fielddescr> auquel cas le champ sons a pour modèle le Vector contruit à partir de cette liste.
De plus une fontion de nom <abbrev> de création du type est engendrée prenant un nombre variable d'argument dans le cas ou <sonsdescr> est un <fielddescr> et un nombre d'arguments égal à la longueur de <fielddescr> autrement. Ces arguments permettent de construire le champ sons des instances (voir exemples).
Enfin un defaccess est fait sur tous les nouveaux champs et tous les champs spéciaux que sont les fils.

Exemples

Nous présentons en exemple l'implémentation d'une syntaxe abstraite pour un sous ensemble du langage de description de transparents Flip de G. Kahn. Il s'agit d'un petit langage géométrique permettant de découper le plan en bandes horizontales (l'opérateur horiz à nombre variable d'arguments), en bandes verticales (l'opérateur vertic à nombre variable d'arguments), en régions rectangulaires atomiques contenant du texte (l'opérateur aligner à nombre variable de chaînes de caractères arguments) et en régions rectangulaires atomiques contenant une diagonale d'une certaine couleur (l'opérateur diag à un argument, sa couleur). Chaque bande ou région possède une proportion qui permet de calculer sa taille propre dans sa bande mère, elle peut être encadrée d'une certaine couleur, peinte d'une certaine couleur, le texte peut y être écrit d'une certaine couleur.

Tous ces attributs communs à tous les objets Flip, seront stockées dans une structure Flip définie par:

```
? (deftree Flip
?          (proportion~fix 1)
?          cadre
?          texte
?          peinture
= #:Tree:Flip
```

Nous définissons les différents opérateurs du langage:

```
? (defcons {Flip}:horiz sons~(List Flip))
= #:Tree:Flip:horiz
? (defcons {Flip}:vertic sons~(List Flip))
= #:Tree:Flip:vertic
? (defcons {Flip}:aligner sons~(List string))
= #:Tree:Flip:aligner
? (defcons {Flip}:diag (color~color))
= #:Tree:Flip:diag
```

et nous illustrons par des exemples les diverses fonctions engendrées:

```
? (setq flip (diag 'rouge))
= #(#:Tree:Flip:diag #[rouge] . #[1 () () ()])
? ({Tree}:sons flip)
= #[rouge]
? ({diag}:color flip)
= rouge
? ({diag}:sons flip)
** eval : fonction indefinie : #:Tree:Flip:diag:sons
   [stack 3]    (lock ...)
   [stack 2]    (tag #:system:error-tag ...)
   [stack 1]    (itsoft ...)
? (setq flip (aligner "toto" "tutu" "titi"))
= #(#:Tree:Flip:aligner (toto tutu titi) . #[1 () () ()])
```

```
? ({aligner}:sons flip)
= (toto tutu titi)
? (setq flip (horiz (diag 'vert) (diag 'bleu)))
= #(#:Tree:Flip:horiz (#(#:Tree:Flip:diag #[vert] . #[1 () () ()]) #(#:Tree:
Flip:diag #[bleu] . #[1 () () ()])) . #[1 () () ()])
? (sendq proportion flip)
= 1
? ({Flip}:cadre flip 'bleu)
= bleu
? (sendq cadre flip)
= bleu
```

L'implémentation complète de Flip est donnée en Annexe IV.

4.4 Les Règles

Nous avons vu comment la construction send donne un rôle tout particulier à son premier argument, puisque c'est sur lui qu'est fait le décodage de type permettant de trouver la fonction qu'il faut appliquer. Parfois il serait agréable de pouvoir faire la recherche de la fonction à appliquer en profitant des informations de types sur plusieurs arguments. CEYX propose une construtions permettant de le faire dans le cas de deux arguments, elle sera étendue ultérieurement au cas de n arguments typés.

Nous expliquons cette construction sur un exemple. Nous voulons définir une opération d'addition sur les réels et les complexes. Pour cela, nous définissons tout d'abord les types Reel et Complexe:

```
? (deftype Reel (Field val number 0))
= Reel
? (defmake Reel reel (val))
= reel
? (defaccess Reel)
= Reel
? (deftype Complexe (Cons (Field reelle number 0)
?                         (Field imaginaire number 0)))
= Complexe
? (defmake Complexe complexe (reelle imaginaire))
= complexe
? (defaccess Complexe)
= Complexe
```

Il nous est assez difficile d'exprimer en CEYX une loi d'addition sans définir une sémantique + sur chacun des deux types envoyant des sémantiques relais +reel ou +complexe à la liste d'arguments renversée. Le but de la construction defrule est de vous éviter toute cette salade.

Ce qu'il faut savoir, c'est qu'au moment du send la recherche de la fonction à appliquer sera faite en tenant compte des types des deux premiers arguments en ordre lexicographique pour la remontée hiérarchique. Dans le cas de type A2 <=p A1 pour le premier argument et B2 <=p B1 pour le deuxième argument, nous inspecterons donc successivement A2×B2, A2×B1, A1×B2, A1×B1.

```
? (defrule + (x~Reel y~Reel)
?            (reel (+ ({Reel}:val x) ({Reel}:val y))))
= +
? (sendq + (reel 1) (reel 2)))
= #(Reel . 3)
? (sendq + (reel 1) (complexe 2 3))
** + : l'argument n'est pas un nombre : #(Complexe 2 . 3)
   [stack 4]    (#:Reel:+ ...)
   [stack 3]    (lock ...)
   [stack 2]    (tag #:system:error-tag ...)
   [stack 1]    (itsoft ...)
? ; et oui puisqu'on ne trouve pas de cas Reel×Complexe, on remonte
? ; jusqu'au package global, ou on trouve la fonction LISP +
? ; classique, qui provoque l'erreur puisque le type de l'argument
```

```
? ; n'est pas le bon.
? (defrule + (x~Reel y~Complexe)
?            (complexe (+ ({Reel}:val x) ({Complexe}:reelle y))
?                      ({Complexe}:imaginaire y)))
= +
? ; et cette fois-ci on peut faire
? (sendq + (reel 1) (complexe 2 3))
= #(Complexe 3 . 3)
? (defrule + (x~Complexe y~Complexe)
?         (complexe (+ ({Complexe}:reelle x) ({Complexe}:reelle y))
?                   (+ ({Complexe}:imaginaire x)
?                      ({Complexe}:imaginaire y))))
= +
? (sendq + (complexe 1 2) (complexe 3 4))
= #(Complexe 4 . 6)
```

(defrule <name> (<arg>1~<type>1 <arg>2~<type>2 . <args>) . <body>)

Définit la règle de nom name sur le produit cartésien <type>1x<type>2. On notera qu'un règle prend donc deux arguments typés et autant d'arguments LISP standards qu'on le désire.

(undefrule <name>)

Enlève toute trace de la règle de nom <name>. Après cette opération la règle <name> n'existe littéralement plus.

4.5 Déstructuration

Il s'agit de constructions permettant de déstructurer automatiquement un objet dans les variables locales d'un let.

(olet (<modelname> <dfields> <obj>) . <body>)

<dfields>= (<dfield>1 ... <dfield>n) ou <dfield> est soit un nom de champ de <modelname>, soit une liste (<fieldname>i <symbol>i).
Cette construction se macrogénère en

```
(let (<exp>1 ... <expn>) . <body>))
```

où

<exp>i = (<fieldname>i (ogetq <modelname> <fieldname>i <object>))
ou
<exp>i = (<symbol>i (ogetq <modelname> <fieldname>i <object>)).

Exemples:

```
? (defmake toto toto (a b c))
= toto
? (olet (toto (a c) (toto 1 2 3)) (print a) (print c))))
1
3
= 3
2
3
= 3
```

(demethod <name> (<obj> . <args>) <dfields> . <body>)

Cette construction permet de déstructurer des champs sélectionnés de son premier argument <obj> dans des variables locales de la fonction. Elle est équivalente à:

```
(de <name> <args> (olet ((packagecell <name>) <dfields> <obj>) . <body>))
```

Exemples

```
? (demethod {toto}:print (x) (a b c)
?       (print "a: " a)
?       (print "b: " b)
```

```
?       (print "c: " c)
?       t)
= #:toto:print
? (pretty #:toto:print)
(de #:toto:print (x)
    (olet (toto (a b c) x)
          (print "a: " a) (print "b: " b) (print "c: " c) t))
= ()
```

(unde <name>)

Enlève la définition de fonction du symbole <name>.

4.6 O..Q

(ochangeq <modelname> <object> <field>1 <val>1 ... <field>n <val>n)

<modelname> et les <field>i qui doivent être des noms de champs ne sont pas évalués, <object> doit être une instance de <modelname>, ses champs <field>i prennent les valeurs <val>i et <object> est ramené en valeur.

(ofunq <modelname> <field> <object> <fun> . <args>)

<modelname> et <field> et <fun> ne sont pas évalués, <object> doit être une instance de <modelname>, son champ field est remplace, par

```
(apply <fun> (cons (ogetq <modelname> <field> <object>) <args>))
```

Un cas particulier:

(oconsq <modelname> <field> <object> <val>)

<modelname> et <field> ne sont pas évalués, <object> doit être une instance de <modelname>, son champ <field> est remplacé par

```
(cons <val> (ogetq <modelname> <field> <object>))
```

Cette construction peut être définie en LISP par:

```
(defmacro oconsq (model field obj val)
    `(ofunq ,model ,field ,obj xcons ,val))
```

Exemples

```
? (setq x (tata 1 2))
= #(tata 1 . 2)
? (ochangeq tata x a () b 0)
= #(tata () . 0)
? (ofunq tata b x + 1)
= #(tata () . 1)
? (oconsq tata a x 'aa)
= #(tata (aa) . 1)
? (oconsq tata a x 'bb)
= #(tata (bb aa) . 1)
```

4.7 Mécanisme de Trace

(tracesems <msg>1 ... <msg>n)

Les arguments ne sont pas évalués. Permet de tracer au sens LISP, toutes les propriétés sémantiques de pname <msg>1, ..., <msg>n.

ANNEXE I

Le Kit de Distribution

Jean-Marie Hullot, Bertrand Serlet

Janvier 1985

```
(SETQ #:SYSTEM:READ-CASE-FLAG T)
```

I.1 Installation

Vous venez de recevoir le kit de distribution CEYX version 15. Vous devez avoir au minimum dans ce kit les fichiers suivants:

- **make.ll**, c'est à partir de ce fichier que sera engendrée une image mémoire standard de CEYX. C'est le fichier que vous êtes en train de lire.
- **ac.ll**, qui contient la définition du macro-caractère {.

Dans le cas d'une distribution *avec les sources*:

- **ceyx.ll**, ce fichier contient tout le noyau de CEYX, correspondant aux chapitres 1 et 2 du manuel.
- **cxcp.ll**, qui contient le précompilateur CEYX décrit au chapitre 3 du manuel.
- **ceyxlib.ll**, qui contient la biliothèque initiale CEYX décrite au chapitre 4.

Dans le cas d'une distribution *sans les sources*:

- **ceyx.ll**, qui regroupe les trois fichiers précédents en format loader.

Une distribution standard contient également généralement les fichiers

- **defrule.ll**, qui contient la construction defrule pour définir des règles (*cf.* chapitre 4).
- **describe.ll**, qui contient un certains nombres d'utilitaires définis au chapitre 1.
- **coord.ll**, dans lequel sont rangées des structures de manipulations d'objets élémentaires dans le plan cartésien. Ce fichier est inclus in extenso dans l'annexe II.
- **union.ll**, dans lequel nous définissons une structure d'ensembles ordonnés. Ce fichier est inclus in extenso dans l'annexe III.
- **leflip.ll**, implémentation du langage Flip. Ce fichier est inclus in extenso dans l'annexe IV.
- **stream.ll**, dans lequel sont définies des structures permettant de jouer avec les entrées sorties. Ce fichier est inclus in extenso dans l'annexe V.
- **vprint.ll**, qui est le code du formatteur dont la documentation est donnée dans le rapport "*VPRINT, Le Composeur CEYX*".

D'autres fichiers peuvent se trouver là, regardez ce qu'ils font, généralement chaque fichier est auto-documenté.

Avant toute chose, vous devez mettre dans la valeur de la variable #:CEYX:directory, l'emplacement où vous voulez installer CEYX sur votre machine. Pour ça, changez dans la ligne suivante /usr/local/ceyxv15/ par l'emplacement désiré.

```
(DEFVAR #:CEYX:DIRECTORY "/usr/local/ceyxv15/")
```

Ceci étant fait, vous n'avez plus qu'à:

- appeler lelisp,
- charger le fichier make.ll par (load "make.ll"),
- appeler (make-ceyx).

et ainsi une image mémoire de nom ceyx.core est construite. Notez que l'image mémoire construite ne contient que les fichiers de base ceyx.ll, ceyxlib.ll, cxcp.ll (ou seulement ceyx.ll qui contient les trois pour les distributions sans le source). Les autres peuvent être chargés à la demande avec la fonction ceyx-load.

La fonction make-ceyx est définie ci-dessous:

```
(DE MAKE-CEYX ()
   (PRINT "Avant chargement : " (GC T))
   (CEYX-LOAD AC)
   (CEYX-LOAD CEYX)
   (CEYX-AUTOLOAD DESCRIBE DESCRIBE MDESCRIBE TRACESEMS HELP)
   (CEYX-AUTOLOAD DEFRULE DEFRULE)
   (PRINT "Avant compilation : " (GC T))
   (CXCP)
   (PRINT "Apres compilation : " (GC T))
   (PROGN (SAVE-CORE (CATENATE #:CEYX:DIRECTORY "ceyx.core"))
      (HERALD)
      (INITTY)
      "CEYX - Version 15"))
```

I.2 Divers

Pour charger seulement une fois les fichiers de la librairie:

```
(DF CEYX-LOAD ARGS
    (MAPC (LAMBDA (X)
             (UNLESS (GET X 'CEYX-LOADED)
                 (PRINT "Loading from CEYX library " X)
                 (LOADFILE (CATENATE #:CEYX:DIRECTORY X ".ll") T)
                    (PUTPROP X T 'CEYX-LOADED)))
          ARGS))

(DF CEYX-AUTOLOAD (FILE . SYMBS)
    (EVAL (MCONS 'AUTOLOAD (CATENATE #:CEYX:DIRECTORY FILE ".ll") SYMBS)))
```

ANNEXE II

Le Plan Cartésien

Jean-Marie Hullot, Bertrand Serlet, Jean Vuillemin

Ce chapitre est un exemple d'utilisation de modèles non auto-typés c'est à dire définis par defmodel, defrecord ... Nous rappelons que les Records CEYX sont implémentés comme des arbres équilibrés de cellules cons.

Toutes les fonctions décrites ici sont utilisables sous CEYX à condition d'avoir chargé le fichier coord.ll de la bibliothèque CEYX, qui n'est d'ailleurs pas autre chose que ce chapitre:

```
? (ceyx-load coord)
= coord.ll
```

Nous présentons ici des structures permettant de représenter des coordonnées et des rectangles dans le plan cartésien et les propriétés sémantiques élémentaires de ces structures.

II.1 Les Coordonnées

Le Modèle: Coord	
Fonction de Création	
Coord	(x y)
Champs	
x	number
y	number
Propriétés Sémantiques	
•	(coord ratio)
+	(coord dxy)
+•	(coord homvect)
-	(coord dxy)
max	(coord1 coord2)
min	(coord1 coord2)
translate	(coord dx dy)
x	obj
y	obj
Sous Modèles	
Dxy	
Ratio	

Nous nous plaçons dans un repère de coordonnées de type écran, c'est à dire que l'axe des x est horizontal de la gauche vers la droite et l'axe des y vertical du haut vers le bas. Les *coordonnées* sont représentées par des records à deux champs:

```
(DEFRECORD COORD (X~NUMBER 0) (Y~NUMBER 0))
(DEFMAKE {COORD} COORD (X Y))
```

Exemples:

```
? (setq c01 (Coord 0 1))
= (0 . 1)
? ({Coord}:x c01)
```

```
= 0
? ({Coord}:y c01)
= 1
```

Le min et le max sur les coordonnées:

```
(DE {COORD}:MIN (COORD1 COORD2)
    (COORD (MIN ({COORD}:X COORD1) ({COORD}:X COORD2))
           (MIN ({COORD}:Y COORD1) ({COORD}:Y COORD2))))

(DE {COORD}:MAX (COORD1 COORD2)
    (COORD (MAX ({COORD}:X COORD1) ({COORD}:X COORD2))
           (MAX ({COORD}:Y COORD1) ({COORD}:Y COORD2))))
```

Exemples:

```
? (setq c1 (Coord 1 3) c2 (Coord 4 2))
= (4 . 2)
? ({Coord}:min c1 c2)
= (1 . 2)
? ({Coord}:max c1 c2)
= (4 . 3)
? c1
= (1 . 3)
? c2
= (4 . 2)
```

Pour translater une coordonnée de dx, dy:

```
(DE {COORD}:TRANSLATE (COORD DX DY)
    ({COORD}:X COORD (+ ({COORD}:X COORD) DX))
    ({COORD}:Y COORD (+ ({COORD}:Y COORD) DY))
    COORD)
```

Exemples:

```
? ({Coord}:translate c1 7 5)
= (8 . 8)
? c1
= (8 . 8)
```

II.2 Les Vecteurs

Le Modèle: Vect	
Fonctions de Création	
mkvect	(xorg yorg xext yext)
Vect	(org ext)
Champs	
org	(Cons ...)
xorg	number
yorg	number
ext	(Cons ...)
xext	number
yext	number
Propriétés Sémantiques	
*	(vect ratio)
+	(vect dxy)
+*	(vect homvect)
dx	(vect)
dy	(vect)
ext	obj
null	(vect)
org	obj
translate	(vect dx dy)
xext	obj
xorg	obj
yext	obj
yorg	obj
Sous Modèle	
HomVect	

Les vecteurs sont représentés par un couple de coordonnées dont l'une est appelée origine et l'autre extrêmité:

```
(DEFRECORD VECT ORG~(ALTER COORD X (FIELD XORG NUMBER 0)
                                Y (FIELD YORG NUMBER 0))
                EXT~(ALTER COORD X (FIELD XEXT NUMBER 0)
                                Y (FIELD YEXT NUMBER 0)))

(DEFMAKE {VECT} VECT (ORG EXT))
(DEFMAKE {VECT} MKVECT (XORG YORG XEXT YEXT))
```

Exemples:

```
? (setq vect (Vect c1 c2))
= ((8 . 8) 4 . 2)
? ({Vect}:org vect)
= (8 . 8)
? ({Vect}:xorg vect)
= 8
? ({Vect}:yext vect)
= 2
```

Nous considèrerons dans la suite que le vecteur est fermé en son origine et ouvert en son extrêmité, c'est à dire que son extrêmité ne lui appartient pas. Ainsi un vecteur est-il vide si et seulement si son origine et son extrêmité sont confondues:

```
(DE {VECT}:NULL (VECT)
    (AND (= ({VECT}:XORG VECT) ({VECT}:XEXT VECT))
         (= ({VECT}:YORG VECT) ({VECT}:YEXT VECT))))
```

Exemples:

```
? ({Vect}:null vect)
= ()
? ({Vect}:null (Vect (Coord 3 4) (Coord 3 4)))
= 4
```

La différence des coordonnées en x et en y:

```
(DE {VECT}:DX (VECT)
    (- ({VECT}:XEXT VECT) ({VECT}:XORG VECT)))
(DE {VECT}:DY (VECT)
    (- ({VECT}:YEXT VECT) ({VECT}:YORG VECT)))
(CXCP-INLINE ({VECT}:DX VECT))
(CXCP-INLINE ({VECT}:DY VECT))
```

Exemples:

```
? vect
= ((8 . 8) 4 . 2)
? ({Vect}:dx vect)
= -4
? ({Vect}:dy vect)
= -6
```

Translation d'un vecteur de dx, dy:

```
(DE {VECT}:TRANSLATE (VECT DX DY)
    ({COORD}:TRANSLATE ({VECT}:ORG VECT) DX DY)
    ({COORD}:TRANSLATE ({VECT}:EXT VECT) DX DY)
    VECT)
```

II.3 Les Rectangles

Le Modèle: Rect	
Fonctions de Création	
mkrect	(x y w h)
Rect	(coord1 coord2)
Champs	
org	(Cons ...)
xorg	number
yorg	number
ext	(Cons ...)
xext	number
yext	number
Propriétés Sémantiques	
•	(vect ratio)
+	(vect dxy)
+•	(vect homvect)
<-Rect	(rect1 rect2)
<-inter	(rect1 rect2)
contains-coord	(rect coord)
contains-rect	(rect1 rect2)
ext	obj
height	(vect)
inter	(rect1 rect2)
inter?	(rect1 rect2)
mkinter	(rect1 rect2)
null	(vect)
org	obj
translate	(vect dx dy)
union	(rect1 rect2)
width	(vect)
xext	obj
xorg	obj
yext	obj
yorg	obj
Sous Modèle	
ClipRect	

Les *rectangles* sont déterminés par leur coin supérieur **gauche et leur coin inférieur droit**. Ils sont implémentés de la même manière que des **Vect**.

```
(DEFMODEL RECT VECT)

(DE RECT (COORD1 COORD2)
    (VECT ({COORD}:MIN COORD1 COORD2) ({COORD}:MAX COORD1 COORD2)))

(DEFMAKE {RECT} RECT)

(DE MKRECT (X Y W H)
    (MKVECT X Y (+ X W) (+ Y H)))

(DEFMAKE {RECT} MKRECT)

(DEFACCESS RECT)
```

Exemples:

```
? (setq rect1 (Rect (Coord 0 0) (Coord 10 10)))
= ((0 . 0) 10 . 10)
? (setq rect2 (Rect (Coord 20 20) (Coord 10 10)))
= ((10 . 10) 20 . 20)
```

Quelques synonymes:

```
(SYNONYMQ {RECT}:NULL {VECT}:NULL)
(SYNONYMQ {RECT}:WIDTH {VECT}:DX)
(SYNONYMQ {RECT}:HEIGHT {VECT}:DY)
(SYNONYMQ {RECT}:TRANSLATE {VECT}:TRANSLATE)
```

Exemples:

```
? ({Rect}:org rect1)
= (0 . 0)
? ({Rect}:width rect1)
= 10
? ({Rect}:height rect2)
= 10
? rect2
= ((10 . 10) 20 . 20)
? ({Rect}:+ rect2 (Coord 1 1))
= ((11 . 11) 21 . 21)
```

Pour déterminer si un **Rect** contient une **Coord**:

```
(DE {RECT}:CONTAINS-COORD (RECT COORD)
    (NOT
     (OR (< ({COORD}:X COORD) ({RECT}:XORG RECT))
         (>= ({COORD}:X COORD) ({RECT}:XEXT RECT))
         (< ({COORD}:Y COORD) ({RECT}:YORG RECT))
         (>= ({COORD}:Y COORD) ({RECT}:YEXT RECT)))))
(CXCP-INLINE ({RECT}:CONTAINS-COORD RECT COORD))
```

Exemples:

```
? rect1
= ((0 . 0) 10 . 10)
? ({Rect}:contains-coord rect1 (Coord 0 0))
= t
? ({Rect}:contains-coord rect1 (Coord 5 5))
= t
? ({Rect}:contains-coord rect1 (Coord 30 40))
= ()
? ({Rect}:contains-coord rect1 (Coord 10 10))
= ()
? ({Rect}:contains-coord rect1 (Coord 3 10))
= ()
```

Pour déterminer si un **Rect** en contient un autre:

```
(DE {RECT}:CONTAINS-RECT (RECT1 RECT2)
    (AND
      ({RECT}:CONTAINS-COORD RECT1 ({RECT}:ORG RECT2))
      ({RECT}:CONTAINS-COORD RECT1 ({COORD}:+ (COORD -1 -1)
                                             ({RECT}:EXT RECT2)))))
(CXCP-INLINE ({RECT}:CONTAINS-RECT RECT1 RECT2))
```

Exemples:

```
? rect1
= ((0 . 0) 10 . 10)
? rect2
= ((6 . 6) 16 . 16)
? ({Rect}:contains-rect rect1 rect2)
= ()
? ({Rect}:contains-rect rect1 (Rect (Coord 3 3) (Coord 10 10)))
= t
```

Ramène en valeur le rectangle enveloppant de deux rectangles:

```
(DE {RECT}:UNION (RECT1 RECT2)
    (RECT
        ({COORD}:MIN ({RECT}:ORG RECT1) ({RECT}:ORG RECT2))
        ({COORD}:MAX ({RECT}:EXT RECT1) ({RECT}:EXT RECT2))))
```

Ramène nil ou le rectangle intersection de deux rectangles.

```
(DE {RECT}:INTER (RECT1 RECT2)
   (IF ({RECT}:INTER? RECT1 RECT2)
    ({RECT}:MKINTER RECT1 RECT2))

(DE {RECT}:INTER? (RECT1 RECT2)
   (AND (< (MAX ({RECT}:XORG RECT1) ({RECT}:XORG RECT2))
           (MIN ({RECT}:XEXT RECT1) ({RECT}:XEXT RECT2)))
        (< (MAX ({RECT}:YORG RECT1) ({RECT}:YORG RECT2))
           (MIN ({RECT}:YEXT RECT1) ({RECT}:YEXT RECT2)))))

(DE {RECT}:MKINTER (RECT1 RECT2)
    (RECT
       ({COORD}:MAX ({RECT}:ORG RECT1) ({RECT}:ORG RECT2))
       ({COORD}:MIN ({RECT}:EXT RECT1) ({RECT}:EXT RECT2))))

(CXCP-INLINE ({RECT}:INTER? RECT1 RECT2))
(CXCP-INLINE ({RECT}:MKINTER RECT1 RECT2))
```

Exemples:

```
? rect1
= ((0 . 0) 10 . 10)
? rect2
= ((11 . 11) 21 . 21)
? ({Rect}:union rect1 rect2)
= ((0 . 0) 21 . 21)
? ({Rect}:inter? rect1 rect2)
= ()
? ({Rect}:translate rect2 -5 -5)
= ((-5 . -5) 16 . 16)
? rect2
= ((6 . 6) 16 . 16)
? ({Rect}:inter rect1 rect2)
= ((6 . 6) 10 . 10)
```

Quelques opérations de modification en place:

```
(DE {RECT}:<-RECT (RECT1 RECT2)
    ({RECT}:XORG RECT1 ({RECT}:XORG RECT2))
    ({RECT}:YORG RECT1 ({RECT}:YORG RECT2))
    ({RECT}:XEXT RECT1 ({RECT}:XEXT RECT2))
    ({RECT}:YEXT RECT1 ({RECT}:YEXT RECT2))
    RECT1)

(CXCP-INLINE ({RECT}:<-RECT RECT1 RECT2))

(DE {RECT}:<-INTER (RECT1 RECT2)
    (COND
       ((NULL ({RECT}:INTER? RECT1 RECT2))
        ({RECT}:XEXT RECT1 ({RECT}:XORG RECT1))
        ({RECT}:YEXT RECT1 ({RECT}:YORG RECT1))
        RECT1)
       (T ({RECT}:XORG RECT1
              (MAX ({RECT}:XORG RECT1) ({RECT}:XORG RECT2)))
          ({RECT}:YORG RECT1
              (MAX ({RECT}:YORG RECT1) ({RECT}:YORG RECT2)))
          ({RECT}:XEXT RECT1
              (MIN ({RECT}:XEXT RECT1) ({RECT}:XEXT RECT2)))
          ({RECT}:YEXT RECT1
              (MIN ({RECT}:YEXT RECT1) ({RECT}:YEXT RECT2))))
```

```
RECT1)))
```

Les rectangles de clip, utilisés surtout pour inclure dans d'autres structures par (Include {ClipRect}).

Le Modèle: #:Rect:ClipRect	
Abréviation: ClipRect.	
Champs	
cliporg	(Cons ...)
clipxorg	number
clipyorg	number
clipext	(Cons ...)
clipxext	number
clipyext	number

```
(DEFMODEL ({RECT}:CLIPRECT CLIPRECT)
          (CONS
             (FIELD CLIPORG (CONS
                              (FIELD CLIPXORG NUMBER 0)
                              (FIELD CLIPYORG NUMBER 0)))
             (FIELD CLIPEXT (CONS
                              (FIELD CLIPXEXT NUMBER 0)
                              (FIELD CLIPYEXT NUMBER 0)))))
```

II.4 Transformations

II.4.1 Définition

Pour voir une coordonnée comme une translation:

Le Modèle: #:Coord:Dxy	
Abréviation: Dxy.	
Fonction de Création	
Dxy	(dx dy)
Champs	
dx	number
dy	number
Propriétés Sémantiques	
dx	obj
dy	obj

```
(DEFMODEL ({COORD}:DXY DXY)
          (ALTER COORD
                 X (FIELD DX NUMBER 0)
                 Y (FIELD DY NUMBER 0)))
(DEFACCESS {DXY})
(DEFMAKE {DXY} DXY (DX DY))
```

Pour voir une coordonnée comme une homothétie:

Le Modèle: #:Coord:Ratio	
Abréviation: Ratio.	
Fonction de Création	
Ratio	(ratiox ratioy)
Champs	
ratiox	number
ratioy	number
Propriétés Sémantiques	
ratiox	obj
ratioy	obj

```
(DEFMODEL ({COORD}:RATIO RATIO)
          (ALTER COORD
                 X (FIELD RATIOX NUMBER 1)
                 Y (FIELD RATIOY NUMBER 1)))
(DEFACCESS {RATIO})
(DEFMAKE {RATIO} RATIO (RATIOX RATIOY))
```

Pour voir un vecteur comme la composition d'une translation **et d'une homothétie:**

Le Modèle: #:Vect:HomVect	
Abréviation: HomVect.	
Fonction de Création	
HomVect	(dxy ratio)
Champs	
dxy	(Cons ...)
dx	number
dy	number
ratio	(Cons ...)
ratiox	number
ratioy	number
Propriétés Sémantiques	
dx	obj
dxy	obj
dy	obj
ratio	obj
ratiox	obj
ratioy	obj

```
(DEFMODEL ({VECT}:HOMVECT HOMVECT)
          (ALTER VECT
                 ORG (FIELD DXY (INCLUDE {DXY}))
                 EXT (FIELD RATIO (INCLUDE {RATIO}))))
(DEFACCESS {HOMVECT})
(DEFMAKE {HOMVECT} HOMVECT (DXY RATIO))
```

II.4.2 Application des Transformations

```
(DE {COORD}:+ (COORD DXY)
    ({COORD}:X COORD (+ ({COORD}:X COORD) ({DXY}:DX DXY)))
    ({COORD}:Y COORD (+ ({COORD}:Y COORD) ({DXY}:DY DXY)))
    COORD)

(DE {COORD}:- (COORD DXY)
    ({COORD}:X COORD (- ({COORD}:X COORD) ({DXY}:DX DXY)))
    ({COORD}:Y COORD (- ({COORD}:Y COORD) ({DXY}:DY DXY)))
    COORD)

(DE {COORD}:* (COORD RATIO)
    ({COORD}:X COORD (* ({COORD}:X COORD) ({RATIO}:RATIOX RATIO)))
```

```
    ({COORD}:Y COORD (* ({COORD}:Y COORD) ({RATIO}:RATIOY RATIO)))
    COORD)

(DE {COORD}:+* (COORD HOMVECT)
    ({COORD}:+ COORD ({HOMVECT}:DXY HOMVECT))
    ({COORD}:* COORD ({HOMVECT}:RATIO HOMVECT))
    COORD)

(DE {VECT}:+ (VECT DXY)
    ({COORD}:+ ({VECT}:ORG VECT) DXY)
    ({COORD}:+ ({VECT}:EXT VECT) DXY)
    VECT)

(DE {VECT}:* (VECT RATIO)
    ({COORD}:* ({VECT}:ORG VECT) RATIO)
    ({COORD}:* ({VECT}:EXT VECT) RATIO)
    VECT)

(DE {VECT}:+* (VECT HOMVECT)
    ({VECT}:+ VECT ({HOMVECT}:DXY HOMVECT))
    ({VECT}:* VECT ({HOMVECT}:RATIO HOMVECT))
    VECT)

(SYNONYMQ {RECT}:+ {VECT}:+)
(SYNONYMQ {RECT}:* {VECT}:*)
(SYNONYMQ {RECT}:+* {VECT}:+*)
```

ANNEXE III

Les Ensembles Ordonnés

Jean-Marie Hullot

Ce chapitre est un exemple d'utilisation de modèles auto-typés c'est à dire définis par deftype, ...

Toutes les fonctions décrites ici sont utilisables sous CEYX à condition d'avoir chargé le fichier union.ll de la bibliothèque CEYX, qui n'est d'ailleurs pas autre chose que ce chapitre:

```
? (ceyx-load union)
= union.ll
```

Nous présentons ici une structure analogue à la structure de liste mais pour laquelle les opérations conc et merge (nconc) sont réalisées en temps constant. Ceci est réalisé en conservant toujours un pointeur vers la dernière cellule de la liste (last).

III.1 Définition et Création

Le Type: Union	
Fonction de Création	
Union	list
Champs	
list	(List ...)
last	•
Propriétés Sémantiques	
car	(union)
clear	(union)
conc	(union x)
cons	(union x)
delete	(union x)
flat	(union)
last	obj
list	obj
member	(union item)
merge	(union1 union2)
pop-down	(union item)
pop-up	(union item)

Un *ensemble ordonné* est une structure à deux champs *list* et *last* telle que:
- ***list*** **pointe toujours vers une liste LISP,**
- ***last*** **pointe vers (last *list*).**

```
(DEFTRECORD UNION LIST~(LIST *) LAST)
(DEFACCESS UNION)
```

Pour construire un ensemble ordonné dont les n premiers éléments sont passés en arguments:

```
(DE UNION LIST
    (OMAKEQ UNION LIST LIST LAST (LAST LIST)))
(DEFMAKE |UNION| UNION)
```

Exemples:

```
? (setq u (Union 'a 'b 'c))
= #(Union (a b c) c)
? ({Union}:list u)
= (a b c)
? ({Union}:last u)
= (c)
```

III.2 Fonctions de Manipulation

Le premier élément:

```
(DE {UNION}:CAR (UNION)
    (CAR ({UNION}:LIST UNION)))

(CXCP-INLINE {UNION}:CAR)
```

Pour vider un ensemble ordonné:

```
(DE {UNION}:CLEAR (UNION)
    ({UNION}:LIST UNION ())
    ({UNION}:LAST UNION ())
    UNION)
```

Pour ajouter un élément en tête:

```
(DE {UNION}:CONS (UNION X)
    (IF ({UNION}:LAST UNION)
        ({UNION}:LIST UNION (CONS X ({UNION}:LIST UNION)))
        ({UNION}:LIST UNION (LIST X))
        ({UNION}:LAST UNION ({UNION}:LIST UNION)))
    UNION)
```

Pour ajouter un élément en queue:

```
(DE {UNION}:CONC (UNION X)
    (IF ({UNION}:LAST UNION)
        (PROGN
           (RPLACD ({UNION}:LAST UNION) (LIST X))
           ({UNION}:LAST UNION (CDR ({UNION}:LAST UNION))))
        ({UNION}:LIST UNION (LIST X))
        ({UNION}:LAST UNION ({UNION}:LIST UNION)))
    UNION)
```

Exemples:

```
? (send 'clear u)
= #(Union ())
? u
= #(Union ())
? (send 'cons u 'b)
= #(Union (b) b)
? (send 'conc u 'c)
= #(Union (b c) c)
? (send 'cons u 'a)
= #(Union (a b c) c)
```

Pour supprimer un élément:

```
(DEMETHOD {UNION}:DELETE (UNION X) (LIST LAST)
    (WHEN LIST
          (IF (NEQ X (CAR LIST))
              (WHEN (EQ (LIST-DELETE LIST X) LAST)
                    ({UNION}:LAST UNION (LAST LIST)))
```

```
                ({UNION}:LIST UNION (CDR LIST))
                (WHEN (EQ LIST LAST) ({UNION}:LAST UNION ()))))
        UNION)
    (DE LIST-DELETE (LIST X)
        (WHEN (CDR LIST)
              (IF (NEQ (CADR LIST) X)
                  (LIST-DELETE (CDR LIST) X)
                  (PROG1 (CDR LIST)
                         (RPLACD LIST (CDDR LIST))))))
```

Exemples:

```
? (send 'delete u 'c)
= #(Union (a b) b)
? (send 'delete u 'a)
= #(Union (b) b)
? (send 'conc u (setq x '(1 2 3)))
= #(Union (b (1 2 3)) (1 2 3))
? (send 'delete u '(1 2 3))
= #(Union (b (1 2 3)) (1 2 3))
? (send 'delete u x)
= #(Union (b) b)
```

L'opération de fusion entre deux ensembles ordonnés, le résultat de la fusion étant stocké dans le premier argument:

```
    (DE {UNION}:MERGE (UNION1 UNION2)
        (IF ({UNION}:LIST UNION1)
            (WHEN ({UNION}:LIST UNION2)
                  (RPLACD ({UNION}:LAST UNION1) ({UNION}:LIST UNION2))
                  ({UNION}:LAST UNION1 ({UNION}:LAST UNION2)))
            (<- UNION1 UNION2))
        UNION1)
```

Pour aplatir un ensemble ordonné, c'est à dire effectuer l'opération:

(Union ... (Union a b c) ...) -> (Union ... a b c ...)

autant que possible:

```
    (DEMETHOD {UNION}:FLAT (UNION) (LIST)
        ({UNION}:CLEAR UNION)
        (WHILE LIST
               (IF (EQ (TYPE (CAR LIST)) 'UNION)
                   ({UNION}:MERGE UNION ({UNION}:FLAT (NEXTL LIST)))
                   ({UNION}:CONC UNION (NEXTL LIST))))
         UNION)
```

Exemples:

```
? (setq u1 (Union 'a 'b 'c))
= #(Union (a b c) c)
? (setq u2 (Union 1 2 3))
= #(Union (1 2 3) 3)
? (send 'merge u1 u2)
= #(Union (a b c 1 2 3) 3)
? u1
= #(Union (a b c 1 2 3) 3)
? u2
= #(Union (1 2 3) 3)
? (setq u3 (Union 'a1 'a2 'a3))
= #(Union (a1 a2 a3) a3)
? (send 'conc u1 u3)
= #(Union (a b c 1 2 3 #(Union (a1 a2 a3) a3)) #(Union (a1 a2 a3) a3))
? (send 'flat u1)
= #(Union (a b c 1 2 3 a1 a2 a3) a3)
```

```
(DEMETHOD {UNION}:MEMBER (UNION ITEM) (LIST)
    (TAG FOUND
         (WHILE LIST (WHEN (EQ (NEXTL LIST) ITEM) (EXIT FOUND T)))))

(DE {UNION}:POP-UP (UNION ITEM)
    (WHEN ({UNION}:MEMBER UNION ITEM)
          ({UNION}:DELETE UNION ITEM)
          ({UNION}:CONC UNION ITEM))
    UNION)

(DE {UNION}:POP-DOWN (UNION ITEM)
    (WHEN ({UNION}:MEMBER UNION ITEM)
          ({UNION}:DELETE UNION ITEM)
          ({UNION}:CONS UNION ITEM))
    UNION)
```

ANNEXE IV

Le Flip

Description de Transparents en CEYX

Jean-Marie Hullot, Bertrand Serlet

Ce chapitre est un exemple d'utilisation des arbres, c'est à dire des structures définies par deftree et defcons.

Toutes les fonctions décrites ici sont utilisables sous CEYX à condition d'avoir chargé le fichier leflip.ll de la bibliothèque CEYX, qui n'est d'ailleurs pas autre chose que ce chapitre.

```
? (ceyx-load leflip)
= leflip

   (CEYX-LOAD COORD)
```

IV.1 Introduction

Nous présentons une version CEYX du langage de descriptions de transparents conçu par Gilles Kahn. Nous définissons une syntaxe abstraite, des fonctions LISP de construction de de ces objets, et un interpréteur graphique permettant de tracer les flips sur plotter.

Le langage Flip est un petit langage géométrique permettant de découper une feuille de papier en bandes horizontales (l'opérateur horiz à nombre variable d'arguments), en bandes verticales (l'opérateur vertic à nombre variable d'arguments), en régions rectangulaires atomiques contenant du texte (l'opérateur aligner à nombre variable de chaînes de caractères arguments) et en régions rectangulaires atomiques contenant une diagonale d'une certaine couleur (l'opérateur diag à un argument, sa couleur). Chaque bande ou région possède une proportion qui permet de calculer sa taille propre dans sa bande mère, elle peut être encadrée d'une certaine couleur, peinte d'une certaine couleur, le texte peut y être écrit d'une certaine couleur. De plus on peut tourner les régions d'un multiple de 90°.

IV.2 Description du Langage

IV.2.1 Les Couleurs

```
(DEFMODEL COLOR (PREDICATE IS-COLOR NIL))
(DE IS-COLOR (X)
    (MEMQ X '(NIL ROUGE VERT BLEU NOIR BLANC
                  CITRON DORE BRUN VIOLET)))
(DEFVAR ROUGE 'ROUGE)
(DEFVAR VERT 'VERT)
(DEFVAR BLEU 'BLEU)
(DEFVAR NOIR 'NOIR)
(DEFVAR BLANC 'BLANC)
(DEFVAR CITRON 'CITRON)
(DEFVAR DORE 'DORE)
(DEFVAR BRUN 'BRUN)
(DEFVAR VIOLET 'VIOLET)
```

IV.3 Les Constructeurs du Langage

```
(DEFTREE FLIP
        (PROPORTION~FIX 1)
        (ROTATION~FIX 0)
        (CADRE~COLOR ())
        (TEXTE~COLOR ())
        (PEINTURE~COLOR ()))
```

Le Type: #:Tree:Flip	
Abréviation: Flip.	
Champs	
sons	•
tree-attributes	(Vector ...)
proportion	fix
rotation	fix
cadre	color
texte	color
peinture	color
Propriétés Sémantiques	
cadre	obj
display	(flip context)
display-flip	(flip context)
horiz-display	(sons context)
peinture	obj
proportion	obj
rotation	obj
texte	obj
vertic-display	(sons context)
Sous Modèles	
aligner	
couvrir	
diag	
horiz	
vertic	

(DEFCONS {FLIP}:HORIZ SONS~(LIST {FLIP}))

Le Type: #:Tree:Flip:horiz	
Abréviation: horiz.	
Fonction de Création	
horiz	sons
Champs	
sons	(List ...)
tree-attributes	(Vector ...)
proportion	fix
rotation	fix
cadre	color
texte	color
peinture	color
Propriétés Sémantiques	
display-flip	(flip context)
sons	obj

(DEFCONS {FLIP}:VERTIC SONS~(LIST {FLIP}))

Le Type: #:Tree:Flip:vertic	
Abréviation: vertic.	
Fonction de Création	
vertic	sons
Champs	
sons	(List ...)
tree-attributes	(Vector ...)
proportion	fix
rotation	fix
cadre	color
texte	color
peinture	color
Propriétés Sémantiques	
display-flip	(flip context)
sons	obj

(DEFCONS |FLIP|:COUVRIR SONS~(LIST |FLIP|))

Le Type: #:Tree:Flip:couvrir	
Abréviation: couvrir.	
Fonction de Création	
couvrir	sons
Champs	
sons	(List ...)
tree-attributes	(Vector ...)
proportion	fix
rotation	fix
cadre	color
texte	color
peinture	color
Propriétés Sémantiques	
display-flip	(flip context)
sons	obj

(DEFCONS |FLIP|:ALIGNER SONS~(LIST STRING))

Le Type: #:Tree:Flip:aligner	
Abréviation: aligner.	
Fonction de Création	
aligner	sons
Champs	
sons	(List ...)
tree-attributes	(Vector ...)
proportion	fix
rotation	fix
cadre	color
texte	color
peinture	color
Propriétés Sémantiques	
display-flip	(flip context)
sons	obj

```
(DEFCONS {FLIP}:DIAG (COLOR~COLOR))
```

Le Type: #:Tree:Flip:diag	
Abréviation: diag.	
Fonction de Création	
diag	(color)
Champs	
sons	(Vector ...)
color	color
tree-attributes	(Vector ...)
proportion	fix
rotation	fix
cadre	color
texte	color
peinture	color
Propriétés Sémantiques	
color	obj
display-flip	(flip context)

IV.3.1. Fonctions spécialisées Flip

```
(DE PEINDRE (COLOR FLIP)
    ({FLIP}:PEINTURE FLIP COLOR)
    FLIP)

(DE ENCADRER (COLOR FLIP)
    ({FLIP}:CADRE FLIP COLOR)
    FLIP)

(DE ECRIRE (COLOR FLIP)
    ({FLIP}:TEXTE FLIP COLOR)
    FLIP)

(DE TOURNE (N FLIP)
    ({FLIP}:ROTATION FLIP (PROD-ROT N ({FLIP}:ROTATION FLIP)))
    FLIP)

(SYNONYMQ TILT TOURNE)

(DE % (N FLIP)
    ({FLIP}:PROPORTION FLIP N)
    FLIP)
```

Fonction utilitaire:

```
(DEFMACRO PROD-ROT (N P) '(\ (+ ,N ,P) 4))
```

IV.4 Visualisation des Flips

Pour pouvoir être tracé sur papier ou sur écran, un flip doit pouvoir répondre au message 'display en relançant les fonctions display-frame, display-box, display-vector, et display-text avec les paramètres appropriés.

Le display se fait dans un certain contexte, et nous définissons donc d'abord le record flip-context.

IV.4.1 Définition du Contexte

Ce contexte utilise la boite rectangulaire dans laquelle on trace le flip. Ce type "Rect" est défini dans le package "coord"

```
(DEFRECORD FLIP-CONTEXT
            RECT~RECT                    ; cadre dans lequel on fait le display
            (ROT ~INTEGER 0)             ; 0/1/2/3 : la rotation courante
            (COLORTEXT~COLOR 'NOIR)      ; couleur courante du texte,par defaut
            )
```

Le Modèle: flip-context	
Champs	
rect	Rect
rot	integer
colortext	color
Propriétés Sémantiques	
colortext	obj
rect	obj
rot	obj

IV.4.2 Réponse au Message display

Cette fonction renvoie apres modification du contexte le message display-flip.

```
(DE {FLIP}:DISPLAY (FLIP CONTEXT)
    (WHEN ({FLIP}:PEINTURE FLIP)
          (DISPLAY-BOX ({FLIP-CONTEXT}:RECT CONTEXT) ({FLIP}:PEINTURE FLIP)))
    (OLET (FLIP-CONTEXT (ROT COLORTEXT) CONTEXT)
          ; on change le contexte
          ({FLIP-CONTEXT}:ROT CONTEXT (PROD-ROT ROT ({FLIP}:ROTATION FLIP)))
          ({FLIP-CONTEXT}:COLORTEXT CONTEXT (OR ({FLIP}:TEXTE FLIP) COLORTEXT))
          (SEND 'DISPLAY-FLIP FLIP CONTEXT)
          ; on retablit le contexte
          ({FLIP-CONTEXT}:ROT CONTEXT ROT) ({FLIP-CONTEXT}:COLORTEXT CONTEXT COLORTEXT))
    (WHEN ({FLIP}:CADRE FLIP)
          (DISPLAY-FRAME ({FLIP-CONTEXT}:RECT CONTEXT) ({FLIP}:CADRE FLIP))))
(DEMETHOD {VERTIC}:DISPLAY-FLIP (FLIP CONTEXT) (SONS)
    (SELECTQ ({FLIP-CONTEXT}:ROT CONTEXT)
             (0 ({FLIP}:HORIZ-DISPLAY SONS CONTEXT))
             (1 ({FLIP}:VERTIC-DISPLAY SONS CONTEXT))
             (2 ({FLIP}:HORIZ-DISPLAY (REVERSE SONS) CONTEXT))
             (3 ({FLIP}:VERTIC-DISPLAY (REVERSE SONS) CONTEXT))))
(DEMETHOD {HORIZ}:DISPLAY-FLIP (FLIP CONTEXT) (SONS)
    (SELECTQ ({FLIP-CONTEXT}:ROT CONTEXT)
             (0 ({FLIP}:VERTIC-DISPLAY SONS CONTEXT))
             (1 ({FLIP}:HORIZ-DISPLAY (REVERSE SONS) CONTEXT))
             (2 ({FLIP}:VERTIC-DISPLAY (REVERSE SONS) CONTEXT))
             (3 ({FLIP}:HORIZ-DISPLAY SONS CONTEXT))))
(DE {FLIP}:HORIZ-DISPLAY (SONS CONTEXT)
    (LET ((SIGMAPROP 0) (SON) (PROP 0) (RECT ({FLIP-CONTEXT}:RECT CONTEXT))
         (XORG) (XEXT) (SONS2 SONS))
         (SETQ XORG ({RECT}:XORG RECT) XEXT ({RECT}:XEXT RECT))
         (WHILE SONS2 (INCR SIGMAPROP ({FLIP}:PROPORTION (NEXTL SONS2))))
         (WHILE SONS
                ({RECT}:XORG RECT (SCALE-AFFINE XORG XEXT PROP SIGMAPROP))
                (SETQ SON (NEXTL SONS))
                (INCR PROP ({FLIP}:PROPORTION SON))
                ({RECT}:XEXT RECT (SCALE-AFFINE XORG XEXT PROP SIGMAPROP))
                (SEND 'DISPLAY SON CONTEXT))
         ({RECT}:XORG RECT XORG) ({RECT}:XEXT RECT XEXT)))
```

```
(DE {FLIP}:VERTIC-DISPLAY (SONS CONTEXT)
    (LET ((SIGMAPROP 0) (SON) (PROP 0) (RECT ({FLIP-CONTEXT}:RECT CONTEXT))
          (YORG) (YEXT) (SONS2 SONS))
         (SETQ YORG ({RECT}:YORG RECT) YEXT ({RECT}:YEXT RECT))
         (WHILE SONS2 (INCR SIGMAPROP ({FLIP}:PROPORTION (NEXTL SONS2))))
         (WHILE SONS
                ({RECT}:YORG RECT (SCALE-AFFINE YORG YEXT PROP SIGMAPROP))
                (SETQ SON (NEXTL SONS))
                (INCR PROP ({FLIP}:PROPORTION SON))
                ({RECT}:YEXT RECT (SCALE-AFFINE YORG YEXT PROP SIGMAPROP))
                (SEND 'DISPLAY SON CONTEXT))
         ({RECT}:YORG RECT YORG) ({RECT}:YEXT RECT YEXT)))

(DEMETHOD {COUVRIR}:DISPLAY-FLIP (FLIP CONTEXT) (SONS)
          (WHILE SONS (SEND 'DISPLAY (NEXTL SONS) CONTEXT)))

(DEMETHOD {ALIGNER}:DISPLAY-FLIP (FLIP CONTEXT) (SONS)
          (DISPLAY-TEXT ({FLIP-CONTEXT}:RECT CONTEXT) SONS
                        ({FLIP-CONTEXT}:ROT CONTEXT)
                        ({FLIP-CONTEXT}:COLORTEXT CONTEXT)))

(DE {DIAG}:DISPLAY-FLIP (FLIP CONTEXT)
    (LET ((RECT ({FLIP-CONTEXT}:RECT CONTEXT)))
         (IF (EVENP ({FLIP-CONTEXT}:ROT CONTEXT))
             (DISPLAY-VECTOR RECT ({DIAG}:COLOR FLIP))
             (DISPLAY-VECTOR
                 (VECT (COORD ({RECT}:XORG RECT) ({RECT}:YEXT RECT))
                       (COORD ({RECT}:XEXT RECT) ({RECT}:YORG RECT)))
                 ({DIAG}:COLOR FLIP)))))

(DE {FLIP}:DISPLAY-FLIP (FLIP CONTEXT) NIL)
```

Fonction auxiliaire qui devrait sans doute se trouver ailleurs **en** particulier dans le package coord.

```
(DE SCALE-AFFINE (X1 X2 PROP SIGMAPROP)
    (+ (SCALE X1 (- SIGMAPROP PROP) SIGMAPROP)
       (SCALE X2 PROP SIGMAPROP)))
```

IV.4.3 Display sur plotter ou sur écran

On utilise les fonctions display-init et display-end de l'output device virtuel.
Pour pouvoir plotter un flip sur hp, on charge la bibliothèque "plotter".
Format transparent:

```
(DE PLOT (FLIP)
    (CEYX-LOAD PLOTTER)
    (LET ((*OUTPUT-DEVICE* *PLOTTER*))
         (DISPLAY-INIT)
         (SEND 'DISPLAY FLIP (OMAKEQ {FLIP-CONTEXT}
                                     RECT (RECT (COORD 0 0)
                                                (COORD 1300 2000))))
         (DISPLAY-END)))
```

Format grande page pour grand plotter:

```
(DE BIGPLOT (FLIP)
    (CEYX-LOAD PLOTTER)
    (LET ((*OUTPUT-DEVICE* *PLOTTER*))
         (DISPLAY-INIT)
         (SEND 'DISPLAY FLIP (OMAKEQ {FLIP-CONTEXT}
                                     RECT (RECT (COORD 0 0)
                                                (COORD 3040 2000))))
         (DISPLAY-END)))
```

Format en continu papier rouleau sur grand plotter

```
(DE MULTIPLOT (FLIPS)
```

```
(CEYX-LOAD PLOTTER)
(SEND 'CUTTER-ENABLE *PLOTTER*)
(WHILE FLIPS (PLOT (NEXTL FLIPS)) (SEND 'ADVANCE-HALF-PAGE *PLOTTER*))
(SEND 'CUTTER-DISABLE *PLOTTER*))
```

ANNEXE V

Les Flux Linéaires

Jean-Marie Hullot

Ce chapitre est un exemple d'utilisation de classes auto-typées c'est à dire de types définis par deftclass. On y trouvera aussi des exemples d'utilisation de la construction defrule. On trouvera ici des exemples très caractéristiques de l'utilisation de la construction send.

Toutes les fonctions décrites ici sont utilisables sous CEYX à condition d'avoir chargé le fichier stream.ll de la bibliothèque CEYX, qui n'est d'ailleurs pas autre chose que ce chapitre:

```
? (ceyx-load stream)
= stream.ll
```

Le premier but suivi avec l'introduction des flux est la prise de contrôle sur les entrées sorties de LISP. Le principe de la manoeuvre est d'intercaler entre le clavier, sur lequel LE_LISP prend son input et l'écran, sur lequel il visualise son output, autant de machines virtuelles intermédiaires qu'il est nécessaire pour décomposer le traitement d'une application donnée.

```
(CEYX-LOAD DEFRULE)
```

V.1 Les Flux

Nous définissons une classe générique **Stream** possédant deux champs:
- *source*, qui pointe sur un objet fournissant de l'input à la stream, d'une manière qui reste à déterminer,
- *destination*, qui pointe sur un objet vers lequel la stream dirige son output, d'un manière qui reste elle-aussi à déterminer.

```
(DEFTCLASS STREAM
           SOURCE
           DESTINATION)
(DEFMAKE {STREAM} STREAM ())
```

Le Type: #:Tclass:Stream	
Abréviation: Stream.	
Fonction de Création	
Stream	()
Champs	
class-attributes	(Vector ...)
source	•
destination	•
Propriétés Sémantiques	
close	(stream)
connect	(stream1 stream2)
destination	obj
open	(stream)
source	obj
Sous Modèle	
LinearStream	

Connecter stream1 avec stream2 consiste à faire pointer la destination de stream1 sur stream2 et la source de stream2 sur stream1:

```
(DE {STREAM}:CONNECT (STREAM1 STREAM2)
    ({STREAM}:DESTINATION STREAM1 STREAM2)
    ({STREAM}:SOURCE STREAM2 STREAM1)
    T)
```

Les propriétés par défaut d'ouverture et de fermeture de flux:

```
(DE {STREAM}:OPEN (STREAM))
(DE {STREAM}:CLOSE (STREAM))
```

Exemples:

```
? (setq stream1 (Stream))
= #(#:Tclass:Stream . #[() ()])
? (setq stream2 (Stream))
= #(#:Tclass:Stream . #[() ()])
? (sendq connect stream1 stream2)
= t
? (sendq source stream1)
= ()
? (eq stream1 (sendq source stream2))
= t
? (eq stream2 (sendq destination stream1))
= t
? (sendq destination stream2)
= ()
```

Pour connecter temporairement une source ou une destination:

```
(DEFMACRO WITH-SOURCE (STREAM SOURCE . BODY)
    `(LET ((ST ,STREAM))
          (LET ((OSOURCE ({STREAM}:SOURCE ST)))
               (PROTECT
                 (PROGN
                    ({STREAM}:SOURCE ST ,SOURCE)
                    ,@BODY)
                  ({STREAM}:SOURCE ST OSOURCE)))))
(DEFMACRO WITH-DESTINATION (STREAM DESTINATION . BODY)
    `(LET ((ST ,STREAM))
          (LET ((ODESTINATION ({STREAM}:DESTINATION ST)))
               (PROTECT
                 (PROGN
                    ({STREAM}:DESTINATION ST ,DESTINATION)
                    ,@BODY)
                  ({STREAM}:DESTINATION ST ODESTINATION)))))
```

V.2 Les Flux Linéaires

V.2.1 Définition

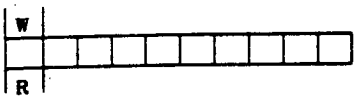
Jusqu'à une période récente, les entrées sorties ont essentiellement été orientées caractères. Pour traiter ce type d'entrées sorties CEYX propose la notion de flux linéaire.

```
(DEFTCLASS {STREAM}:LINEARSTREAM)
```

Nous allons définir par la suite plusieurs modèles de flux linéaires qui vont tous suivre une même idée de construction, l'implémentation étant elle particulière à chaque cas. L'idée conductrice est la suivante: une **LinearStream** possède un buffer de stockage de dimension fixe ou illimitée selon les cas et maintient sur ce buffer deux pointeurs:

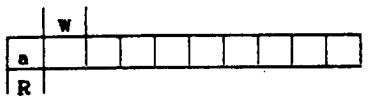
- un pointeur dit de lecture ***inpos***,
- un pointeur dit d'écriture ***outpos***.

Nous représentons picturalement le buffer et ses deux pointeurs, dans le cas d'un flux de caractères de taille 10 par:



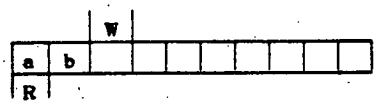
où R désigne le pointeur de lecture et W le pointeur d'écriture. Pour écrire un caractère dans ce flux, il suffit de lui envoyer le message ***new*** avec le code ascii du caractère comme argument. Ceci a a pour effet d'introduire ce caractère dans le flux à la position outpos et de décaler outpos d'un cran vers la droite

```
? (sendq new stream #/a)
= 97
```



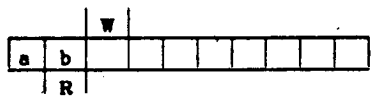
Ecrivons un nouveau caractère dans ce flux:

```
? (sendq new stream #/b)
= 98
```



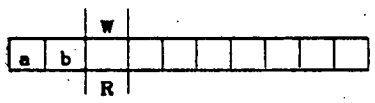
Pour lire un caractère dans le flux il suffit de lui envoyer le message ***next*** sans arguments. Le code ascii du caractère à la place inpos dans le flux est alors renvoyé en valeur et le pointeur de lecture est avancé d'un cran:

```
? (sendq next stream)
= 97
```



Un nouvel appel à ***next*** nous met dans la situation:

```
? (sendq next stream)
= 98
```

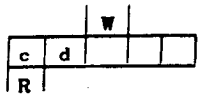


Le pointeur de lecture est alors égal au pointeur d'écriture. Nous dirons que le flux est vide, dans le sens ou il n'y a plus rien à lire.

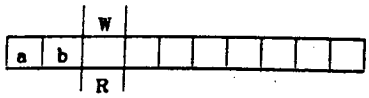
```
? (sendq null stream)
= 2
```

Comme il n'y a plus rien à lire, le flux va réagir au message ***next*** en relayant ce message sur sa source. Nous introduisons pour ce faire un second flux stream1, tel que la source de stream vaille stream1. Et nous écrivons les caractères c et d dans ce flux:

STREAM1



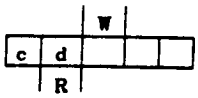
STREAM



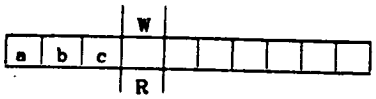
Si nous envoyons maintenant ***next*** à stream, ce message est relayé sur sa source stream1, et le résultat est écrit dans stream puis lu.

```
.? (sendq next stream)
= 99
```

STREAM1



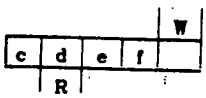
STREAM



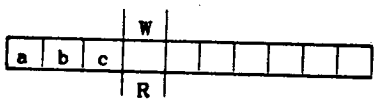
Une autre opération intéressante sur les flux linéaires, consiste à tranférer l'information contenue dans un flux dans un autre flux. Cette opération est réalisée en envoyant le message ***bitstream*** aux deux flux stream1 et stream. L'opération ***bitstream*** est implémentée comme une règle CEYX, de manière que la façon dont s'effectue le transfert dépende à la foix du type de stream et de stream1, permettant ainsi une implémentation optimale pour chaque situation du transfert. Par défaut, le tranfert s'effectue au caractère par caractère.

```
? (sendq new stream1 #/e)
= 101
? (sendq new stream1 #/f)
= 102
```

STREAM1



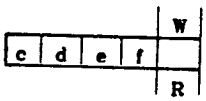
STREAM



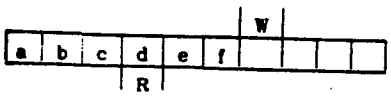
Nous transférons le contenu de stream1 dans stream:

```
? (sendq bitstream stream1 stream)
= 4
```

STREAM1



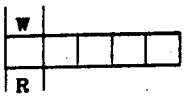
STREAM



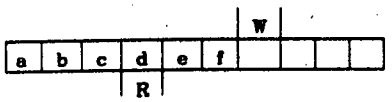
Pour nettoyer un flux, c'est à dire pour remettre à zéro les pointeurs de lecture et d'écriture, on lui envoie le message *clear*:

```
? (sendq clear stream1)
= 0
```

STREAM1



STREAM



Nous définissons un certain nombre d'autres opérations sur les flux linéaires:

- *flush* qui consiste à transférer le contenu d'un flux dans sa destination, puis à remettre à zéro son buffer en lui envoyant le message clear;
- *next?* qui permet de tester s'il y a quelque chose à lire dans le flux ou dans un des flus chaînés en source et, si oui, de le lire.
- *peek* qui fait un *next* sans déplacer le pointeur de lecture.
- *return* qui remat à zéro le pointeur de lecture et permet ainsi de recommencer une lecture.

Le Type: #:Tclass:Stream:LinearStream	
Abréviation: LinearStream.	
Champs	
class-attributes	(Vector ...)
source	•
destination	•
Propriétés Sémantiques	
bltstream	(x y)
bltstream!1	(x y)
bltstream!2	(x y)
bltstream!3	(x y)
bltstream!4	(stream dest)
bltstream!5	(stream dest)
destination	obj
eos	(stream)
flush	(stream)
next?	(stream)
class-attributes	obj
source	obj
Sous Modèles	
CharStream	
InChannel	
InputBuffer	
ListStream	
OutChannel	
OutputBuffer	

Les bltstream!n sont des sémantiqes intermédiaires engendrées par le defrule.

```
(DE {LINEARSTREAM}:FLUSH (STREAM)
    (SENDQ BLTSTREAM STREAM ({STREAM}:DESTINATION STREAM))
    (SENDQ CLEAR STREAM))
```

En cas de dédordement, on fait par défaut un flush.

```
(DE {LINEARSTREAM}:EOS (STREAM)
    (SENDQ FLUSH STREAM))

(DEFRULE BLTSTREAM (X~{LINEARSTREAM} Y~{LINEARSTREAM})
    (UNTIL (SENDQ NULL X)
           (SENDQ NEW Y (SENDQ NEXT X))))
```

Pour voir s'il y a quelque chose à lire et si oui le lire:

```
(DE {LINEARSTREAM}:NEXT? (STREAM)
    (IF (SENDQ NULL STREAM)
        (SENDQ NEXT? ({STREAM}:SOURCE STREAM))
        (SENDQ NEXT STREAM)))
```

V.2.2 Les Cas Limites

Les objets qu'on retrouve souvent aux extrêmités de chaînes de flux linéaires:

- () joue le rôle de /dev/null

```
(DE {NULL}:NEW (NULL VAL) ())

(DE {NULL}:NEXT (NULL)
    (SYSERROR '{NULL}:NEXT "Nothing in nil" ()))

(DE {NULL}:FLUSH (NULL))

(DEFRULE BLTSTREAM (X~NULL Y~{LINEARSTREAM})
    NIL)
```

- le canal d'entrée, tyi, tys dans le jargon des flux.

Le Type: #:Tclass:Stream:LinearStream:InChannel	
Abréviation: InChannel.	
Fonction de Création	
InChannel	()
Champs	
class-attributes	(Vector ...)
source	•
destination	•
Propriétés Sémantiques	
bltstream	(x y)
destination	obj
flush	(channel)
next	(channel)
next?	(channel)
class-attributes	obj
source	obj

```
(DEFTCLASS {LINEARSTREAM}:INCHANNEL)
(DEFMAKE {INCHANNEL} INCHANNEL ())
(DE {INCHANNEL}:NEXT (CHANNEL) (TYI))
(DE {INCHANNEL}:NEXT? (CHANNEL) (TYS))
(DE {INCHANNEL}:FLUSH (CHANNEL) ())

(DEFRULE BLTSTREAM (X~{INCHANNEL} Y~{LINEARSTREAM})
        NIL)
```

- le canal de sortie, tyo, tyflush dans le jargon des flux.

Le Type: #:Tclass:Stream:LinearStream:OutChannel	
Abréviation: OutChannel.	
Fonction de Création	
OutChannel	()
Champs	
class-attributes	(Vector ...)
source	•
destination	•
Propriétés Sémantiques	
destination	obj
flush	(channel)
new	(channel val)
class-attributes	obj
source	obj

```
(DEFTCLASS {LINEARSTREAM}:OUTCHANNEL)
(DEFMAKE {OUTCHANNEL} OUTCHANNEL ())
(DE {OUTCHANNEL}:NEW (CHANNEL VAL) (TYO VAL))
(DE {OUTCHANNEL}:FLUSH (CHANNEL) (TYFLUSH))
(DE {OUTCHANNEL}:NEWLINE (CHANNEL)
    (WHEN #:SYSTEM:REAL-TERMINAL-FLAG (TYO #\RETURN))
    (TYO #\LF)
    (TYFLUSH))
```

V.2.3 Les Flux de Caractères

Il s'agit de flux dont le buffer est implémenté comme un chaîne de caractères.

Le Type: #:Tclass:Stream:LinearStream:CharStream	
Abréviation: CharStream.	
Fonction de Création	
CharStream	(size)
Champs	
class-attributes	(Vector ...)
source	•
destination	•
string	•
size	fix
inpos	fix
outpos	fix
Propriétés Sémantiques	
clear	(stream)
current	(stream)
init	(stream size)
inpos	obj
new	(stream val)
next	(stream)
null	(stream)
outpos	obj
peek	(stream)
return	(stream)
size	obj
string	obj

```
(DEFTCLASS {LINEARSTREAM}:CHARSTREAM
                STRING
                SIZE~FIX
                (INPOS~FIX 0)
                (OUTPOS~FIX 0))
(DE CHARSTREAM (SIZE)
```

```
    (SENDQ INIT (OMAKEQ CHARSTREAM) SIZE))
(DEFMAKE |CHARSTREAM| CHARSTREAM)
(DE |CHARSTREAM|:INIT (STREAM SIZE)
    (OCHANGEQ CHARSTREAM STREAM
              STRING (MAKESTRING SIZE #\SP)
              SIZE SIZE))

(DE |CHARSTREAM|:CLEAR (STREAM)
    (FILLSTRING (|CHARSTREAM|:STRING STREAM)
                0
                #\SP
                (|CHARSTREAM|:SIZE STREAM))
    (|CHARSTREAM|:INPOS STREAM 0)
    (|CHARSTREAM|:OUTPOS STREAM 0))
(DE |CHARSTREAM|:NULL (STREAM)
    (= (|CHARSTREAM|:INPOS STREAM) (|CHARSTREAM|:OUTPOS STREAM)))
(DE |CHARSTREAM|:CURRENT (STREAM)
    (CHRNTH (|CHARSTREAM|:INPOS STREAM) (|CHARSTREAM|:STRING STREAM)))
(DE |CHARSTREAM|:NEXT (STREAM)
    (IFN (= (|CHARSTREAM|:INPOS STREAM)
            (|CHARSTREAM|:OUTPOS STREAM))
         (PROG1
            (|CHARSTREAM|:CURRENT STREAM)
            (|CHARSTREAM|:INPOS STREAM (1+ (|CHARSTREAM|:INPOS STREAM))))
         (|CHARSTREAM|:NEW STREAM
                          (SENDQ NEXT (|STREAM|:SOURCE STREAM)))
         (|CHARSTREAM|:NEXT STREAM)))
(DE |CHARSTREAM|:PEEK (STREAM)
    (PROG1
      (|CHARSTREAM|:NEXT STREAM)
      (|CHARSTREAM|:INPOS STREAM (1- (|CHARSTREAM|:INPOS STREAM)))))
(DE |CHARSTREAM|:RETURN (STREAM)
    (|CHARSTREAM|:INPOS STREAM 0))
(DE |CHARSTREAM|:NEW (STREAM VAL)
    (COND
      ((= (|CHARSTREAM|:OUTPOS STREAM)
          (|CHARSTREAM|:SIZE STREAM))
       (SENDQ EOS STREAM)
       (|CHARSTREAM|:NEW STREAM VAL))
      (T (CHRSET (|CHARSTREAM|:OUTPOS STREAM)
                 (|CHARSTREAM|:STRING STREAM)
                 VAL)
         (|CHARSTREAM|:OUTPOS STREAM (1+ (|CHARSTREAM|:OUTPOS STREAM)))
         VAL)))
```

V.2.4 Les Flux sous Forme de Liste

Ici le buffer est implémenté sous forme de liste. Ces flux peuvent donc stocker un nombre arbitraire d'objets. De plus il peuvent stocker des objets quelconques et pas seulement des caractères.

Le Type: #:Tclass:Stream:LinearStream:ListStream	
Abréviation: ListStream.	
Fonction de Création	
ListStream	()
Champs	
class-attributes	(Vector ...)
source	•
destination	•
list	•
inpos	•
outpos	•
Propriétés Sémantiques	
clear	(stream)
current	(stream)
init	(stream)
inpos	obj
list	obj
new	(stream val)
next	(stream)
null	(stream)
outpos	obj
peek	(stream)
return	(stream)

```
(DEFTCLASS {LINEARSTREAM}:LISTSTREAM
                LIST
                INPOS
                OUTPOS)

(DE LISTSTREAM ()
    (SENDQ INIT (OMAKEQ {LISTSTREAM})))

(DEFMAKE {LISTSTREAM} LISTSTREAM)

(DE {LISTSTREAM}:INIT (STREAM)
    (LET ((BUFFER (LIST ())))
         (OCHANGEQ {LISTSTREAM} STREAM
                LIST BUFFER
                INPOS BUFFER
                OUTPOS BUFFER)))

(DEMETHOD {LISTSTREAM}:CLEAR (STREAM) (LIST)
    (RPLACD LIST ())
    ({LISTSTREAM}:OUTPOS STREAM LIST)
    ({LISTSTREAM}:INPOS STREAM LIST))

(DE {LISTSTREAM}:NULL (STREAM)
    (EQ ({LISTSTREAM}:INPOS STREAM)
        ({LISTSTREAM}:OUTPOS STREAM)))

(DE {LISTSTREAM}:CURRENT (STREAM)
    (CAR ({LISTSTREAM}:INPOS STREAM)))

(DE {LISTSTREAM}:NEXT (STREAM)
    (IFN (EQ ({LISTSTREAM}:INPOS STREAM)
             ({LISTSTREAM}:OUTPOS STREAM))
         (PROG1
            (CAR ({LISTSTREAM}:INPOS STREAM))
            ({LISTSTREAM}:INPOS STREAM (CDR ({LISTSTREAM}:INPOS STREAM))))
         ({LISTSTREAM}:NEW STREAM
                        (SENDQ NEXT ({STREAM}:SOURCE STREAM)))
```

```
            ({LISTSTREAM}:NEXT STREAM)))

    (DEMETHOD {LISTSTREAM}:PEEK (STREAM) (INPOS)
       (PROG1
          ({LISTSTREAM}:NEXT STREAM)
          ({LISTSTREAM}:INPOS STREAM INPOS)))

    (DE {LISTSTREAM}:RETURN (STREAM)
        ({LISTSTREAM}:INPOS STREAM ({LISTSTREAM}:LIST STREAM)))

    (DE {LISTSTREAM}:NEW (STREAM VAL)
        (RPLACA ({LISTSTREAM}:OUTPOS STREAM) VAL)
        (RPLACD ({LISTSTREAM}:OUTPOS STREAM) (LIST ()))
        ({LISTSTREAM}:OUTPOS STREAM (CDR ({LISTSTREAM}:OUTPOS STREAM))))
```

V.2.5 Le Tampon d'Entrée LE_LISP

Pour manipuler le tampon d'entrée LE_LISP comme un flux linéaire:

Le Type: #:Tclass:Stream:LinearStream:InputBuffer	
Abréviation: InputBuffer.	
Fonction de Création	
InputBuffer	()
Champs	
class-attributes	(Vector ...)
source	•
destination	•
inpos	fix
inmax	fix
Propriétés Sémantiques	
bltstream	(stream dest)
bol	(stream)
clear	(stream)
close	(stream)
inmax	obj
inpos	obj
new	(stream val)
newl	(stream l)
next	(stream)
null	(stream)
open	(stream)
peek	(stream)
return	(stream)

```
(DEFTCLASS {LINEARSTREAM}:INPUTBUFFER
                (INPOS~FIX 0)
                (INMAX~FIX 0))

(DEFMAKE {INPUTBUFFER} INPUTBUFFER)

(DE INPUTBUFFER () (OMAKEQ {INPUTBUFFER} SOURCE (INCHANNEL)))

(DE {INPUTBUFFER}:OPEN (STREAM)
    (INPOS 0) (INMAX 0))

(DE {INPUTBUFFER}:CLOSE (STREAM)
    (INPOS 0) (INMAX 0))

(DE {INPUTBUFFER}:CLEAR (STREAM)
    ({INPUTBUFFER}:INPOS STREAM (INPOS 0))
    ({INPUTBUFFER}:INMAX STREAM (INMAX 0)))

(DE {INPUTBUFFER}:NULL (STREAM)
    (= (INMAX) (INPOS)))

(DE {INPUTBUFFER}:NEXT (STREAM)
    (IFN (= (INPOS) (INMAX))
         (PROG1 (INBUF (INPOS))
                (INPOS (1+ (INPOS))))
         ({INPUTBUFFER}:NEW STREAM (SENDQ NEXT ({STREAM}:SOURCE STREAM)))
         ({INPUTBUFFER}:NEXT STREAM)))

(DE {INPUTBUFFER}:PEEK (STREAM)
    (PROG1
      ({INPUTBUFFER}:NEXT STREAM)
      (INPOS (1- (INPOS)))))

(DE {INPUTBUFFER}:RETURN (STREAM)
    (INPOS 0))

(DE {INPUTBUFFER}:NEW (STREAM VAL)
    (CHRSET (INMAX) (INBUF) VAL)
    (INMAX (1+ (INMAX))))
```

Pour faire l'ouput d'une liste de codes ascii dans le tampon d'entrée:

```
(DE {INPUTBUFFER}:NEWL (STREAM L)
    (WHILE L ({INPUTBUFFER}:NEW STREAM (NEXTL L))))

(DEFRULE BLTSTREAM (STREAM~INPUTBUFFER DEST~{LINEARSTREAM})
    (LET ((N (INPOS)))
      (UNTIL (= N (INMAX))
            (SENDQ NEW DEST (INBUF N))
            (INPOS (INMAX))
            (INCR N))))

(DE {INPUTBUFFER}:BOL (STREAM)
    ({INPUTBUFFER}:PEEK STREAM))
```

V.2.6 Le Tampon de Sortie LE_LISP

Pour manipuler le tampon de sortie LE_LISP comme un flux linéaire:

Le Type: #:Tclass:Stream:LinearStream:OutputBuffer	
Abréviation: OutputBuffer.	
Fonction de Création	
OutputBuffer	()
Champs	
class-attributes	(Vector ...)
source	*
destination	o
outpos	fix
lmargin	fix
rmargin	fix
Propriétés Sémantiques	
bltstream	(stream dest)
clear	(stream)
close	(stream)
eol	(stream)
getl	(obuf)
lmargin	obj
new	(stream val)
null	(stream)
open	(stream)
outpos	obj
rmargin	obj

```
(DEFTCLASS |LINEARSTREAM|:OUTPUTBUFFER
                (OUTPOS~FIX 0)
                (LMARGIN~FIX 0)
                (RMARGIN~FIX 79))

(DEFMAKE |OUTPUTBUFFER| OUTPUTBUFFER)

(DE OUTPUTBUFFER () (OMAKEQ |OUTPUTBUFFER| DESTINATION (OUTCHANNEL)))

(DE |OUTPUTBUFFER|:OPEN (STREAM)
    (LMARGIN (|OUTPUTBUFFER|:LMARGIN STREAM))
    (RMARGIN (|OUTPUTBUFFER|:RMARGIN STREAM))
    (CLRBUFOUT))

(DE CLRBUFOUT ()
    ;pourquoi le fillstring merde-t-il sur outbuf?
    (LET ((N 0)) (REPEAT (RMARGIN) (OUTBUF N #\SP) (INCR N)))
    (OUTPOS (LMARGIN)))

(DE |OUTPUTBUFFER|:CLOSE (STREAM)
    ((|OUTPUTBUFFER|:LMARGIN STREAM (LMARGIN))
    ((|OUTPUTBUFFER|:RMARGIN STREAM (RMARGIN)))

(DE |OUTPUTBUFFER|:CLEAR (STREAM)
    (CLRBUFOUT))

(DE |OUTPUTBUFFER|:NULL (STREAM)
    (= (OUTPOS) 0))

(DE |OUTPUTBUFFER|:NEW (STREAM VAL)
    (COND
       ((= (OUTPOS) (RMARGIN))
        (EOL)
        ((|OUTPUTBUFFER|:NEW STREAM VAL))
       (T (OUTBUF (OUTPOS) VAL)
          (OUTPOS (1+ (OUTPOS))))))

(DEFRULE BLTSTREAM (STREAM~OUTPUTBUFFER DEST~LINEARSTREAM)
    (LET ((N 0))
         (UNTIL (= N (OUTPOS))
```

```
                (SENDQ NEW DEST (OUTBUF N))
                (INCR N))))
```

Pour récupérer le contenu du buffer de sortie comme une liste:

```
(DE {OUTPUTBUFFER}:GETL (OBUF)
    (LET ((L ())
          (N 0))
         (UNTIL (= N (OUTPOS))
                (NEWL L (OUTBUF N))
                (INCR N))
         (NREVERSE L)))

(DE {OUTPUTBUFFER}:EOL (STREAM)
    (SENDQ FLUSH STREAM)
    (SENDQ NEWLINE ({STREAM}:DESTINATION STREAM)))
```

V.3 Flux d'Entrée, de Sortie, d'Erreur

```
(DEFVAR {LISP}:OUTPUTBUFFER (OUTPUTBUFFER))
(DEFVAR {LISP}:INPUTBUFFER  (INPUTBUFFER))
```

Nous conservons dans des variables globales un flux d'entrée courant, un flux de sortie courant, et un flux d'erreur courant. Ils sont initialisés avec le buffer d'entrée LE_LISP pour le premier et avec le buffer de sortie LeLISP pour les deux autres.

```
(DEFVAR {CEYXSYS}:INSTREAM {LISP}:INPUTBUFFER)
(DEFVAR {CEYXSYS}:OUTSTREAM {LISP}:OUTPUTBUFFER)
(DEFVAR {CEYXSYS}:ERRORSTREAM {LISP}:OUTPUTBUFFER)

(DE {CEYX}:BOL () (SENDQ BOL (INSTREAM)))
(DE {CEYX}:EOL () (SENDQ EOL (OUTSTREAM)))
```

Pour modifier le flux d'entrée courant, on utilisera la fonction instream qui, appelée avec un argument qui doit être une stream:
- ferme l'ancien flux courant,
- ouvre le flux passé en argument,
- et le met dans la valeur de la variable {CeyxSys}:instream.

Appelée sans argument, cette fonction ramène en valeur le flux d'entrée courant, c'est à dire la valeur de la variable {CeyxSys}:instream.

```
(DE INSTREAM ARG
    (IFN ARG {CEYXSYS}:INSTREAM
             (SENDQ CLOSE {CEYXSYS}:INSTREAM)
             (SETQ {CEYXSYS}:INSTREAM (CAR ARG))
             (SENDQ OPEN {CEYXSYS}:INSTREAM)
             {CEYXSYS}:INSTREAM))

(DE OUTSTREAM ARG
    (IFN ARG {CEYXSYS}:OUTSTREAM
             (SENDQ CLOSE {CEYXSYS}:OUTSTREAM)
             (SETQ {CEYXSYS}:OUTSTREAM (CAR ARG))
             (SENDQ OPEN {CEYXSYS}:OUTSTREAM)
             {CEYXSYS}:OUTSTREAM))

(DE ERRORSTREAM ARG
    (IFN ARG {CEYXSYS}:ERRORSTREAM
             (SENDQ CLOSE {CEYXSYS}:ERRORSTREAM)
             (SETQ {CEYXSYS}:ERRORSTREAM (CAR ARG))
             (SENDQ OPEN {CEYXSYS}:ERRORSTREAM)
             {CEYXSYS}:ERRORSTREAM))
```

Index des fonctions LISP

Table des matières

Table des matières

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique