

IRIA

UNITE DE RECHERCHE
IRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105

78153 Le Chesnay Cedex
France

Tel (1) 39 63 55 11

Rapports Techniques

N° 72

YAFOOL: ENCORE UN LANGAGE OBJET À BASE DE FRAMES!

Version 2.1

Roland DUCOURNAU
Joël QUINQUETON

Août 1986

Y A F O O L :
ENCORE UN LANGAGE OBJET A BASE DE FRAMES !
VERSION 2.1

Roland Ducournau
Joel Quinqueton
SEMA.METRA / INRIA

Juillet 1985



PAPIER RECLUPERE ET RECYCLE

Roland Ducournau

SEMA.METRA
16-18 rue Barbès
92126 MONTROUSE CEDEX

Joel Quinqueton

INRIA Projet CLOREC-SHOM
Domaine de Voluceau
78153 LE CHESNAY CEDEX

YAF00L: ENCORE UN LANGAGE OBJET A BASE DE FRAMES !

Résumé: YAF00L est un langage orienté objets, de type "frames", pour des applications en Représentation des Connaissances (Intelligence Artificielle et systèmes experts). Ecrit en Le_Lisp, il s'agit moins d'un véritable langage, que d'un ensemble de primitives permettant, comme en CEYX, en LOOPS ou dans les Flavors, un style de programmation orientée objets, simultanément avec une programmation procédurale classique.

YAF00L: YET ANOTHER FRAME-BASED OBJECT ORIENTED LANGUAGE !

Abstract: YAF00L is an object oriented language, based on "frames", and has been developed for applications in the field of Knowledge Representation (Artificial Intelligence and Expert Systems). It is written in Le_Lisp, and is rather a package of functions than a real language. This package allows simultaneously an object-oriented programming and a classical one (in Le_Lisp), as CEYX, LOOPS, or the Flavors.

Ce travail a été réalisé par SEMA.METRA et INRIA initialement pour le compte du Centre de Programmation de la Marine (marché DTCN No A8476265004057578). Que soient ici remerciés, pour leurs sarcasmes constructifs dans le rôle de cobayes, Anne Rouillé (INRIA) et Eric Demonchaux, Rémi Sourisse et Jean-Loup Zinger (SEMA).

Y A F 0 0 L 2.1

Première partie: les grands principes.

1. Présentation générale.
2. Les concepts de YAFOOL.
3. L'héritage.

Deuxième partie: le noyau dur.

4. Implémentation.
5. Accès en lecture.
6. Accès en écriture.
7. Comportements.
8. Contrôle d'environnement et communication.
9. Autres primitives.

Troisième partie: le noyau mou.

10. Principes généraux.
11. Espace dual des slots et facettes.
12. Univers.
13. Macros et applicateurs d'accès.
14. Univers noyau.

Quatrième partie: les extensions.

15. Les extensions.
16. Temporalité.
17. La Video.
18. Exemples.

Cinquième partie: manuel d'utilisation.

19. Programmer en YAFOOL.

Annexes.

20. Toutes les fonctions.
21. Chargement et utilisation.
22. Perspectives.
23. Bibliographie.

Avertissement.

Ce rapport est à la fois un rapport technique et un manuel de référence et d'utilisation. Deux ordres de lecture sont ainsi possibles: à la fin de la première partie, voire des 2 premiers chapitres, il est possible, et même souhaitable pour un lecteur non averti, de passer directement à la cinquième partie, "Programmer en YAFOOL".

Il nécessite une bonne connaissance de LISP en général et de Le_Lisp en particulier: [Winston 84] et [Chailloux 85] sont, respectivement, une bonne introduction au premier et le manuel de référence du second. Sauf oubli, toutes les fonctions citées non référencées sont des fonctions Le_Lisp.

PREMIERE PARTIE
LES GRANDS PRINCIPES

YAF00L 2.1

1. YAFOOL: "YET ANOTHER FRAME-BASED OBJECT-ORIENTED LANGUAGE".

YAFOOL est un langage orienté objets développé pour des applications de systèmes experts en maquettage de programme, simulation, apprentissage, langage naturel et éditeur de règles "intelligent".

Deux de ces applications ont fortement influencé YAFOOL: la simulation est la cause des développements temporels du langage. L'éditeur a motivé une partie de l'implémentation: celle qui fait que tout "concept" relatif à un objet (attribut, facette, réflexe etc..) est, lui aussi, un objet.

Plus que d'un véritable langage, il s'agit d'un ensemble de fonctions qui autorise une programmation de type objet (à la Smalltalk) aussi bien que de type "frame". L'accès à LISP est total et immédiat, puisqu'il ne s'agit que de fonctions LISP, sans qu'il n'ait été défini d'interprète ou de compilateur spécifique.

En ce sens, ses références seraient les Flavors de ZetaLisp [Moon 81] ou LOOPS de Interlisp-D [Bobrow 83].

Ce système est écrit en Le_Lisp Version 15: dans ce rapport, la connaissance de LISP en général (1) et de ce dialecte en particulier [Chailloux 85], sera supposée connue.

1.1. Quatre paradigmes de programmation.

Pour reprendre la classification de Bobrow et Stefik [Bobrow 83], il existe actuellement 4 paradigmes (non exclusifs) de programmation, généralement admis par la communauté informatique: les programmations orientées vers les procédures, les objets, les données et les règles.

1.1.1. Procédures.

Le paradigme procédural est encore dominant dans l'informatique d'aujourd'hui. Tous les langages classiques, dont LISP, en relèvent. Deux sortes d'entités sont considérées: les procédures et les données. Les premières sont actives, les secondes passives.

Les programmes sont organisés en procédures, les effets de bords se produisant lorsque 2 procédures partageant la même donnée et la modifient séparément. La programmation structurée est censée limiter ces effets de bords.

1.1.2. Objets.

La programmation orientée objets est principalement issue de SIMULA et de SMALLTALK [Goldberg 83]. Dans ce type de programmation, données et procédures ne sont pas séparées, mais forment conjointement des entités appelées objets.

Les objets ont des procédures locales (les comportements) et des données locales (les attributs). Toutes les actions de ces langages peuvent se décrire comme la passation de messages entre objets, chaque objet produisant sa propre interprétation du message reçu.

(1) [Winston 84] est une bonne introduction à LISP et aux frames. Les crochets [] renvoient à la bibliographie [83] et à la numérotation des paragraphes.

Un trait important de ces langages est l'existence d'un graphe d'héritage qui permet de structurer hiérarchiquement les objets en classes et sous-classes, chaque objet pouvant alors hériter des propriétés (comportements et attributs) des classes auxquelles il appartient.

1.1.3. Données.

Dans les 2 paradigmes précédents, toute action se fait par invocation directe (de procédure ou de message). La programmation orientée vers les données (2) établit, entre données et procédures, un mode d'invocation original: des actions-réflexes sont déclenchées par un accès spécifique (lecture, écriture ou autre) à une donnée. Il s'agit, littéralement, de "réflexes". La programmation orientée données est principalement représentée par les frames [Minsky 75] et les valeurs actives des Flavors [Moon 81].

1.1.4. Règles.

Dans la programmation orientée vers les règles, dite aussi programmation logique, le comportement du système est dicté par des ensembles de couples conditions-actions ou prémisses-conclusions.

Ces ensembles de règles jouent le même rôle que les sous-programmes dans la programmation procédurale. Dans chacun d'eux, l'invocation d'une règle est guidée par filtrage (pattern matching) sur les données.

1.2. YAFOOL = langage orienté objets et données (LOOD).

YAFOOL réunit les paradigmes de programmation orientées, objets et données.

1.2.1. Objets.

Du premier, il prend la notion d'objet, entité de base du langage. Chaque objet possède des données locales, les attributs, et des procédures locales, les comportements. Attributs et comportements seront appelés, de façon générique, propriétés ou slots (le terme anglo-saxon usuel). Ces objets sont structurés hiérarchiquement par un lien d'héritage, est-un, qui représente à la fois une relation d'inclusion (de sous-classe à classe) et d'appartenance (d'instance à classe).

Des langages orientés objets (LOO), et plus particulièrement de leur variantes langages d'acteurs (3), YAFOOL a aussi retenu la notion de continuation, ainsi que les idées de bufferisation ("tamponnisation" ?) de messages.

1.2.2. Frames.

Du paradigme données, YAFOOL retient la notion de réflexe et les principes de base d'implémentation des frames (4). Les frames sont des

(2) Il ne faut pas confondre cette programmation orientée vers les données, avec la programmation dirigée par les données ("data-driven programming"), classique en Lisp, qui peut être considérée comme un embryon de programmation orientée objets. Dans la programmation dirigée par les données, des attachements procéduraux permettent de définir pour des symboles des propriétés fonctionnelles.

(3) Les langages d'acteurs sont une variété de LOO, issus de PLASMA [Hewitt 73 et Durieux 83] principalement orientés vers des applications temps réelles et le parallélisme. La communication entre acteurs est l'élément le plus novateur de ces langages: les messages peuvent y être "bufferisés" et précéder leur propre "continuation", c'est-à-dire ce qui doit les suivre.

entités assez semblables aux objets, puisqu'elles possèdent des données locales, les attributs, dont la signification est identique à celle des attributs des objets. Par contre, les frames ne possèdent pas en propre de procédures locales: ce sont leurs attributs qui les possèdent (d'où leur nom de valeurs actives dans les Flavors). Ces procédures locales sont les réflexes.

De façon classique, les frames sont implémentées uniformément sous forme de liste d'association à 2 niveaux, définissant des triplets frame-slot-facette. Les facettes définissant des modalités sur les attributs (modalités de valeurs, de contraintes, de réflexes etc..).

Enfin, comme les objets, les frames possèdent des mécanismes d'héritage.

1.2.3. Frames de YAFOOL.

A partir de maintenant, nous entendrons par frame, la fusion des descriptions des 2 paragraphes précédents: ce seront donc les frames classiques, avec des comportements, des continuations et des bufferisations de message.

1.2.4. Règles.

YAFOOL ne possède pour l'instant aucun des traits spécifiques de la programmation logique. Il a été utilisé dans une application de Système Expert [Ducourneau 85], dans laquelle les règles de production étaient écrites en LISLOG (Prolog écrit en LISP [Bourgault 83]), la base de faits étant constituée d'objets YAFOOL. L'éditeur développé en YAFOOL traduisait les règles d'un "langage quasi naturel" en clauses LISLOG.

Nous travaillons actuellement au développement d'un moteur d'inférence en logique d'ordre 1, écrit en YAFOOL, pour une base de faits YAFOOL.

1.2.5. Procédures.

Le langage présenté ici, qui se définit plus comme un "paquet de primitives" que comme un véritable langage, préserve entièrement la programmation procédurale de LISP. L'utilisateur YAFOOL peut donc définir librement des fonctions LISP, et réciproquement, l'utilisateur LISP peut développer une partie de son application en YAFOOL.

(4) Le mot frame n'a pas encore trouvé d'équivalent satisfaisant en français: schéma et stéréotype ont été proposés sans qu'aucun consensus ne se dégage. Nous garderons donc le mot anglais qui, dans la littérature, implique bien l'idée de programmation, à la fois, orientée objets et dirigée par les données. Enfin, contrairement à l'anglais, le français autorise une vaste méditation sur le sexe des frames !

2. LES CONCEPTS DU LANGAGE.

Ce chapitre décrit brièvement la philosophie de YAFOOL, c'est à dire l'optique dans laquelle ont été implémentées les intentions définies au chapitre précédent.

2.1. Organisation du langage.

On peut distinguer 2 niveaux nettement distincts.

2.1.1. Noyau "dur".

Il contient les primitives du système, c'est-à-dire toutes les fonctions d'accès élémentaire aux triplets frame-slot-facette (création, lecture, écriture, effacement, etc...), de gestion des mécanismes d'héritage, de déclenchement des attachements procéduraux (réflexes et comportements), auxquelles il faut ajouter les primitives de continuation, de buffering des messages, des fonctions de marquage et de liaison dynamique d'objets et de valeurs.

2.1.2. Noyau "mou".

Le noyau dur peut se suffire: il possède toutes les caractéristiques d'un LOOD normalement constitué. C'est un noyau effectivement exploitable (à quelques réserves près !).

Mais, l'ambition de YAFOOL, est de présenter, avec des performances identiques, un langage plus puissant, exploitant mieux "sa" propre connaissance en autorisant une syntaxe plus souple et plus elliptique. Enfin, les extensions du langage doivent être aisées, mais ne pas s'imposer à tous les utilisateurs, grâce à une modularité totale.

Ces exigences sont prises en charge par le "noyau mou", qui définit l'ensemble des objets (au sens LOO) et des choix (arbitraires ?) d'utilisation standard du noyau dur.

Ce "noyau mou" intervient donc au niveau de la syntaxe (arbitraire), des concepts (originaux), des objets système et des utilitaires. Il est extensible ou modifiable, par l'utilisateur, à chacun de ces niveaux.

2.1.3. Interactions noyau dur / noyau mou.

Elle est faible, mais non nulle, puisque des choix d'implémentation profonde et de primitives de haut niveau dépendent typiquement de certains choix du noyau mou. En ce sens, le noyau dur seul comporterait quelques erreurs.

Ces points d'interaction seront signalés au fur et à mesure.

2.2. Utilisation de YAFOOL.

2.2.1. Niveaux utilisateurs.

Il y a 3 niveaux d'utilisation de YAFOOL. L'utilisateur de base est un utilisateur du noyau mou et n'a besoin d'une primitive du noyau dur qu'exceptionnellement. L'utilisateur évolué, lui, étend le langage, et doit connaître le noyau dur aussi bien que certains détails plus ardues du

noyau mou. Enfin, l'utilisateur de haut niveau a besoin d'un accès au source de YAFOOL.

2.2.2. Documentation YAFOOL.

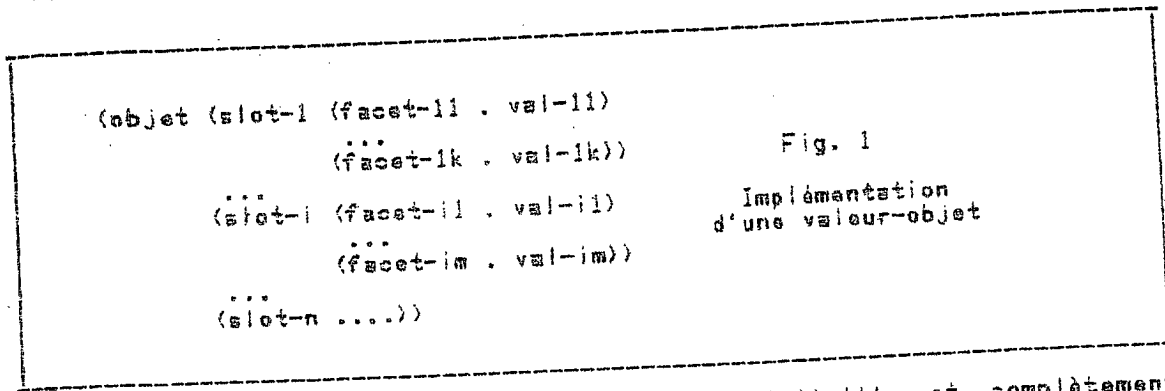
Ce rapport se place au niveau de l'utilisateur évolué. Un accès à l'utilisateur de base est possible, à partir du chapitre Programmer en YAFOOL.

2.3. Les concepts de YAFOOL.

Ce paragraphe décrit les concepts de base des noyaux "dur" et "mou", dont le noyau mou n'est qu'une instance particulière. Ces concepts renvoient à des détails d'implémentation aussi bien qu'à des idées classiques en LOG(D). Ils sont accompagnés de quelques grands principes.

2.3.1. Principe d'implémentation.

Un objet est un symbole autovaleué ayant une valeur d'objet. Cette valeur d'objet est une liste, dont le CAR est le nom de l'objet et le CDR une A-liste à 2 niveaux. Les clés du premier niveaux sont appelées slots, celles du second, facettes.



Le nombre des facettes ou des slots est illimité, et complètement dynamique: à tout instant, il est toujours possible d'en ajouter ou d'en enlever. Un objet peut ainsi n'actualiser qu'une partie de ses propriétés potentielles (1). L'unité de base d'information est donc le triplet frame-slot-facette qui permet de retrouver la valeur associée.

2.3.2. Principe de dualisation.

Presque tout concept YAFOOL est un objet, au sens de la définition précédente. En particulier, tout slot et toute facette est (doit être) un objet. En fait, ce principe s'applique surtout à eux: leurs grandes classes forment l'essentiel des objets systèmes.

Slots et facettes peuvent donc être hiérarchisés.

2.3.3. Principaux objets duaux.

Le noyau dur impose la présence d'un certain nombre de facettes et de

(1) Contrairement aux LOG comme SMALLTALK, ou bien basés sur les "records-types" (à la Lisp, Pascal, Cobol ou PLI).

types de slots. Chacun de ces types de slots possède une facette standard, la facette représentant l'essence du slot, qui contient sa "valeur" (2). Au triplet frame-slot-facette, on peut donc souvent substituer la paire frame-slot, où la facette sous-entendue est la facette standard du slot.

2.3.3.1. Attribut.

Ils représentent les données, ou variables, locales de l'objet. Leur facette standard, celle qui contient la valeur de l'attribut, est value.

2.3.3.2. Comportement.

Ils sont de 2 types, méthodes et applicateurs (3), de facettes standards respectives méthode et applic. Ce sont les procédures locales de l'objet. Les premières sont bien connues: c'est la base de la programmation orientée objet. Les seconds sont inspirés de MERING [Ferber 83]: ils possèdent leurs propres réflexes, et permettent des attachements procéduraux à 2 dimensions et non à une seule. Ce sont (pour l'instant ?) des concepts assez systèmes.

2.3.3.3. Réflexe.

C'est la base de la programmation orientée données: il en existe 2 sortes. Les réflexes standards, liés à des accès spécifiques à un couple frame-attribut: chacun d'eux est déclenché par une (des) primitive(s) spécifique(s): ils sont donc en nombre fini (dans une version donnée). Il y en a 3: si-besoin, pour calculer la valeur absente d'un attribut, si-ajout et si-enlève en cas d'écriture ou d'effacement de la valeur d'un attribut.

Les réflexes d'applicateurs, qui leur sont spécifiques, sont en nombre illimité.

N.B. De façon classique dans les LOD, on distingue des réflexes a priori et des réflexes a posteriori: les premiers s'exécutent avant le corps de la primitive, les seconds après. En YAFOOL, tous les réflexes sont a posteriori (bien que le cas des si-besoin soit un peu discutable). Des réflexes a priori d'écriture (si-possible) sont à l'étude.

2.3.3.4. Facette.

D'un point de vue implémentation, c'est la clé de deuxième niveau des A-listes des valeurs objets.

D'un point de vue sémantique, elles servent à typer les slots (attributs, méthodes ou applicateurs), et, essentiellement pour les attributs, représentent des modalités: bien qu'elles ne fassent pas partie d'un système minimum, on peut distinguer des facettes de valeur (autre valeur, les facettes de valeur par défaut, initiale, d'exception, de valeur fonctionnelle etc..), les facettes de filtre ou de contraintes de valeur (prédicat devant être vérifié par toute valeur d'un couple objet-attribut).

C'est essentiellement par elles que le langage peut s'étendre.

(2) Le terme de valeur est ambigu: pour une facette, c'est celle du triplet frame / slot / facette. Pour un slot celle du triplet frame / slot / facette-standard-du-slot.

(3) Les méthodes sont les comportements classiques. L'idée des applicateurs est due à MERING [Ferber 83].

R.B. D'une certaine manière, les réflexes sont des cas particuliers de facettes: ce sont aussi des clés de 2-ième niveau.

2.3.4. Héritage.

Il représente l'un des éléments clés de la programmation orientée objet: l'héritage est le mécanisme qui permet la factorisation de la connaissance entre objets. Il en existe plusieurs types: sans nous y attarder, précisons néanmoins que les mécanismes d'héritage de YAF00L fonctionnent par délégation [Liebermann 86]: si un objet ne peut répondre à un message, il délègue ce message à ses successeurs (ancêtres) dans le graphe d'héritage. L'héritage peut être multiple, avec un algorithme de recherche et de résolution des conflits original, qui préserve l'ordre du graphe d'héritage. Enfin, des exceptions à l'héritage sont possibles.

Il n'est pas fait de distinction entre relation d'instanciation (entre une instance et sa classe) et relation de spécification (entre classe et sur-classe). En particulier, la distinction classe/instance est sans objet.

2.3.5. Autres concepts.

Les LOO, en particulier leur variante langages d'acteurs, ont introduit deux structures de contrôle particulières: les continuations d'un processus, et la buffering des messages. Elles concourent toutes deux à l'élaboration de techniques de programmation asynchrone et de type agenda. Aucune de ces structures ne connaît de définition aussi précise que celles d'héritage ou de comportements. L'implémentation proposée ici n'est donc qu'une interprétation très liée au contexte du langage.

2.3.5.1. Continuations.

La continuation est la structure de contrôle par laquelle un processus (un message) indique par quoi il doit se "continuer".

2.3.5.2. Retardements.

Ordinairement, un envoi de message (plus généralement un appel fonctionnel) donne le contrôle à l'objet récepteur, l'émetteur attendant la réponse pour continuer. Il est souvent possible (nécessaire) de différer l'exécution d'un tel message. C'est le rôle des retardements.

3. L'HERITAGE, LES LIENS ET LA HIERARCHIE.

Les mécanismes d'héritage permettent de partager, ou factoriser, les propriétés des objets. La relation d'héritage peut s'interpréter de diverses manières, logique, ensembliste, conceptuelle [Brachman 83].

Les mécanismes d'héritage en YAFOOL sont réalisés par un attribut particulier, est-un (1), et des fonctions spécifiques: l'héritage est "cabi", et non pas redéfinissable par chaque objet. Seul le graphe d'héritage est modifiable.

3.1. Interprétation.

3.1.1. Interprétation ensembliste.

Une interprétation ensembliste de l'héritage nécessite 2 relations: une relation d'inclusion et une relation d'appartenance. Il y a alors 2 catégories, non exclusives, d'objets: les classes et les instances. Les premières sont reliées entre elles par la relation d'inclusion, les secondes aux premières par celle d'appartenance. Le propre d'une classe est alors d'être génératrice d'instances.

3.1.2. YAFOOL: interprétation conceptuelle.

En YAFOOL, les 2 relations ensemblistes sont confondues. Bien qu'il puisse être intéressant de revenir localement à une interprétation ensembliste, le lien est-un doit plutôt être considéré comme une relation de spécification / généralisation conceptuelle.

3.1.3. Lien est-un et instanciement.

Bien que le graphe défini par le lien (2) est-un puisse contenir des cycles, il faut le comprendre comme un arbre dont la racine représente l'abstraction la plus grande de l'univers décrit. Plus on descend (3) dans l'arbre, plus l'abstraction diminue, chaque objet étant une instanciement de son (ses) lien(s) est-un.

Il semble cependant impossible de tracer une limite précise entre objet abstrait et instances concrètes, chaque objet pouvant à son tour devenir l'abstraction d'un objet plus concret. Dans l'interprétation ensembliste, on pourrait dire que toute feuille du graphe d'héritage est à la fois un élément, le sous-ensemble réduit à cet élément ou bien un sous-ensemble vide qu'une future instanciement viendrait "remplir".

N.B. Le paradigme classe / instance, prioritaire dans de nombreux langages objets (SMALLTALK et record-types) est ici assez inopérant. Si l'on définit une instance comme un objet sans instance (les feuilles de l'arbre), et une classe comme un objet avec instance(s), toute instance peut, à tout moment, devenir une classe.

(1) C'est le IS-A ou AKO (a kind of) anglo-saxon.

(2) Pour fixer le vocabulaire, un lien sera un attribut à valeur, unique ou multiple, dans l'ensemble des objets de l'univers.

(3) La descente graphique ou généalogique, de la racine aux feuilles, est une remontée métaphoriquement botanique !

3.1.4. Héritage et autres liens.

D'autres liens peuvent aussi comporter des connotations d'héritage de propriétés: faut-il alors les inclure dans le lien `est-un` ou leur laisser leur autonomie ?

Soit un univers décrivant des automobiles et des individus.

```
(objet-ideal)
(automobile-ideal (est-un (value objet-ideal)))
(2-CV-citroen (est-un (value automobile-ideal)))
(individu-ideal (est-un (value objet-ideal)))
(M-Dupont (est-un (value individu-ideal)))
```

Le problème se complique si l'on veut introduire la voiture de M-Dupont ..

```
(auto-de-M-dupont (est-un (value automobile-ideal)))
```

.. si l'on connaît son modèle, dont ce véhicule particulier va hériter les propriétés, doit-t-on avoir:

```
(auto-de-M-Dupont (est-un (value 2-CV-citroen)))
ou bien
(auto-de-M-Dupont (est-un (value automobile-ideal)
(modèle (value 2-CV-citroen))) ?
```

Rajouter des liens, et le code qui permet de s'en servir, ou augmenter les liens "est-un", en se reposant donc sur le code général qui ne s'applique peut-être qu'imparfaitement ? C'est un dilemme fréquent pour lequel il ne semble y avoir que des cas d'espèces. On reverra d'ailleurs ce problème avec les liens temporels.

3.1.5. Liens et réseau sémantique.

Les liens permettent donc de considérer un système de frames comme un réseau sémantique; c'est-à-dire un graphe dont les arêtes sont étiquetées par le nom du lien, ce dernier étant lui-même un noeud (4).

(4) Contrairement à ce que semble en donner Chouraqui [Chouraqui 81], les frames ne sont qu'une implémentation dont les possibilités s'étendent aux réseaux sémantiques.

3.2. Héritage et délégation.

[Liebermann 85 et 86] distingue deux mécanismes de partage de la connaissance dans les LOO: l'héritage et la délégation. Sans entrer dans les détails, disons qu'il oppose ainsi des mécanismes statiques de copie à des mécanismes dynamiques de délégation. Dans le premier cas, toute modification du père, après la naissance du fils, ne modifie pas ce dernier. Dans la délégation, au contraire, la modification d'une propriété du père, que le fils ne possède pas en propre, modifie celle du fils.

L'héritage concerne l'inné, la délégation l'acquis.

De plus, dans une véritable délégation, chaque objet peut lui-même redéfinir, pour chacune de ses propriétés, sa conception de la délégation: la délégation idéale n'est pas cablée.

YAFOOL pratique de la délégation cablée (5).

N.B. Le terme d'héritage en YAFOOL est utilisé dans son sens général de partage de la connaissance, et non dans le sens spécifique de Liebermann, en opposition à délégation.

3.3. La multiplicité de l'héritage.

L'héritage étant multiple, son graphe n'est plus un arbre, mais un treillis. Il possède néanmoins toujours une unique racine.

Quelle relation existe-t-il alors entre 2 successeurs ("pères") d'un même objet, ou plutôt entre les 2 relations entre cet objet et ses 2 pères? La plupart des LOO à héritage multiple considèrent que cette relation est une relation d'ordre: si l'on dit "A est un B et C", on sous-entend "A est un B puis C". Cet ordre dans les héritages sera appelé multiplicité, ou priorité de la multiplicité. Il est difficile de lui donner une interprétation très précise: elle la trouvera de manière très pragmatique avec l'algorithme de recherche.

N.B. Dans toutes les figures, la multiplicité se lira de gauche à droite.

3.4. Les exceptions à l'héritage: la facette sauf.

L'un des enjeux de la représentation des connaissances est celui des propriétés par défaut et de leurs exceptions.

Il y a ainsi de nombreuses façons de représenter le fait que les oiseaux volent mais que l'autruche, qui en est un, ne vole pas.

Une façon de faire consiste à définir un attribut capacité-de-voler à valeur oui/non: l'incapacité de l'autruche masque ainsi la capacité de l'oiseau.

```

(oiseau (est-un (value animal))
         (capacite-de-voler (value . oui)))
(autruche (est-un (value oiseau))
           (capacite-de-voler (value . non)))

```

Par contre si l'on veut définir un objet animal-volant, il faut alors pouvoir dire que l'autruche est-un oiseau mais n'est-pas-un animal-volant; c'est le rôle de la facette sauf.

```
(animal (est-un (value objet-ideal)))
(animal-volant (est-un (value animal)))
(oiseau (est-un (value animal-volant)))
(autruche (est-un (value oiseau animal)
                  (sauf animal-volant)))
```

Les exceptions portent sur tous les ancêtres de la facette sauf: il s'agit de chemins et non de noeuds. Il faut donc rappeler que l'autruche, bien que n'étant pas un animal-volant, est un animal.

La prise en compte par les mécanismes d'héritage de cette facette permet, au prix d'une recherche dans le graphe un peu plus coûteuse, une plus grande généralité: tous les attributs des animaux-volants sont gérés en même temps, et ces mécanismes peuvent s'appliquer à tous les liens.

3.5. Recherche dans la hiérarchie.

La recherche dans la hiérarchie sert à résoudre les conflits issus de la présence de la même propriété dans plusieurs éléments de la hiérarchie d'un objet: de laquelle doit-il hériter ?

3.5.1. Algorithmes de recherche.

Toute recherche dans la hiérarchie consiste à remonter dans le graphe à partir d'un noeud: l'objet récepteur du message. Comme le graphe n'est pas un arbre, mais un treillis, deux méthodes sont a-priori possibles: en profondeur ou en largeur d'abord.

3.5.1.1. Algorithmes naïfs.

La justification de la première (6), comme de la seconde (7), réside dans le fait que les liens multiples sont présumés ordonnés, par la relation de multiplicité. La largeur d'abord permet de reculer au maximum dans la recherche les objets les plus abstraits, sous réserve que le treillis soit "équilibré", ce que ne fait pas la profondeur d'abord.

Dans l'un et l'autre cas, la recherche s'arrête (le long d'un chemin) lorsqu'est rencontré un noeud qui est déjà apparu: des occurrences multiples, seule la première est conservée.

En fait, la raison principale du choix de tels algorithmes réside dans leur simplicité.

(5) Et non pas (de la dénotation) basée (Rémi Sourisse).

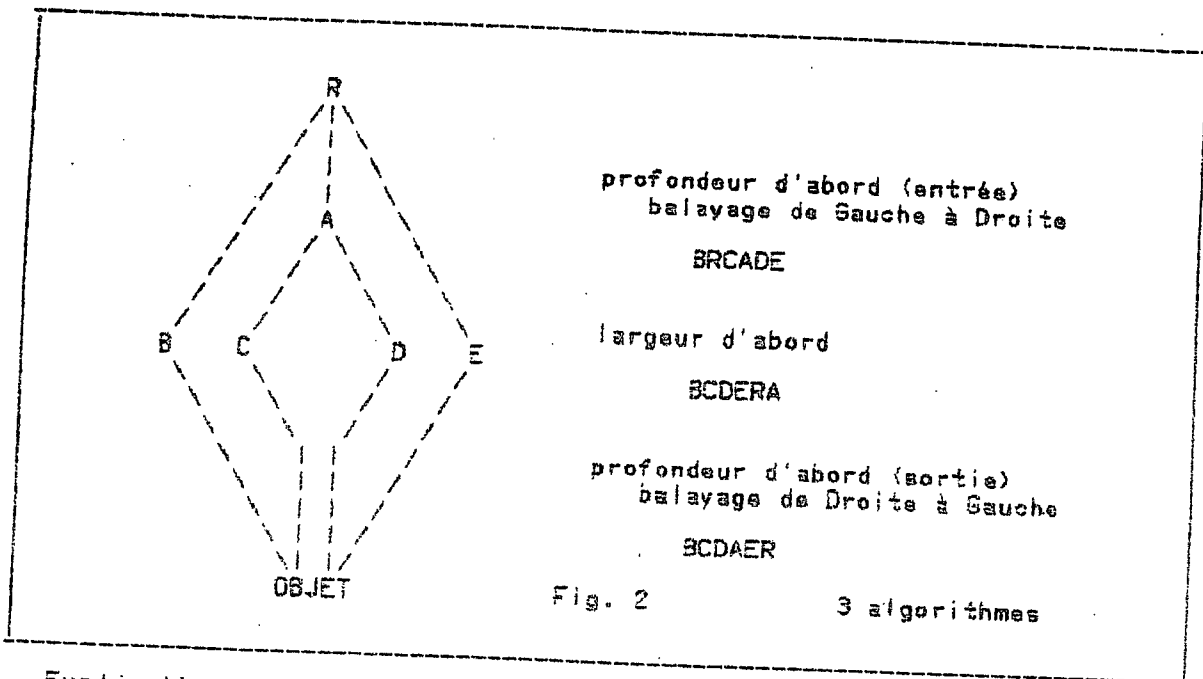
(6) Adoptée par la quasi totalité des LO à héritage multiple.

(7) Adoptée plus rarement, entre autres par PERINS (Ferber ESJ).

3.5.1.2. L'algorithme de YAFODL.

Un troisième mode de recherche a finalement été adopté dans une tentative de réconciliation des deux points de vue.

L'ordre de parcours est l'inverse de l'ordre de sortie de pile d'un parcours en profondeur d'abord (8), en prenant les liens multiples dans l'ordre inverse de leur importance.



Explication: toute recherche en profondeur d'abord, qui se limite aux noeuds nouveaux, fournit 2 ordres totaux sur l'ensemble des sommets du graphe: l'ordre d'entrée dans un noeud (entrée de pile), et l'ordre de sortie (outstack). L'algorithme utilise l'inverse de l'ordre de sortie, en balayant les successeurs d'un sommet dans l'ordre inverse de leur priorité.

3.5.2. Ordres partiels et totaux, extension linéaire.

Pour justifier cet algorithme, et le caractère aberrant des 2 autres, il faut remarquer que le graphe d'héritage forme un treillis, c'est-à-dire un ordre partiel. Or tout parcours de ce graphe, s'il s'arrête aux sommets déjà rencontrés produit un ordre total sur les sommets du graphe. Le moins que l'on puisse demander à cet ordre total, est d'être une extension linéaire de l'ordre partiel de la hiérarchie, que parcourir un ordre le respecte.

Or il est démontré [Habib 86] que l'ordre obtenu par cet algorithme est une extension linéaire de l'ordre de l'héritage.

(8) La méthode utilisée au début consistait à faire de la profondeur d'abord en gardant, pour toute occurrence multiple d'un encêtre, la dernière, et non la première. Il est apparu, à la lecture de [Habib 86], que cet algorithme des "dernières occurrences" donnait le même résultat que celui de "outstack" (Habib), mais en moins performant.

3.5.3. Développements.

Il serait tentant d'étendre ce résultat à l'"ordre" de la multiplicité de l'héritage, mais 2 problèmes se posent.

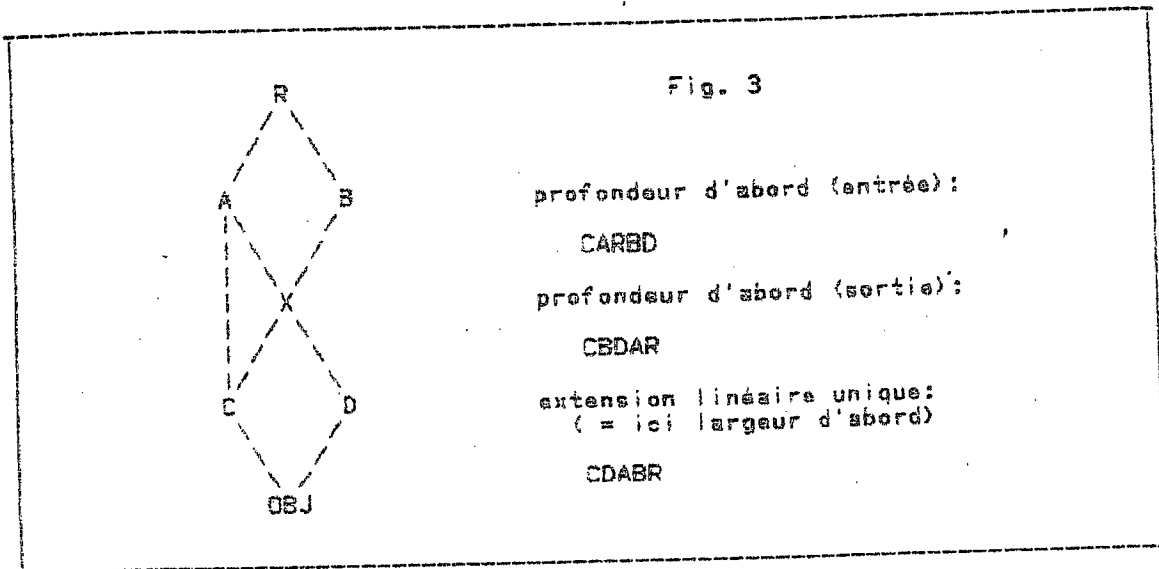
3.5.3.1. La multiplicité n'est pas un ordre.

En effet, ce n'est pas un ordre, même partiel, car il peut toujours y avoir 'OBJET' d'héritage ordonné C et B, et si X est un OBJET puis un OBJET', la relation de multiplicité n'est évidemment pas un ordre, dans la hiérarchie de X.

De plus, rien n'empêche les 2 ordres, si la multiplicité en est un, d'être contradictoires.

3.5.3.2. Contre exemple.

Même si les 2 ordres sont compatibles, le résultat n'est pas acquis: l'ordre obtenu n'est pas forcément une extension linéaire des 2 ordres (fig. 3).



Il faudrait donc trouver un autre algorithme, qui produise une extension linéaire des 2 ordres, s'ils sont compatibles, et sinon, uniquement de l'ordre de l'héritage. Cet algorithme devrait bien entendu rester suffisamment performant.

3.5.4. Les exceptions.

En cas de présence de facette sauf l'algorithme utilisé est identique, tous les chemins parcourus (en profondeur d'abord), s'arrêtant à la première exception.

On voit que cette méthode ne gère pas les cas de redondance contradictoire (9), ni ne détecte les ambiguïtés pures (fig. 4 et 5).

(9) Mais faut-il les gérer ? Pour plus de détail voir [Touratzky 84].

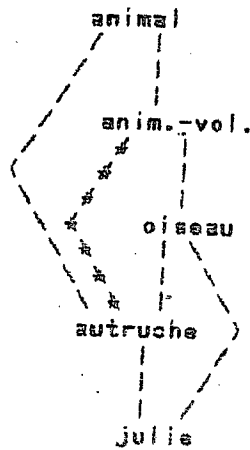


Fig. 4

Redondance contradictoire.
Julie est-elle un anim.-vol.?

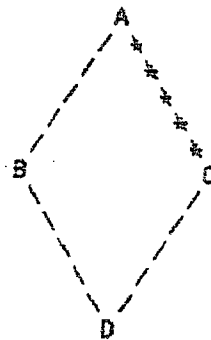


Fig. 5

Ambiguïté.
D est-il un A?

pour YAFOOL, oui !

3.5.5. La facette Herit.

Pour des raisons de performances, une facette herit a été rajoutée pour le lien est-un: elle contient la fermeture transitive ordonnée de ce lien pour l'objet récepteur et est calculée au premier besoin.

En cas de modification du lien est-un, cette facette est effacée.

Par la suite, la hiérarchie d'un objet sera entendue comme la liste formée de cet objet et de tous les éléments de la facette herit de son lien est-un.

3.5.6. L'environnement d'appel

Au cours de la recherche dans la hiérarchie, l'environnement d'appel — l'objet récepteur du message et les arguments d'accès — est sauvegardé dans le quadruplet (FRAME+ / SLOT+ / FACET+ / VALUE+) qui représentent respectivement l'objet (10), l'attribut, la facette et la valeur en argument de cet accès.

La connaissance de ce quadruplet n'est nécessaire que dans les activations de réflexes ou de comportements. C'est lui qui est en jeu dans les continuations. Ces questions d'environnement seront développées au chapitre [29].

3.5.7. Hiérarchies définies par d'autres liens.

Les mécanismes d'héritage décrits ici sont utilisables pour n'importe

(10) FRAME+ est ainsi l'équivalent du SELF de SMALLTALK, du THIS de SIMULA ou du % de PERL.

quel lien, mais leur déclenchement automatique n'a lieu que pour est-un.

Toutes les primitives mettant en jeu la hiérarchie du lien est-un, le font par référence à la variable #:YAF00L:EST-UN (cf. troisième partie). Il est ainsi possible à tout moment de modifier dynamiquement le lien utilisé par l'héritage standard.

Enfin, il est envisagé d'étendre les mécanismes d'héritage à d'autres liens, simultanément avec le lien est-un: dans l'exemple automobile, l'héritage pourrait se faire par modèle et par est-un.

N.B. On trouvera au paragraphe [#15.1] 5 facettes supplémentaires, like, init, herit-from, herit-by et value-of qui permettent à peu de frais une extension spécifique de l'héritage, pour un attribut particulier.

DEUXIEME PARTIE

LE NOYAU DUR

YAF001 2.1

4. PRINCIPES GÉNÉRAUX D'IMPLEMENTATION.

4.1. Liste d'association.

La structure de base de l'implémentation est la A-liste à 2 niveaux (figure en [2.3.1]).

4.1.1. Valeur objet.

En Le Lisp, elle est dans le champ O-VAL des symboles, accessible par la variable-fonction OBJVAL. Dans d'autres dialectes, il faudrait la mettre dans la P-liste du symbole, sous une propriété quelconque.

Le premier accès en écriture à un symbole dont la valeur-objet est NIL, se traduit par une autoévaluation du symbole et une initialisation de sa valeur objet par une liste réduite au symbole lui-même.

4.1.2. Remise du slot en tête.

Tout accès à un slot place celui-ci en tête de la liste de slots de l'objet. L'ordre des slots dans la A-liste suit donc statistiquement celui de la fréquence de leur usage. En particulier, il n'est pas beaucoup plus coûteux de faire 2 accès successifs au même slot, que de mémoriser dans une variable auxiliaire la valeur obtenue au premier appel.

4.2. Marquage.

Certaines primitives de YAFOOL utilisent le "bit invisible" et les TCONS de Le Lisp, qui permettent de marquer n'importe quel CONS. Toutes les primitives de marquage de YAFOOL ont un nom formé à partir de "MARK" (l'utilisateur peut ainsi en retrouver la liste par la fonction LHOBLIST).

Dans la mesure du possible, leur emploi sera cité, pour éviter les conflits avec l'utilisateur.

4.2.1. Des objets.

Les objets sont marqués par le CONS de leur valeur d'objet.

4.2.2. Des slots.

Les slots sont marqués par le CONS du slot dans la A-liste des slots.

4.3. Multiplicité et atomicité des valeurs.

Les valeurs des triplets frame-slot-facette peuvent être uniques ou multiples suivant les slots et les facettes. Une valeur multiple est une liste (éventuellement vide) de valeurs élémentaires. Une valeur unique est une valeur élémentaire. Certaines facettes (clés de 2-ième niveau) sont toujours à valeur unique (comportements par exemple), d'autres toujours à valeur multiple (réflexes), d'autres encore (value) uniques ou multiples suivant le slot.

Chacune des valeurs élémentaires peut être elle-même un atome ou une liste.

La multiplicité des valeurs est un concept logique, alors que l'atomicité est un concept physique.

Dans la pratique, il y a confusion possible entre valeur multiple et valeur non atomique.

Certaines primitives peuvent avoir un résultat aberrant sur les valeurs élémentaires non atomiques: FADD* et FREM* considèrent que leur argument est une valeur multiple s'il est non atomique. Il faut donc rajouter un niveau de parenthèses si l'on veut manipuler une unique valeur élémentaire non atomique. Sur ce sujet, voir le paragraphe [#11.2.1] et [#6.3.2] sur les valeurs (VALUE*) dans les réflexes d'écriture..

4.4. Dualisation des réflexes.

Au paragraphe [#2.3.3.3], il a été dit que les réflexes pouvaient être considérés comme des cas particuliers de facettes. Dans l'implémentation réelle, il est fait une permutation circulaire du triplet frame-slot-réflexe au triplet slot-réflexe-frame. Deux avantages très nets en découlent.

4.4.1. Répartition des accès entre frames et slots.

En général, les couples frame-slot qui ont des réflexes, n'ont que des réflexes: par cette permutation, le slot disparaît de la frame. Comme les objets duaux sont assez petits, il y a ainsi une meilleure répartition des informations dans l'ensemble des objets, avec des parcours moyens de A-liste plus courts.

4.4.2. Accès unique au slot.

Tous les déclenchements de réflexes (sauf si-besoin) commencent par une collecte de ceux-ci le long de la hiérarchie de l'objet receveur. La permutation fait donc passer le nombre d'accès à une A-liste de $2xN$ à $N+1$, où N est la taille de la hiérarchie. En particulier, un seul accès suffit pour savoir si le slot a des réflexes ou pas: il n'est donc pas coûteux de les déclencher systématiquement.

4.5. Valeur NIL.

Comme pour les P-listes, et de façon classique en LISP, il y a confusion entre l'absence de valeur et la valeur NIL. La plupart des primitives traitent NIL comme une absence d'action.

4.6. Les primitives.

4.6.1. Fonctions décrites.

Ne sont pas décrites dans ce rapport les primitives d'accès à la valeur-objet d'un symbole, ni les primitives d'accès à un slot, c'est-à-dire à l'ensemble des facettes du couple objet-slot, ni celles d'accès à une facette dans un slot.

4.6.2. Convention de nom.

La plupart des primitives ont un nom commençant par la lettre F.

4.6.2.1. Convention de la facette standard.

Toutes les primitives prenant comme argument le triplet objet-slot-facette sont doublées par des macros qui permettent l'élimination de la

facette si celle-ci est la facette standard du slot. (cf [#11.5.1] pour une description plus précise de ce mécanisme).

Les noms de ces fonctions et macros correspondantes sont identiques, au suffixe "-V" près des fonctions: à la fonction fxxx-V correspond donc la macro fxxx, que l'utilisateur (ainsi que ce rapport) utilisera toujours, même en cas de présence explicite de la facette.

4.6.2.2. Mode d'accès.

L'héritage et les réflexes offrent une combinatoire importante de primitives d'accès. Celles-ci se différencient, pour un même type d'accès, lecture par exemple, par leur suffixe, le radical étant mnémotechnique, quand c'est possible identique à celui des fonctions d'accès aux P-listes.

5. LES ACCES EN LECTURE.

Toute lecture consiste à rechercher la valeur associée à un (ou plusieurs) triplet ~~frame-slot-facette~~.

Le système offre une combinatoire importante de modes d'accès: avec ou sans l'activation de réflexe si-besoin et avec ou sans recherche dans la hiérarchie. Enfin des accès plus complexes sont possibles: fermeture transitive ou collecte de toutes les valeurs dans la hiérarchie.

La primitive de lecture standard est FGET, avec les différents suffixes ("-xx") décrits ci-dessous.

5.1. Lecture simple.

Elle se fait soit sur une facette précise, soit, pour les attributs sur la facette implicite value avec activation du réflexe si-besoin en cas d'absence de la première.

5.1.1. Recherche dans la hiérarchie.

En cas d'absence de valeur pour le triplet ~~frame-slot-facette~~ cherché, il est possible de rechercher ce triplet dans la hiérarchie de la frame.

De plus la recherche dans la hiérarchie avec activation de réflexe peut se faire de 2 manières différentes: en recherchant d'abord toutes les valeurs (facette value) puis tous les réflexes ou bien pour chaque objet de la hiérarchie en recherchant successivement la valeur puis le réflexe.

5.1.2. Les accès standards.

Quelque soit le mode choisi, c'est la première valeur (non NIL) obtenue qui est la bonne.

Si l'on représente les facettes par l'axe horizontal et les objets de la hiérarchie par l'axe vertical, on a ainsi 3 démarches possibles:

- en I, recherche sans réflexe,
- en Z, recherche de toutes les facettes d'un objet avant d'en essayer un autre,
- en N, recherche, sur tous les objets, de la facette value, puis, toujours sur tous les objets, du réflexe si-besoin.

5.2. Les accès multiples.

Ces accès concernent plusieurs triplets simultanément, l'objet seul changeant.

5.2.1. Fermeture transitive des liens.

Cette recherche utilise l'algorithme décrit en [§3.5.1.2], en tenant compte des éventuelles facettes sauf et des cycles possibles. Il y a 3 modes d'accès de ce type:

- en H, pour les liens multiples avec facette Herit: est utilisé pour est-un.
- en L, identique à H, mais sans facette herit: pour les autres liens à valeur multiple.
- en L1, comme L, pour les liens à valeur unique.
- enfin les accès en H⁰ et L⁰ permettent d'accéder uniquement aux "ancêtres", objet de départ exclus: ils permettant une économie d'un CONS par appel de fonctions qui sont appelées des (dizaines de) milliers de fois !

5.2.2. Collecte sur toute la hiérarchie.

Ce mode de recherche fait la collecte de toutes les valeurs d'un slot, le long de la hiérarchie de l'objet, avec élimination des doubles (qui vérifient EQ):

- en A (All) pour les valeurs multiples et en A1 pour les valeurs uniques.
- en A-INV pour la recherche en ordre inverse, en suivant le lien est-un à l'envers, c'est-à-dire le lien d'instanciation.

5.3. La facette défaut.

Il est usuel d'implémenter les frames avec une facette défaut dont la priorité se situe entre les facettes value et si-besoin [Winston 84]. Cette facette représente bien sûr la valeur par défaut de l'attribut (1).

En fait, cette facette semble, soit être superflue — telle que Winston l'implémente, car elle peut être remplacée par l'utilisation de facette value ou init dans un objet d'abstraction supérieure ou par un réflexe si-besoin —, soit nécessiter des modes d'accès plus complexes — à 1, 2 ou 3 facettes en variant l'ordre des priorités des facettes et la recherche hiérarchique suivant la facette. Il faudrait alors implémenter les modes d'accès en comportements, ou bien rajouter une facette réflexe si-lu, comme dans MERING [Ferber 83].

Dans les 2 cas, le coût en performance est sévère.

Actuellement, cette facette n'est pas implémentée.

5.4. Activation des réflexes si-besoin.

Les réflexes si-besoin sont des λ -expressions sans argument: l'environnement d'appel y est connu par le doublet de variables globales frame et slot. Ces réflexes sont évalués en remontant la hiérarchie de l'objet (frame+), jusqu'à ce que l'un d'eux retourne une valeur non NIL.

Quatre points sont à souligner dans leur activation: affectation de la valeur calculée, déclenchement des réflexes si-ajout, contrôle de boucle et vérification a posteriori de l'absence de la valeur cherchée.

(1) Les problèmes logiques posés par les facettes défaut et seul sont comparables. Pour plus de détail voir entre autres: [Reizer 81 et 83], [Tourretsky 81 et 84].

5.4.1. Affectation.

D'abord toute valeur retournée par un réflexe si-besoin est affectée à l'attribut considéré sous la facette value: tout réflexe n'est donc activé qu'une seule fois. Tout se passe comme s'il y avait un FPUT+ de la valeur calculée.

5.4.2. Réflexes si-ajout.

Il y a alors activation des éventuels réflexes si-ajout, à l'issue desquels la valeur retournée par la primitive d'accès est la valeur réellement présente dans l'objet et non pas la valeur calculée avant le déclenchement des réflexes si-ajout. En d'autres termes, un réflexe si-ajout peut modifier de façon effective la valeur précédemment retournée par le réflexe si-besoin.

5.4.3. Contrôle de boucle.

Ensuite un contrôle de boucle est prévu à toute activation: le SLOT+ de l'objet FRAME+ (objet récepteur) est marqué [4.2]. En cas d'essai d'accès en lecture avec activation de réflexe, la présence de ce marquage fait renvoyer la valeur NIL, sans activer le réflexe. Ce flag est protégé par un PROTECT: il est donc enlevé en cas d'échappement.

Ce contrôle de boucle n'est pas si inutile qu'il n'y paraît: on peut ainsi imaginer 2 attributs A et B, ayant chacun 2 méthodes de calcul suivant que l'autre est connu ou pas. Or le réflexe calculant B n'est pas censé savoir que A est en cours de calcul: la boucle infinie paraît ainsi difficile à éviter, sauf à alourdir considérablement le code des réflexes et — ce qui est pire — l'analyse du problème.

5.4.4. Vérification a posteriori.

Enfin, il y a une vérification à posteriori de l'absence de la valeur cherchée. En effet, il est possible que, par effet-de-bord indirect, la valeur recherchée ait été ajoutée dans l'environnement d'appel de façon détournée. On se retrouve alors avec 2 valeurs: l'une mise subrepticement au cours de l'activation du réflexe et l'autre calculée par celui-ci. C'est la première, qui l'est aussi par la chronologie, qui est considérée comme la bonne et retournée après l'activation du réflexe. Les réflexes si-ajout ne sont alors pas activés.

Cette dernière facilité tend, comme la précédente à faire admettre un certain indéterminisme dans l'écriture des réflexes. Le choix chronologique est évidemment arbitraire: il est fondé essentiellement sur cette chronologie et sur le fait qu'écrire 2 fois de suite la même valeur au même endroit retourne NIL (cf. accès en ajout).

5.5. Primitives de vérification.

Ce sont des fonctions qui testent la présence d'une valeur donnée dans la valeur d'une facette d'un attribut d'un objet. Le test utilise la fonction LISP EQ et non EQUAL: il s'agit donc d'un test d'identité et non d'isomorphisme.

Il y en a 2 de base: FCHECK vérifie l'identité des 2 valeurs et FREQD l'appartenance de la valeur proposée dans la liste des valeurs de la facette.

En cas de succès la première retourne la valeur proposée tandis que la

seconde retourne la sous-liste des valeurs commençant par celle-ci.

Les suffixes possibles sont les mêmes que pour la lecture (FSET), mais ils ne sont pas tous possibles pour les 2 primitives:

- pour les 2: I, N et Z.
- pour FMEMB seul: A, A1, H, L et L1.

6. ACCES EN ECRITURE.

Il y a trois types d'écriture: l'affectation, l'ajout et le retrait de valeurs, et la création et l'effacement des facettes et des slots.

Tout écriture se fait dans le triplet frame-slot-facet argument, sans aucun héritage.

De plus toute écriture standard comporte un test sur la valeur (affectée, ajoutée ou enlevée), qui doit être non NIL, ainsi que sur son absence dans la valeur actuelle de la facette. Si l'un de ces tests n'est pas vérifié, il n'y a aucune action entreprise, et la primitive retourne NIL (1). Ces tests se font avec la fonction EQ (et non EQUAL): sont donc testés les valeurs physiques (égalité des pointeurs) et non logiques (identité d'impression) (2).

Dans les autres cas ces primitives renvoient comme valeur la "partie modifiée" de la valeur, après avoir éventuellement déclenché les réflexes appropriés (si-ajout ou si-enlève).

6.1. Création, affectation et ajout.

Le fonctionnement standard de la création d'un slot ou d'une facette consiste à la (la) rajouter en tête de la liste des attributs ou facettes.

Les primitives d'écriture sont FPUT, pour l'affectation, FADD, pour l'ajout d'une valeur, et FADD* pour l'ajout d'une ou plusieurs valeurs, avec comme unique suffixe "*" en cas d'activation de réflexe si-ajout.

```

? ^Va
  (<)                               : la valeur-objet de a est (<)
= a
? (fadd* 'a 'b 'c 'd)
= d
? (fadd a 'b 'c 'd)
= (<)
? (fadd* a 'b 'c '(e d f t))
= (e f t)
? (fnew a 'b 'c 'f)
= f
? (fpush a 'b 'c 'd)
= d                               : empilement d'une nouvelle facette
? ^Va
(a (b (c . d)
      (c f d e f t)))
= a

```

Fig. 6

(1) Ainsi la suite de 2 écritures successives de la même valeur au même endroit retourne systématiquement NIL !
 (2) On retrouve ainsi le problème classique en LISP de l'égalité des nombres, chaînes de caractères ou vecteurs.

D'autres primitives, moins standards, permettent des affectations sans aucun test (FCONC) ou des ajouts de valeur sans test et en tête de valeurs (FADD). Enfin FPUSH permet d'empiler dans le slot une nouvelle paire facette-valeur.

Combinées respectivement avec FNEXT et FREM, FREN et FPUSH permettent une gestion de pile LIFO.

N.B. Il faut se méfier avec FADD de ses effets sur les valeurs multiples à valeur liste [4.3], plus précisément si l'en veut ajouter une unique valeur multiple à valeur liste ! Elle est considérée comme une liste de valeurs.

5.2. Accès en retrait ou effacement.

L'accès en retrait ou effacement se fait par l'unique primitive FREM qui enlève la valeur proposée de la facette ou de la liste des valeurs. Si la valeur proposée est T il y a effacement de la valeur présente, quelle qu'elle soit. Sinon il n'y a effacement qu'en cas d'égalité ou d'appartenance de la valeur. S'il y a plusieurs occurrences de la valeur à enlever, seule la première l'est.

FREM retourne, en cas de réussite, la valeur effacée ou enlevée.

Après retrait, si la facette n'a plus de valeur, elle est effacée, de même pour le slot.

? (frem a 'b 'c 't)	: suite de la Fig. 6
= d	
? (frem+ a 'b 'c '(t))	: la seule façon d'enlever T
= (t)	
? (fget a 'b 'c)	
= (f d e f)	
? (frem+ a 'b 'c '(f g))	
= (f)	
? (fget a 'b 'c)	
= (d e f)	
? (fnext 'a 'b 'c)	
= d	
? (frem 'a 'b 'c 't)	Fig. 7
= (e f)	

FREM+ permet le retrait de plusieurs valeurs, (même remarque que pour FADD+), FREM1 ne permet que l'effacement d'une valeur unique, et FNEXT enlève la première valeur, quelle qu'elle soit.

Comme pour les ajouts, le seul suffixe possible "+" entraîne une activation des réflexes si-enlève.

N.B. Il faut noter que FREM (et ses variantes) fait une double modification physique: la valeur retournée (effacée) et la valeur restante se partagent tous les CONS initiaux: les résultats sont donc imprédictible en cas de partage de valeur.

6.3. Déclenchement des réflexes.

Les réflexes d'écriture (si-ajout ou si-enleve) sont évalués après l'écriture si elle réussit.

6.3.1. Evaluation des réflexes.

Ces réflexes sont des λ-expressions sans argument, l'environnement d'appel étant connu par le quadruplet `frame+`, `slot+`, `facet+` et `value+`, cette dernière variable étant liée à la valeur retournée par la primitive d'écriture.

6.3.2. VALUE+: la valeur dans les réflexes d'écriture.

Dans les réflexes d'écriture, VALUE+ est lié à la valeur effectivement écrite (affectée, ajoutée ou effacée). Dans le cas des valeurs multiples, VALUE+ pourrait donc être une valeur unique (avec FADD ou FREM) ou une liste (d'une ou plusieurs) valeur(s) (avec FPUT, FADD+ ou FREM). Pour unifier le point de vue du réflexe, quelque soit la primitive utilisée, VALUE+ est liés à la sous-liste des valeurs effectivement écrites, à l'exception des primitives FNEXT+ et FNEW+ pour lesquelles la liaison se fait sur la valeur unique.

MAIS, dans tous les cas ces primitives retournent la valeur qu'aurait retournée la primitive sans déclenchement de réflexe: valeur unique ou multiple suivant l'argument d'appel.

6.3.3. Ordre d'évaluation.

Tous les réflexes de la hiérarchie de l'objet récepteur (`frame+`) sont évalués, dans l'ordre inverse de celle-ci, c'est-à-dire en redescendant de l'objet le plus général au plus spécifique (3).

6.4. Cas particuliers des liens.

Ces cas particuliers concernent le calcul de la hiérarchie et la facette hérité, pour le lien est-un, et l'activation des réflexes si-enleve en cas d'affectation, pour tous les liens.

6.4.1. L'affectation des liens.

Les liens servent à définir un graphe des objets plus général que le graphe d'héritage. Toute affectation écrase la valeur précédente et la cohérence du graphe (par exemple la maintien des liens inverses) doit être assurée, en déclenchant les réflexes si-enleve sur l'ancienne valeur, avant l'affectation, qui elle-même activera les réflexes si-ajout. La primitive FPUT-1 assure cette cohérence.

6.4.2. Cas particulier du lien est-un.

Les modifications du lien est-un posent des problèmes particuliers d'activation de réflexes: doivent-ils être recherchés dans la hiérarchie pré-existant à la modification, ou bien dans celle résultant de celle-ci.

(3) C'est l'ordre logique d'évaluation des réflexes "A POSTERIORI". L'ordre des réflexes "A PRIORI" étant au contraire en remontant (cf. la combinaison des méthodes dans les Flavors [Macov 81]). Ainsi, chaque accès pour un objet se fait en faisant quelque-chose spécifique "a priori", puis en déléguant le message au-dessus, enfin en faisant autre chose "a posteriori". En remontant "a posteriori", on mélangeait, l'action spécifique et la délégation. En ce sens, les réflexes si-besoin sont des réflexes a priori.

Le choix qui a été fait ici est de prendre l'ancienne hiérarchie pour l'effacement, et la nouvelle pour l'ajout. L'affectation a été considérée comme la succession d'un effacement et d'un ajout, comme pour les autres liens.

Enfin chaque écriture nécessite l'effacement de la facette hérit. Ces actions sont réalisées par les primitives de suffixe -H: FPUT-H, qui est identique à FPUT-L, à la facette hérit près, FADD-H et FREM-H qui correspondent respectivement à FADD++ et FREM++.

6.5. Réflexes si-possible.

Il n'existe pas à l'heure actuelle de réflexes a priori, déclenchables avant l'écriture pour tester la validité de celle-ci. Ils sont actuellement à l'étude, sous le nom de si-possible.

6.6. Écriture et héritage.

Si l'on veut ajouter ou enlever une valeur à un slot hérité, il est nécessaire de le recopier dans l'objet considéré (figure 6). La facette init permet cette copie à la création de l'objet [#15.1.2].

```

? ^VA1
(A1 (b (c d e f g)))
= A1
? ^VA2
(A2 (est-un (value A1)))
= A2
? (fget A2 'b 'c)
= ( )
? (fget-i A2 'b 'c)
= (d e f g)
? (frem A2 'b 'c 'f) ; F n'est pas dans A2 mais dans A1
= ( )
? (fadd A2 'b 'c 'h) ; A1 n'est pas modifié
= h
? ^VA1
(A1 (b (c d e f g)))
= A1
? (frem A2 'b 'c 't)
= h
? (fput A2 'b 'c (fget-i A2 'b 'c))
= (d e f g)
? (frem* A2 'b 'c '(d f h))
= (d f)
? ^VA1 ; mais A2 et A1 se partagent la même valeur
(A1 (b (c e g))) ; il aurait fallu faire un COPYLIST
= A1

```

Fig. 6

7. LES COMPORTEMENTS.

7.1. Généralités.

La programmation par comportement est particulièrement bien adaptée aux fonctions dont l'écriture pourrait se faire avec un SELECT (Lisp ou PLI) ou un CASE Pascal, le test portant sur le "type" d'une "variable".

La "variable" devient l'objet (pour les méthodes) ou l'attribut (pour les applicateurs) et le "type", un des ancêtres de la "variable". Le test porte alors sur la présence du "type" dans la hiérarchie de la "variable".

7.1.1. Nom générique et valeurs fonctionnelles.

Le concept de comportement, à la base des LOD, permet d'utiliser, sous un nom générique, des fonctions différentes dépendant de l'objet auquel on les applique.

Les comportements de YAFOOL, méthodes ou applicateurs, sont des valeurs fonctionnelles (symbole ayant une valeur fonctionnelle, ou bien λ -expression) sous la facette méthode ou applie du slot ayant pour nom le nom générique du comportement (1).

7.1.2. Arguments.

Contrairement aux réflexes standards, les comportements peuvent avoir des arguments, en nombre quelconque. L'objet récepteur du message n'en fait jamais partie. D'autre part il est possible, mais guère souhaitable, qu'un même comportement générique ait des "instances" n'ayant pas le même nombre d'arguments.

7.1.3. Environnement d'appel.

Il est défini par les variables globales `frame*`, l'objet récepteur, et `herite` qui contient la sous-liste de la hiérarchie commençant par l'objet contenant le comportement. Cette dernière variable est essentiellement utilisée pour les super-comportements (voir plus loin). Pour les applicateurs, `slot*` est lié à l'objet dual.

7.2. Activation des méthodes.

Les méthodes sont la version habituelle des comportements: si les réflexes font les LOD, valeurs actives ou frames, les méthodes sont le critère des LOD.

La primitive `FMETH` (2) applique une méthode à un objet, et à quelques arguments: elle cherche dans la hiérarchie la méthode associée à l'objet (recherche équivalente à `FGET-I`) et l'applique (`APPLY`) au reste des arguments d'appel (objet non compris). `FMETH` retourne la valeur retournée par la méthode.

(1) Le terme de comportement, méthode ou applicateur sera toujours ambigu, dans la mesure où il désigne à la fois le nom générique de celui-ci, et ses différentes "instances" dans tel ou tel objet.

(2) C'est l'équivalent de la fonction `SEND` dans certains LOD (CERN par exemple [Maillet 85]), bien que `HERINE` [Ferber 83] utilise `SEND` pour les continuations).

N.B. Il serait envisageable de définir des méthodes dans lesquelles il ne s'agirait pas d'activer la première méthode rencontrée, mais toutes les méthodes présentes dans la hiérarchie de l'objet, un attribut spécial indiquant le mode de déclenchement (un peu comme les réflexes d'écriture par rapport aux réflexes si-besoin): cela se fait très bien avec les super-comportements (voir plus loin).

7.2.1. Absence de méthode.

En cas d'absence du comportement cherché, il est alors fait appel à la méthode (3) pré-définie comport-error qui, dans le cas général, édite le message d'erreur " il n'y a pas de méthode ... pour ...".

Dans ce cas FMETH retourne NIL, et non pas la valeur de la méthode COMPORT-ERROR. Si l'on veut que cette dernière retourne une valeur, celle-ci doit être renvoyée par un double continuation, (<=< valeur à retourner)); voir chapitre suivant.

Il est ainsi possible à l'utilisateur de redéfinir simplement l'action à entreprendre dans ce cas de figure, et de façon adaptée à chaque comportement.

7.2.2. Comportement de masque.

En particulier, s'il veut ne rien faire, il lui faut masquer la méthode comport-error générale par une méthode particulière qui ne fasse rien: c'est le rôle de la fonction METH-NIL, qui sert de fonction générale de masque pour les comportements.

METH-NIL se définit en Le_Lisp par: (de meth-nil u). Elle admet donc un nombre quelconque d'arguments.

7.2.3. Simulation par un SELECTQ.

En faisant abstraction de la hiérarchie, il est possible de simuler simplement des méthodes par un SELECTQ Lisp (ou CASE Pascal).

```
(selectq frame+
  (A   ...
       méthode de A
       ...))
(B   ...
     méthode de B
     ...))
(...
 (t   ... ; en cas d'insuccès dans la recherche
      application de la méthode COMPORT-ERROR
      à la méthode cherchée, avec comme arguments
      ceux de l'appel
      ...)))
```

(3) Il y a donc récursivité, mais celle-ci s'arrête naturellement si la méthode COMPORT-ERROR est bien définie pour la méthode COMPORT-ERROR: on a là un cas d'interférence avec le nouveau moi.

7.3. Activation des applicateurs.

Le concept d'applicateur généralise celui de méthode et permet d'étendre la définition des réflexes: d'une certaine manière, il peut se définir comme un couple méthode-réflexes propres à l'applicateur. Son introduction dans les LOOD semble due à [Ferber 83].

La primitive FAPPL applique un applicateur à un objet, un slot (4) et quelques arguments spécifiques. Elle cherche, dans la hiérarchie du slot, l'applicateur associé à celui-ci (recherche par FGET-I) et l'applique au reste des arguments d'appel.

7.3.1. Activation des réflexes.

Si l'applicateur a retourné une valeur non NIL (5), les réflexes sont activés automatiquement par FAPPL, en cas de présence de l'attribut reflex-apply dans l'applicateur.

Sa valeur (ANY, EVERY, MAPC (6) ou NIL) donne le mode de déclenchement des réflexes.

Le concept d'applicateur ainsi défini est inséparable de ses réflexes: sans eux il est équivalent à celui de méthode. Ils réalisent, en quelque sorte, des comportements à 2 dimensions, les méthodes n'étant qu'à une dimension.

7.3.1.1. Héritage à 2 dimensions.

Dans l'activation des réflexes standards (si-ajout, si-enleve et si-besoin), l'attribut est fixé, et l'on fait varier l'objet dans sa hiérarchie en collectant tous les réflexes rencontrés.

Dans le cas des réflexes d'applicateur, la recherche va se faire, non pas dans le "vecteur" de la hiérarchie de l'objet, mais dans la "matrice" (7) formée par les hiérarchies de l'objet et du slot.

Si les objets représentent les indices des colonnes, et les slots ceux des lignes, la recherche se fait ligne par ligne: pour chaque élément de la hiérarchie du slot, on parcourt celle de l'objet. Dans le cas où seuls le slot argument (SLOT+) possède des réflexes, on est ramené au cas des réflexes standards.

7.3.1.2. Arguments.

Enfin, il faut noter que les réflexes des applicateurs ont les mêmes arguments que ceux-ci, contrairement aux réflexes standards qui n'en avaient pas.

(4) En réalité, n'importe quel objet peut jouer ce rôle, mais en l'absence d'exemple, c'est un objet quel que permet de réaliser le mieux la sémantique de ces applicateurs.

(5) Il s'agit encore une fois de réflexes a posteriori. Des réflexes a priori sont concevables et pourraient facilement être implémentés si le besoin s'en fait sentir. Mais l'ensemble du concept d'applicateur sera sans doute revu et simplifié [a21].

(6) Ou tout autre valeur fonctionnelle (à 2 arguments) définie par l'utilisateur: par exemple fonction de mapping à l'envers etc...

(7) Les termes de vecteur et matrice sont assez mal choisis puisqu'il s'agit en fait, respectivement, d'ensemble et d'ensemble produit.

7.3.1.3. Rôle de VALUE+

Avant l'activation de ces réflexes, la valeur retournée par l'applicateur est sauvegardée dans VALUE+, dont la valeur est retournée par FAPPL: les réflexes peuvent donc modifier cette valeur.

7.3.2. Exemples.

Pour appréhender ce concept d'applicateur-réflexe, on peut imaginer que toutes les primitives des frames sont des applicateurs virtuels liés à l'objet racine de l'univers dual. A FGET, FPUT et FREM sont alors associés des réflexes si-besoin, si-ajout et si-enleve, avec des facettes reflexe-applic de valeur respective ANY, MAPC et MAPC. Pour toutes les autres primitives, la facette reflexe-applic est vide.

```
(objet-ideal
  (besoin (applic ())
    (let ((val (fget-i frame+ slot+))
          (if val (→ val) t)))
      ;pseudo-continuation
      ;pour court-circuiter
      ;les réflexes.

      (reflex-applic . any))
  (ajout (applic (val)
                (fput frame+ slot+ val))
    (reflex-applic . mapc)))
```

Les applicateurs besoin et ajout, couplés aux réflexes si-besoin et si-ajout sont approximativement identiques aux primitives FGET-W et FPUT+. L'équivalence est totale pour ajout. Par contre, pour besoin, il faudrait que le réflexe prenne en charge tous les mécanismes décrits en [#5.3].

Pour des exemples concrets d'applicateurs, voir les chapitres [#13] sur les :applicateurs et [#16] sur les extensions temporelles.

7.3.3. Simulation par un SELECTQ.

Avec la même restriction que pour les méthodes, on peut simuler une activation d'applicateur par des SELECTQ emboîtés.

L'activation des réflexes se fait, sur tous ceux qui sont rencontrés, par l'intermédiaire de l'une des fonctions de mapping MAPC (tous les réflexes), EVERY (tant qu'ils retournent une valeur non-NIL) ou ANY (jusqu'à la première valeur non-NIL), suivant la valeur de l'attribut reflex-applic de l'applicateur.

7.3.4. Asymétrie des applicateurs.

Les applicateurs ainsi définis sont asymétriques: ils s'appliquent à un couple d'objets, l'applicateur étant déterminé par le deuxième objet, et les réflexes par l'ensemble des 2. Mais l'ordre de ceux-ci pourrait être défini, colonne par colonne, et non ligne par ligne, sans que ce soit réductible à la forme précédemment définie.

Il ne semble pas utile pour l'instant d'introduire la forme symétrique de celle que nous avons choisie, ce choix ayant d'ailleurs été essentiellement dicté par des raisons de performance.

```

(selectq slot+
  (A ...
    applicateur de A
    (selectq frame+
      (X ...
        réflexes de A-X)
      (Y ...
        réflexes de A-Y)
      ...))
  (B ...
    applicateur de B
    (selectq frame+
      (X ...
        réflexes de B-X)
      ...))
  ...
  (t ... ; en cas d'insuccès dans la recherche
    application de la méthode COMPORT-ERROR
    à l'applicateur cherché, avec comme arguments
    ceux de l'appel
    ...))

```

7.4. Héritage biaisé.

Dans certains cas, il est souhaitable de pouvoir appliquer à un objet un autre comportement que celui dont il hérite normalement (le premier trouvé dans sa hiérarchie).

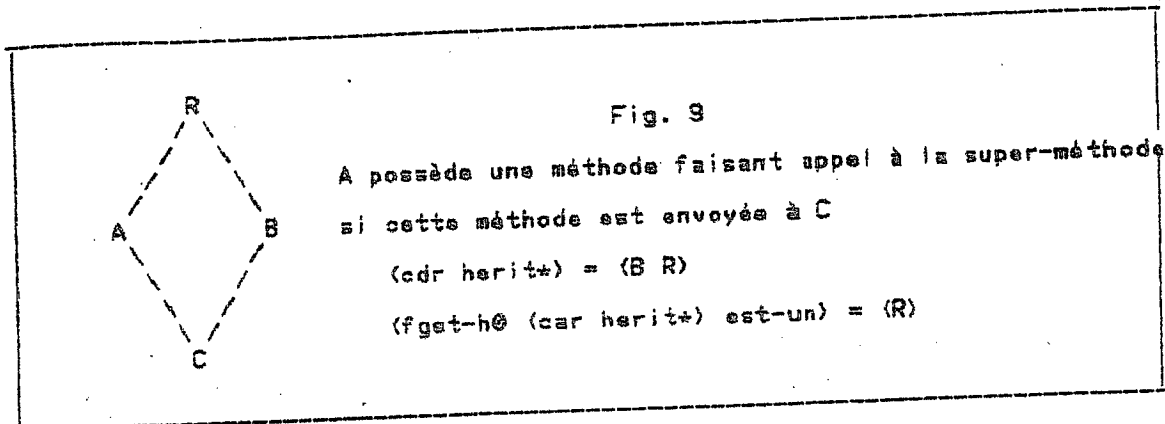
Les primitives `FAPPL` et `FMETHOD` reçoivent comme argument une liste d'objets, la sous-hiérarchie à utiliser, au lieu de l'objet (ou de l'attribut pour les applicateurs) à partir duquel se fait la recherche. Le comportement s'applique implicitement à `FRAME+` (et `SLOT+` pour les applicateurs).

7.4.1. Super-comportements.

La principale utilisation de ces primitives est l'écriture de méthodes spécifiques utilisant récursivement la même méthode, mais en plus général (8). La sous-hiérarchie en argument est alors (car hérite).

Un autre cas particulier de hiérarchie biaisée est celui où l'on prend (`fgat-n@` (car hérite) est-`un`) ; c'est à dire l'héritage de l'objet contenant le comportement (conf. Fig. 9).

(8) Elles réalisent ainsi l'équivalent du `SUPER` de `SMALLTALK` ou `HERITE` et du `←SUPER` de `LOOPS`, et permettent une combinaison des méthodes aussi riche que celle des `Flavors` (Moon 81).



7.4.2. Environnement.

Les primitives d'héritage biaisé `FMETH` et `FAPPL` ne comportent pas en propre de ~~tag-continuation~~ (voir le chapitre suivant). Elles utilisent donc celui de la primitive `FMETH` ou `FAPPL` initiale, la seule variable d'environnement qui y soit liée à nouveau étant `HERITE`.

En conséquence, l'héritage biaisé devrait être utilisé uniquement à l'intérieur du comportement spécifique, et non pas médiatisé par des attachements procéduraux intermédiaires (sauf à utiliser des continuations pour se ramener au niveau du comportement spécifique).

7.4.3. Réflexes.

`FAPPL` ne fait pas appel en propre à des réflexes: ceux-ci seraient alors en effet activés 2 fois: par `FAPPL` et par `FAPPL`.

8. CONTROLE D'ENVIRONNEMENT ET COMMUNICATION.

Le mode de communication de base des langages objets est la passation de messages. Ce paragraphe présente deux autres structures de communication: les continuations, par lesquelles un comportement peut préciser sa propre "continuation" et les retardements qui permettent de bufferiser (tamponner) les messages.

8.1. Liaison dynamique d'objets.

Les primitives de liaison dynamique des objets sont FLETF et FSET: leur syntaxe est identique à celle du LET, leur effet consistant à lier dynamiquement une (des) variable(s) à un (des) objet(s) initialisé(s) par son (leur) lien est-un.

```

? (fletf ((a 'b 'c) (x 'y))
      (fpretty a)
      (fpretty x))
(a (est-un (value b c)))
(x (est-un (value y)))
= x

```

A la sortie normale de FLETF ou FSET, l'environnement est restauré: a et x retrouvent leur valeur-objet d'origine. La différence entre ces 2 primitives tient à leur comportement en cas d'échappement: FLETF utilise WITH et donc PROTECT: l'environnement est protégé, même en cas d'échappement, ce que ne fait pas FSET.

8.2. Liaison dynamique de valeurs.

Les primitives FWITH et FAVEC associées à FGEN permettent des liaisons dynamiques au niveau de la facette et non de l'objet.

FGEN est une variable-fonction (1): avec 3 arguments, c'est FGET, alors qu'avec 4 arguments c'est FPUT sans test, c'est-à-dire FCONC.

FWITH et FAVEC possèdent la syntaxe du LET, en remplaçant les couples variable-valeur par des quadruplets frame-attribut-facette-valeur.

La différence entre FWITH et FAVEC est la même que celle entre FLETF et FSET (fig. 10).

N.B. Le problème des réflexes d'écriture se pose pour ces fonctions de modification dynamique d'environnement: dans la version actuelle, il n'y en a pas. Des primitives FWITH+ etc.. sont concevables: reste à les utiliser dans des conditions de cohérence parfaites: les réflexes si-ajout et si-

(1) Les variable-fonctions Le_Lisp sont des fonctions (éventuellement macros) à M ou M+1 arguments. De façon standard, avec M arguments, il s'agit d'une lecture et, avec M+1, d'une écriture. La fonction WITH permet de modifier dynamiquement ces variable-fonctions, avec protection de leur ancienne valeur par PROTECT.

enlève doivent alors être totalement symétriques, pour que le rétablissement de l'environnement s'étende aux effets de bord de ces réflexes.

8.3. Continuations et retards.

Ces structures ont un intérêt multiple: continuation classique, par laquelle on peut faire de la pseudo-récursivité, sans empilement, buffering de messages, structure uniforme de sortie anormale (échappement) de message, possibilité de répondre à l'émetteur d'un message sans le connaître etc... Toutes ces possibilités sont mises en oeuvre à partir d'une unique structure d'implémentation des déclenchements d'attachements procéduraux.

8.3.1. Le principe "officiel" de la continuation.

On entend habituellement continuation, comme l'action de faire définir par un objet -- par un comportement précis de cet objet -- ce qu'il faut faire après, la "continuation" du message en cours d'interprétation. Cette continuation est activée au Top-level de l'interpréteur, après un échappement qui termine abruptement le comportement en cours.

8.3.2. Continuation locale.

Dans YAFOOL, ce concept a été réinterprété localement, l'échappement se faisant non pas au Top-Level mais au "niveau" du message en cours d'interprétation. YAFOOL n'ayant pas de top-level en propre, c'est celui de Le_Lisp qui est utilisé, la continuation classique n'est pas implémentable.

Par contre, qui peut le plus peut le moins, il est possible de concevoir une continuation globale qui ferait un échappement au niveau du message initial: celui du "haut" de la pile. On aurait alors un équivalent assez exact de la continuation classique.

8.3.2.1. Continuation simple.

Les continuations sont activées par la primitive => dont les arguments sont une suite de formes LISP.

Son action est de restaurer l'environnement précédant le dernier lien dynamique -- créé par FLET, FSET, FWITH ou FAVEC ou bien par liaison du quadruplet FRAME+--SLOT+--FACET+--VALUE+ dans le déclenchement d'un attachement procédural. L'évaluation, comme un PROG, des formes LISP arguments a lieu après l'échappement, dans un environnement plus ancien.

Il s'agit d'un échappement: il y a donc interruption du FLET ou de la primitive en cours d'évaluation.

Il est possible de pérenniser les modifications dues à FAVEC ou FSET par une continuation (ou tout échappement). Par contre FWITH et FLET restaurent forcément l'environnement antérieur.

8.3.2.2. Pseudo-continuation.

La pseudo-continuation --> fait un simple échappement, avec évaluation avant l'échappement. C'est une simple sortie anormale d'une primitive.


```

? (fput 'a 'b 'c)
= c
? (fwith (('a 'b 'value 'd))
?      (print (fget 'a 'b))
?      (==> (fget 'a 'b)))
d
= a
? (favec (('a 'b 'value 'd))
?      (print (fget 'a 'b))
?      (==> (fget 'a 'b)))
d
= d

```

Fig. 10

FWITH, FAVEC et ==>

L'utilisation des pseudo-continuations se limite à contrarier un mécanisme standard dans le déclenchement des réflexes et des applicateurs.

La pseudo-continuation (—> valeur-a-retourner) aura pour effet dans un réflexe si-besoin de court-circuiter la réécriture, le déclenchement des réflexes si-ajout et la vérification à posteriori. La valeur retournée sera VALEUR-A-RETOURNER.

Dans un réflexe si-ajout et si-enleve, cette pseudo-continuation empêchera l'activation des autres réflexes. Si l'on veut que la primitive d'écriture retourne sa valeur normale, il faudra utiliser (—> valeur).

Dans une méthode ou un applicateur sans réflexe, la pseudo-continuation n'a guère de sens, la valeur retournée sera simplement VALEUR-A-RETOURNER.

Dans un applicateur avec réflexes, il y aura inactivation de ceux-ci, et dans un de ces réflexes, inactivation des autres et valeur-a-retourner sera la valeur de l'applicateur: comme pour les réflexes d'écriture, il faut utiliser (—> valeur) si l'on ne veut pas modifier la valeur retournée par l'applicateur.

8.3.3. Retardements.

A l'usage (2), le besoin s'est fait sentir d'un mécanisme équivalent, sans échappement, permettant de retarder une évaluation, de signifier la "continuation" du message en cours, mais sans interrompre celui-ci: c'est la continuation retardée, ou retardement.

Comme pour les continuations locales, elles existent en 2 versions (← et ←=) suivant que l'évaluation se fera dans l'environnement de la continuation, ou en dehors: on parlera donc de retardement interne ou externe.

Ce sont des FEXPR qui gèrent une pile FIFO de messages (voir paragraphe suivant) et retournent toujours la valeur NIL.

(2) Les continuations (—> et ==>) sont souvent utilisées, même au niveau des primitives, sous leurs formes les plus simples.

8.3.4. Mécanisme général des continuations.

8.3.4.1. Structure d'un "tag-continuation".

Pour bien comprendre le mécanisme de ces continuations, il est nécessaire de rentrer dans les détails de la cellule de base d'activation des attachements procéduraux et des liaisons dynamiques d'objets (FLETF et FSET) ou de valeurs (FWITH et FAVEC): ce que l'on nommera par la suite le TAG-CONTINUATION.

```
(let ((#:continu:message=)
      (#:continu:cont=liste))
  (progl (tag continuation
          (let ((#:continu:message-)
                (#:continu:cont-liste)
                ...
                liaison du quadruplet s'il y a lieu
                ...))
          (setq value+
                ...
                corps de la primitive
                ...))
          éventuels réflexes a posteriori
          ...
          (avmessage #:continu:message-)
          value+))
  (avmessage #:continu:message=)))
```

AVMESSAGE est une fonction à peu près équivalente à EPROGN, qui évalue séquentiellement la liste des formes de son argument, en le dépilant (NEXTL) après chaque évaluation. Il est ainsi possible dans une des formes évaluées (même la dernière) de rajouter à la pile de messages un message supplémentaire.

#:CONTINU:MESSAGE- (resp. #:CONTINU:MESSAGE=) est la pile FIFO de messages créés par ← (resp. ←=).

#:CONTINU:CONT-LISTE (resp. #:CONTINU:CONT=LISTE) est la pile FIFO de variables gérées par ←-let- (resp. ←=let=).

#:CONTINU:CONT=LISTE est aussi utilisée en LIFO par =let=.

8.3.4.2. Rôle de VALUE+.

Comme le montre la structure du tag-continuation, value+ sert à pointer sur la valeur que doit retourner normalement la primitive. Il est donc possible de la modifier, soit dans les réflexes a posteriori (3), soit dans les continuations retardées internes.

Enfin, en cas de continuation (ou pseudo-continuation) locale, la valeur retournée par la primitive est celle qui est retournée par la continuation. Pour la pseudo-continuation uniquement, ce peut être

(3) Ce sont tous les réflexes, à l'exception des ai-bezoïn.

value+.

8.3.4.3. Ordre des évaluations.

On a donc la séquence d'évaluation suivante:

- corps de la primitive,
- réflexes a posteriori,
- messages internes,
- messages externes.

Toute continuation locale dans l'un des 3 premiers points inhibe le restant de ces 3 points. Par contre les messages externes sont bien évalués.

Tout retardement dans l'un de ces 3 premiers points, sera évalué plus tard, à moins que n'intervienne entre temps une interruption (continuation ou autre) qui l'inhibe.

Le cas des messages externes est plus particulier:

les retards externes y seront bien évalués plus tard; par contre, toute autre continuation concerne le tag-continuation du niveau supérieur, s'il existe.

8.3.4.4. Liaisons des λ-variables.

Les continuations et retards faisant des évaluations en-dehors de l'environnement d'appel (4), par bufferisation ou échappement, il est impossible d'y utiliser des λ-variables liées au moment de la continuation:

```

(let ((var (mess1 frame+)))
  (=> (mess2 frame+ var)))
  applique mess2
  à une vieille liaison de var.

```

Il faut donc utiliser des variables globales, en empilant et dépilant:

```

(newl pile-var (mess1 frame+))
(=> (mess2 frame+ (nextl pile-var)))
  permet de faire passer à MESS2
  la valeur retournée par MESS1.

```

Les primitives `≠ET=>`, `<-LET-` et `<=LET=` (5) de syntaxe quasi identique à celle du LET, permettent une écriture généralisée de ce mécanisme. Elles utilisent pour le passage des valeurs les variables globales `#:CONTINU:CONT-LISTE` et `#:CONTINU:CONT=LISTE`.

(4) A l'exception des pseudo-continuations qui ne sont pas concernées par ce problème.
 (5) Prononcer let-fleche.

```

<=>let= ((var-1 val-1)
         var-2
         (var-3 val-3)
         ...)

```

équivalent à

```

<=>let= ((var-1 val-1)
         (var-2 var-2)
         (var-3 val-3)
         ...)

```

et s'expande en

```

(prog (newl #:continus:cont-liste
        (list val1 var2 val3))
      (=> (letvq (var1 var2 var3)
          (nextl #:continus:cont-liste)
          ...)))

```

NEWL et NEXTL sont les fonctions d'empilement et de dépilement. Cette expansion autorise une utilisation récursive de #:CONTINU:CONT-LISTE dans l'évaluation des VALI.

L'exemple précédent s'écrit alors:

```

(=let=) ((var (mess1 framew))
        (mess2 framew var))

```

La sémantique de =let= est la même, sauf qu'elle utilise NEWL (empilement LIFO) au lieu de NEWR (FIFO).

8.3.5. Continuations en cascade.

La notion de cascade de continuation consiste en un passage de message, non pas au niveau où l'on est, mais N niveaux au-dessus.

Elle mérite d'être étudiée de plus près, indépendamment de tout problème de passage de λ -variables.

8.3.5.1. Cascade de retardements.

On peut remarquer que les retardements sont quasiment idempotents: l'unique différence entre $(\ll = (\ll \dots))$ et $(\ll = \dots)$ réside dans le fait que la première va être une nouvelle fois retardée, et donc évaluée après les autres messages déjà présents dans la file, mais toujours au même niveau.

Pour une cascade, il faut donc une séquence alternée de retardements externes et internes, commençant par un externe si l'on déclenche la cascade dans un attachement procédural (sinon, on est déjà dans une continuation):

$\langle\langle = \langle\langle - \dots \langle\langle = \dots \rangle\rangle \rangle\rangle$.

8.3.5.2. Cascade avec continuation locale.

La pseudo-continuation est idempotente:

$\langle\langle - \rangle \langle\langle - \dots \rangle\rangle$ et $\langle\langle - \dots \rangle\rangle$ sont strictement équivalents.

On remarque aussi que la cascade

$\langle\langle - \rangle \langle\langle - \dots \rangle\rangle$ n'a aucun sens, le retardement n'étant jamais évalué.

Enfin, les 2 cascades

$\langle\langle - \rangle \langle\langle = \dots \rangle\rangle$ et $\langle\langle = \rangle \langle\langle = \dots \rangle\rangle$ sont équivalentes.

A ces 3 remarques près, toutes les combinaisons de continuations et de retardements sont licites.

8.3.5.3. Cascades avec liaisons de λ -variables.

En cas d'emboîtement de continuation, la syntaxe est allégée et le passage des λ -variables amélioré.

```

<<=let= ((var-1 val-1))
  (<<- ...))
                                     équivaut à:
<<=let= ((var-1 val-1))
  (<<-let= ((var-1 var-1))
    ...))

```

Cette propagation des λ -variables est valable même en cas d'emboîtement de liaison de λ -variables.

```

<<=let= ((var-1 val-1))
  (<<-let= ((var-2 val-2))
    ...))
                                     équivaut à:
<<=let= ((var-1 val-1))
  (<<-let= ((var-1 var-1)
    (var-2 val-2))
    ...))

```

Ce dernier exemple s'expande de la façon suivante.

```

(prog (newr #:continus:cont=liste
        (list val-1))
  (<=> (newr #:continus:cont=liste
            (nconc (nextl #:continus:cont=liste)
                  (list val-2)))
  (<-- (letvq (var-1 var-2)
        (nextl #:continus:cont=liste)
        ...))))

```

8.3.6. Les continuations dialogantes.

8.3.6.1. Continuations emboîtées.

La continuation permet à un objet de répondre au message que lui envoie un autre objet. Se pose alors la question de la réponse à la réponse.

Soit A qui envoie un message à B, qui envoie à son tour un message à C, ce dernier envoyant, par une continuation, un message à B. Si dans ce message, B active une continuation, est-ce une réponse à C ou à A ?

Un même mécanisme ne peut pas faire les deux. L'échappement expliqué plus haut montre bien que la réponse est pour A: il s'agit de dépiler un cran de plus. Tout se passe comme si B, recevant une réponse de C, croyait à un message de A.

On en déduit que, si la continuation est bien un mécanisme de réponse à un message, ce n'est pas un mécanisme récursif au sens propre du terme.

8.3.6.2. Continuation dialogante.

La définition d'une continuation dialogante peut se faire néanmoins à peu de frais: la primitive `<=>` se comporte comme `==`, à cette différence près qu'au lieu de dépiler d'un cran, elle inverse les 2 premiers éléments de la pile: dans les 2 cas le nouveau sommet est bien le même, mais le précédent change. Cette inversion des environnements se fait par dépilement et empilement: seuls les quadruplets sont réempilés, ce qui exclut cette continuation dialogante du champ des primitives de liaison d'objets ou de valeurs.

La sortie d'une continuation dialogante met fin au "dialogue" en dépilant les 2 "interlocuteurs":

en effet ceux-ci ont été empilés dans la continuation elle-même (contrairement à la continuation simple).

8.3.6.3. Exemples.

Quelques exemples deviennent indispensables pour saisir l'action de ces continuations, ainsi que leur appel récursif.

Supposons que A, à la réception d'un message (il est alors en tête de pile), envoie à B le message

(mess1 b).

: la pile est AB dans MESS1 (6)

Pour répondre à A, l'activation de MESS1 doit déclencher une continuation simple:

(=>) (mess2 frame+)).

: la pile est AA dans MESS2 (7).

Supposons maintenant que A envoie un message à B, qui à son tour envoie un à C, dans lequel C répond à B le même message MESS1, par la continuation

(=>) (mess1 frame+))

: la pile est ABB dans MESS1.

Ce qu'il veut mieux ne pas faire:

Si MESS1 veut répondre à A, il lui faut alors déclencher une double continuation:

(=>) (=>) (mess2 frame+)))

: la pile est bien AA dans MESS2 (8).

On voit que la réponse à un même message ne se fait pas de la même manière suivant que ce message est émis par une continuation, ou non.

Restant avec A, B et C, supposons maintenant que la continuation de réponse à B est dialogante:

(<=>) (mess1 frame+))

: la pile est ACBB dans MESS1.

La réponse de B à C peut alors avoir les 2 formes:

(=>) (<=>) (mess2 frame+)))

: la pile est ABCC dans MESS2.

ou

(=>) (=>) (mess2 frame+)))

: la pile est ACC dans MESS2.

Il est clair que l'emboîtement des continuations n'est pas d'une grande simplicité !

(6) Il s'agit ici d'une syntaxe "idéale": MESS1 doit être considéré comme une quelconque primitive s'appliquant à un objet (B) et à un certain nombre d'arguments qui sont ici omis.

(7) Ce doublement du sommet de la pile s'explique ainsi: MESS2 est un message envoyé à lui-même par FRAME+. Dans le cas de la continuation (=>) (mess2 (mess3 frame+))), la pile est AA dans MESS3, mais AX dans MESS2, où X est la valeur retournée par MESS3.

(8) Expliquons l'évolution de la pile sur cet exemple:

```

=>                                     : la pile est ABC
=>                                     : dans la continuation elle devient AB
(mess2 frame+                          : dans la seconde A
)                                       : dans MESS2, AA par duplication du sommet,
)                                       : le nouveau A
)                                       : elle reste A.

et sur celui-ci

=>                                     : la pile est toujours ABC
<=>                                    : dans la continuation elle devient AB
(mess2 frame+                          : dans la seconde BA
)                                       : dans MESS2, BAA par duplication du sommet,
)                                       : le nouveau BA
)                                       : elle est vide.

```

Ce qu'il vaut mieux faire:

Une méthode moins imprévisible consiste à éviter les continuations dans les messages activés eux-mêmes dans une continuation, mais à forcer la réponse par une continuation dans la continuation.

Ainsi:

```
(=> (mess1 frame*)           :réponse à B
      (=> (mess2 frame*)))    :réponse forcée à A
```

ou

```
(<=> (mess1 frame*)           :réponse à B
      (=> (mess2 frame*)))    :réponse forcée à C.
```

8.3.7. Continuations globales.

Elles réalisent approximativement la continuation au top-level classique, qui se traduit ici par un échappement au niveau du tag-continuation "le plus haut".

Elles existent en 3 versions: <<=>, ->> et =>> (9), et sont définies récursivement.

Leurs variantes avec liaison de λ -variables <<=let, -let->> et =let=>> existent aussi.

(en cours d'implémentation)

(9) <<- est impossible à implémenter, puisque la définition récursive de ces continuations globales repose sur un test d'arrêt qui consiste à vérifier la présence "en avant" d'une pile de:CONTINU:MESSAGE.

9. AUTRES PRIMITIVES.

Quelques primitives diverses sont inclassables ailleurs:

FREM-2L efface les facettes *value* et *sauf* du lien *est-un* avec un déclenchement correct des réflexes *si-enleve* en conservant la hiérarchie d'origine. Utilisé par **FREMOB**.

FREMOB détruit (**REMOB**) son argument, après avoir effacé son lien *est-un* (avec **FUNCLAMP**), pour activer ses réflexes *si-enleve*.

FCLAMP et **FCLAMP+** permettant de réunir physiquement le même slot de 2 objets différents qui se partagent alors l'ensemble de leur facettes, et ce jusqu'à effacement du slot dans les 2 objets, ou bien appel à **FUNCLAMP**. **FCLAMP+** active en plus des réflexes *si-enleve* et *si-ajout* sur les facettes *value* et *sauf*. **FCLAMP+** est essentiellement destiné au **FCLAMP**age des liens *est-un*.

FCLAMP2 à la même rôle que **FCLAMP**, pour 2 slots différents.

FUNCLAMP défait le "**FCLAMP**age" des fonctions précédentes par recopie du slot. C'est une copie sur 2 niveaux: les valeurs ne sont pas **EQ** (seulement **EQUAL**), mais les valeurs élémentaires des valeurs non atomiques sont **EQ**.

FMOD modifie la valeur d'une facette, en lui appliquant une λ -expression prenant comme argument la valeur elle-même et la suite des arguments de l'appel de **FMOD**. **FMOD+** active en plus des réflexes *si-ajout*.

FGET-H donne la fermeture transitive d'un lien (son 2-ième argument), à partir, non pas d'un objet (comme **FGET-H** ou **FGET-L**) mais d'une liste d'objets.

FRPLAC remplace une valeur par une autre, dans une facette donnée, à valeur multiple. **FRPLAC+** active aussi des réflexes *si-enleve* puis *si-ajout*.

FCOPY crée une frame copie (**COPYLIST**) d'une autre, avec **FCLAMP**age des liens *est-un*. C'est une copie sur 3 niveaux seulement: les valeurs sont **EQUAL** mais pas **EQ**, mais, pour les valeurs non atomiques, les valeurs élémentaires sont **EQ**.

FPRETTY (abrégée en **~y**) est la fonction de pretty-print des frames.

Enfin, **FPJCA** retourne le plus jeune commun ancêtre d'une liste d'objets. Cette fonction bizarre sert entre autres, à optimiser le placement de réflexe portant sur un attribut.

TROISIEME PARTIE

LE NOYAU NOU

YAF00L 2.1

10. LE NOYAU MOU.

Cette troisième partie décrit le noyau mou du langage, "mou" parce que plus facilement modifiable. C'est ce qu'un utilisateur "moyen" en voit, ce qu'un utilisateur "évolué" peut étendre ou redéfinir. Cette nouvelle couche se définit à la fois par des principes, et leur actualisation. Les 2 étant intimement liés.

10.1. Principes.

Les idées de départ, les motivations, consistaient à donner au système une certaine "intelligence", qui permette:

- une syntaxe et des déclarations elliptiques: compréhension d'un certain implicite et absence de redondance;
- une "grande" puissance de représentation;
- des grandes facilités d'extension et de redéfinition: l'idée est de fournir à l'utilisateur, en guise d'exemple, un langage parmi d'autres possibles, en plus d'une espèce de MECCANO (1), où chacun viendrait prendre ce qui lui plait.
- le tout, sans (trop de) perte de performances (temps d'exécution et espace mémoire) par rapport à l'utilisation directe du noyau dur.

10.2. Résultats.

Les exigences précédentes ont abouti à:

- des objets durs pour tous les slots ou facettes: c'était un minimum pour représenter une certaine méta-connaissance du système;
- des univers d'objets et des mécanismes associés, permettant à la fois une modularisation du système et une dualisation "automatique";
- des accès personnalisés aux objets et aux slots, par l'intermédiaire d'applicateurs;
- des macros pour interfacer ces applicateurs pour qu'ils ne pénalisent pas les temps d'exécution par une recherche dans la hiérarchie à chaque accès;
- des objets slots et facettes en autoloading, pour que tous les développements ne soient pas présents nécessairement en mémoire;
- quelques mécanismes syntaxiques simples, à base des macros précédentes pour l'émission d'arguments implicites.

Tous ces points sont interdépendants et seront donc présentés de façon légèrement circulaire. Il faut aussi noter que ne sera décrit maintenant que ce qui est inséparable de l'essence du noyau mou: tout ce qui est contingent ne sera décrit qu'aux chapitres [#14 et #15].

(1) Registered trademark.

11. LE DUAL.

Il définit les grandes catégories de slots et facettes, les principaux concepts du langage: attributs, méthodes, applicateurs, réflexes et facettes.

11.1. Généralités.

Lors de la création d'un univers, tout concept dual (clé de 1-er ou 2-ième niveau) est affecté à l'une de ces grandes catégories (ou sous-catégories).

11.1.1. Obligation de consistance.

Le système ne garantit rien en cas de définition contradictoire, ou d'utilisation contradictoire d'un objet. Un slot présent avec des facettes valeur et méthode peut se retrouver attribut, méthode ou les 2, sans que ce soit prévisible, ou qu'une erreur (LISP ou YAFOOL) soit déclenchée. Des vérifications de ce genre sont possibles, bien que très coûteuses pour la phase de création d'univers.

11.1.2. Facette-standard.

Tous les éléments du dual, à l'exception des réflexes et des facettes, ont un attribut particulier, *facette-standard* qui désigne, comme son nom l'indique, sous quelle facette se trouve la vraie valeur (éventuellement fonctionnelle) du slot. Elle permet l'élimination de la facette [#4.5.2.1 et #11.5.1].

Cette facette-standard est un critère de classement des clés de 1-er niveau en phase de création d'univers.

11.2. Attributs.

C'est la catégorie implicite des clés de premier niveau. Leur *facette-standard* est *valeur*. Attribut est l'élément du dual dont la décomposition est la plus importante (fig. 11).

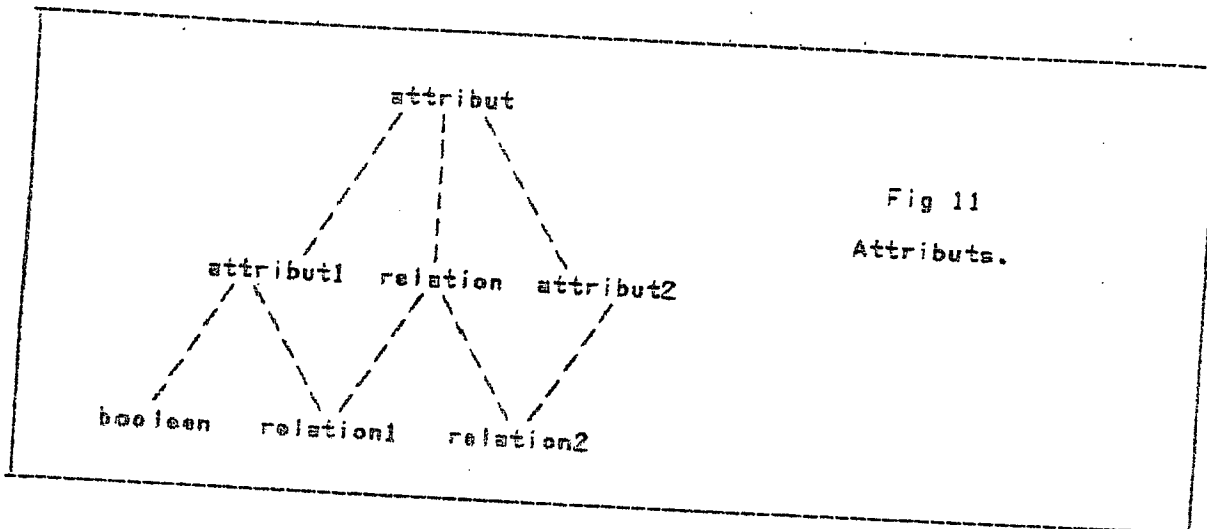


Fig 11
Attributs.

11.2.1. Attribut1 / attribut2.

Ils sont partitionnée en 2 classes, suivant la multiplicité de leur valeur: attribut1 pour les valeur uniques et attribut2 pour les multiples. Cette partition est faite automatiquement à la première affectation de valeur au slot, suivant que celle-ci est un atome ou une liste.

N.B. Si l'on veut un attribut à valeur unique de liste (voir la discussion de ce problème en [#4.3]), il faut donc le forcer comme étant un attribut1 (cf en [#12.3.2 et #12.3.3] les facettes non standards). Booléem est décrit dans les extensions [# 15].

11.2.2. Relation.

C'est une sous-catégorie d'attribut, pour les liens. Cette sous-catégorie est actuellement repérés par l'existence d'un attribut lien-inverse qui désigne l'inverse du lien: toute relation a un lien inverse. Il est bien sûr possible d'élargir la définition.

Leur définition est donc plus restrictive que celle donnée pour les liens [#3.1.3]. De façon plus générale, on pourrait dire qu'une relation est un lien dont les affectations doivent être précédées d'un effacement: il faut donc utiliser FPUT-L (ou -L1) et non FPUT [#6.4]. On retrouve cette notion de relation (et de son inverse) dans SRL2 [Wright 84].

11.2.2.1. Relation1 / relation2.

Comme attribut, relation est partitionné en 2: relation1 qui est aussi un attribut1, et relation2 qui est aussi un attribut2.

11.2.2.2. Exemples.

Est-un, et son inverse instanciation sont des relation2, lien-inverse une relation1.

11.3. Comportement.

Il n'a pas semblé nécessaire de définir un objet comportement. Il est décrit par 2 catégories distinctes: methode et applicateur, qui portent toutes deux le nom de leur facette-standard. Elles sont à valeur fonctionnelle unique: symbole possédant une valeur fonctionnelle ou λ -expression. Elles possèdent une méthode particulière, comport-error qui définit l'action à entreprendre en cas d'absence du comportement cherché [#7.2.1].

11.3.1. Méthodes.

11.3.2. Appicateurs.

De nom applic, ils possèdent en plus un attribut, reflex-applic, dont la valeur est une fonction de mapping (ANY, MAPC ou EVERY) indiquant le mode de déclenchement des réflexes associés, dont le nom est formé par la concaténation de "si-" et du nom de l'appicateur.

11.4. Réflexes.

Trois sont standards: si-ajout, si-besoin et si-enleve. En création d'univers, ils sont reconnus à leur nom, commençant par "si-". Cette création fait une permutation circulaire, du triplet frame-slot-réflexe au

triplet slot-réflexe-frame. A part si-besoin qui est réservé aux attributs, les réflexes sont attachables à n'importe quelle catégorie d'objets.

Ils sont tous à valeur fonctionnelle multiple.

11.5. Facettes.

Trois sont standards: value, sauf et hérit. Une sous-catégorie, facette-user, est utilisée pour toutes les facettes reconnues, par défaut, en création d'univers, c'est-à-dire pour les clés de 2-ième niveau inconnues.

11.5.1. Facettes et slots standards.

Si une facette est liée de façon univoque à un slot, qu'on appellera slot standard, il est possible d'y accéder sans mentionner le slot. Cette élision se fait au niveau des primitives elles-mêmes, comme pour la facette-standard [§11.1.2] ou bien par les applicateurs d'accès.

N.B. Dans le cas où aucune expansion satisfaisante de la macro n'est possible, le message d'erreur "expansion anormale pour ..." est édité, et la facette value supplée à l'argument manquant.

N.B. La facette possède dans ce cas la sémantique d'un attribut, l'attribut lui-même pouvant voir son rôle réduit au regroupement physique de ses facettes dans la A-liste des slots de l'objet. Un bon exemple est celui de l'attribut univ-stat de univers (chapitre suivant) qui possède une douzaine de facettes. Une telle implémentation permet un accès aux autres slots statistiquement plus rapide.

11.6. Slot-auto-load.

C'est une classe qui contient tous les objets, slots ou facettes en auto-load, définis dans des extensions du langage. Tous les comportements standards (:applicateurs [§13] et méthodes de création d'univers [§12]) d'accès à ces slots déclenchent le chargement des fichiers de l'attribut in-file. Ces mécanismes sont décrits en détail au paragraphe [§15.5].

12. UNIVERS.

Leur rôle est triple:

- rendre modulaire l'ensemble des objets définis par l'utilisateur en lui permettant de manipuler des sous-treillis (avec racine unique) du graphe d'héritage.
- assurer la dualisation de façon automatique et aussi "intelligente" que possible.
- permettre une certaine ~~meta~~-circularité de la définition du noyau mou: l'univers noyau du système, tel qu'il est décrit aux chapitres [#11 et #14], se définit en grande partie lui-même.

12.1. Les modèles.

On appellera modèle un objet défini en phase de création d'univers. Les modèles ne sont absolument pas formalisés: par rapport à la distinction classe / instance [#3.1.3], les modèles ne sont pas les seules classes mais ce sont les seules qui peuvent définir des concepts (slots ou facettes) nouveaux.

N.B. Par abus de langage on appellera aussi modèle d'un objet l'un des éléments de son lien est-un.

12.2. L'objet univers.

C'est la catégorie des racines d'univers: il possède des attributs statistiques (univ-stat) sur les objets qu'il contient, le répertoire du système de fichiers hôte qui contient les fichiers de définition de l'univers (attribut im-lib) ainsi qu'un attribut autovalué (dont la valeur est celle de l'univers), lei-univers qui permet à ses descendants de reconnaître aisément leur univers.

Cet objet possède l'intéressante propriété de causer un cycle dans le graphe d'héritage: c'est-un racine de l'univers noyau, qui lui-même est-un univers !

Tous les objets duaux d'un univers contiennent celui-ci dans leur hiérarchie.

L'attribut univ-stat définit un certain nombre de facette-user, une par grande catégorie duale, de même nom que la catégorie, avec un "s" en plus. Chacune de ces facettes contient la liste des éléments de la catégorie pour l'univers considéré. Un attribut univ-stat-sté permet d'associer à chaque catégorie sa facette statistique.

12.3. La création d'univers.

C'est la phase de création de tous les concepts définis dans un univers: modèles d'objets, slots et facettes. Il n'est pas possible de créer de nouveaux éléments duaux en-dehors de cette phase (voir plus loin, la création en insertion).

C'est une sorte de compilation, qui se fait en 4 passes. Elle débute par la fonction **BIG-BANS**.

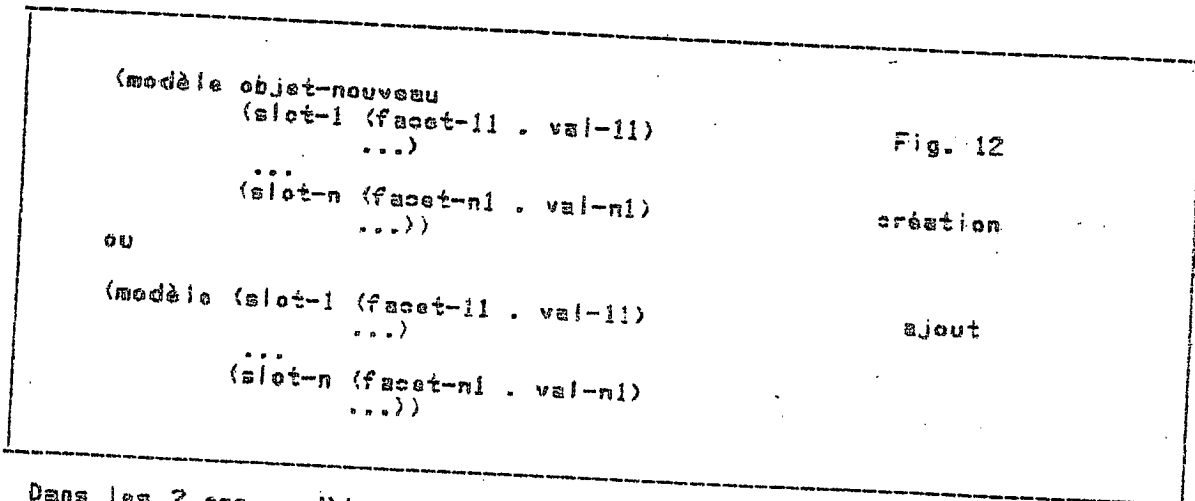
La syntaxe de BIG-BANG est la même que celle de la figure 12 (forme du haut), BIG-BANG remplaçant modèle. objet-nouveau étant le nom de l'univers à créer, et le rôle de modèle étant tenu par l'univers courant au moment de la création, valeur de la variable globale #:YAFDOL:IDEAL.

La fonction BIG-BANG définit un nouveau TOP-LEVEL sur le canal d'entrée courant, ainsi que quelques variables globales [cf 19.7.1]. Elle s'interrompt à la lecture d'un atome (1), ou en cas de fin de fichier sur ce même canal.

12.3.1. Passe 1: initialisation des objets.

C'est la passe d'initialisation des valeurs d'objet des objets à définir, et de leur lien est-un.

Chacune des formes lues a l'une des structures de la figure 12.



Dans les 2 cas, modèle est un nom d'objet déjà défini (2), éventuellement dans le même univers, et les valeurs aussi bien que les listes de facettes peuvent être absentes. La forme lue est mémorisée, pour les passes suivantes.

Dans le premier cas, objet-nouveau est l'objet à définir, avec comme valeur d'objet la liste de slots qui suit, et comme premier lien est-un, en plus de ceux qui peuvent figurer dans les slots présents, modèle. La pré-existence de objet-nouveau en tant qu'objet produit des effets imprévisibles, outre l'écrasement de son ancienne valeur d'objet.

Dans le deuxième cas, il n'y a pas de création d'un nouvel objet, mais ajout de nouvelles propriétés. Les nouvelles propriétés vont être ajoutées à l'ancienne définition de modèle, facette par facette. Cette création en ajout sert à définir pour un univers précédent des propriétés nouvelles.

La passe 1 s'interrompt avec le premier atome rencontré en lecture, ou bien la fin de fichier, sur le canal d'entrée courant, qui déclenche les 3 autres passes, et un échappement pour sortir du TOP-LEVEL défini par BIG-

(1) Si cet atome est NIL, il y a déchargement de toutes les fonctions de création d'univers, et remise en auto-load de BIG-BANG et INSERT-UNIVERS. Voir README-FILE-UNIV [p.18.4].

(2) Toutes les formes dont les CAR n'est pas un objet sont évaluées normalement: il est ainsi possible en mode interactif de faire autre chose que créer des objets.

BANG.

12.3.2. Passe 2: init-slot et init-facet.

C'est la passe de reconnaissance des clés de 1-er niveau (slot) et d'initialisation de leur lien est-un. Cette reconnaissance se fait essentiellement par les facettes standards présentes comme clés de 2-ième niveau. Les facettes non standards, c'est-à-dire les clés de 2-ième niveau (autres que les réflexes) qui ne sont pas connues comme étant des facettes, sont mémorisées (triplet d'accès), pour la passe suivante.

Cette passe se fait par application des méthodes init-slot (resp. init-facet) à toutes les clés de 1-er (resp. 2-ième) niveau des formes lues mémorisées en phase 1. Plus précisément, la méthode init-slot est appliquée à tous les slots lus, et son effet habituel, ainsi que celui de sa méthode caract-error est d'appliquer init-facet à toutes ses facettes, ce qui permet d'en déduire la catégorie du slot, si celui-ci est inconnu.

12.3.3. Passe 3: caract.

Cette passe commence par l'initialisation comme attribut de toutes les clés de 1-er niveau encore inconnues.

Toutes les facettes non standard sont passées en revue, avec application de la méthode caract à la facette, avec comme argument le slot, l'objet et la valeur. La méthode comport-error de caract sert à définir les clés encore inconnues en tant que facette-user.

```
(modèle objet
  (slot-1 (value . aaa)
    (si-ajout fct ((liste-de-forme)))
    (slot-2 (a b c) (liste-de-forme)))
  (slot-2 (methode . yyy)))
```

Fig. 13
avant

Le rôle de caract consiste à dualiser les réflexes (permutation circulaire du triplet), et pour les autres slots, à transformer le triplet frame-slot-facette en slot-facette avec la facette standard de la facette. Enfin de légères corrections sont faites pour les valeurs fonctionnelles: rajout de lambda, et de la liste d'arguments (NIL) pour les réflexes standards (fig. 13 et 14).

```
(objet (est-un (value modèle))
  (slot-1 (value . aaa))
  (slot-2 (methode . yyy)))

(slot-1 (est-un (value attribut))
  (si-ajout (objet fct
    (lambda () (liste-de-forme))))
  (slot-2 (methode lambda (a b c) (liste-de-forme))))

(slot-2 (est-un (value methode)))
```

Fig. 14
après

12.3.4. Passe 4: creer-slot et creer-facet.

Aucun réflexe n'a été déclenché pendant les 3 premières passes, au moins pour tous les objets, slots et facettes nouveaux: c'était impossible puisque les nouveaux réflexes n'étaient pas encore dualisés, les facettes non standards non plus.

C'est donc essentiellement la passe de déclenchement des réflexes si-ajout sur les liens est-un de tous les objets nouveaux (duaux compris), et sur toutes les facettes non standards. Ces dernières sont alors effacées (en tant que clé de 2-ième niveau).

Cette passe est réalisée avec les méthodes creer-slot et creer-facet qui sont dans le même rapport que les méthodes init-slot et init-facet de la passe 2.

12.3.5. Les méthodes de création d'univers.

Il y en a 5, dont une seule, caract, est utilisable en dehors de la création. Elles sont toutes redéfinissables et extensibles.

12.3.5.1. Init-slot.

C'est la méthode d'initialisation des clés de 1-er niveau. Son action générale consiste à faire appel à init-facet sur toutes les clés de 2-ième niveau du slot. Si le slot est encore inconnu (pas de lien est-un), la première valeur non-NIL retournée par une de ces méthodes init-facet est affectée à son lien est-un.

N.B. Init-slot, comme la méthode suivante, ainsi que leurs méthodes comport-erreur, ne peuvent être en facette non standard d'un dual apparaissant en clé de 1-er niveau (pour init-slot) ou de 2-ème niveau (pour init-facet), sous peine de ne pas voir cette méthode s'appliquer à cette clé, pour l'univers en cours.

12.3.5.2. Init-facet.

C'est la méthode d'initialisation des clés de 2-ième niveau. Elle retourne en valeur la catégorie (ou la liste de catégorie) que l'on peut déduire pour le slot de la présence de la facette.

12.3.5.3. Caract.

Pour les slots qui ont une facette standard, (caract slot frame val) a pour effet d'affecter au SLOT de FRAME la valeur VAL, sous la facette-standard du slot.

Pour les réflexes, (caract rflx frame slot val) a pour effet d'affecter au RFLX de SLOT, sous la facette FRAME, la valeur VAL.

Pour toutes les valeurs fonctionnelles, VAL est aussi vérifié (présence de LAMBDA si ce n'est pas un atome).

Enfin, il faut noter que caract ne déclenche pas de réflexe si-ajout pendant la phase de création d'univers, mais qu'il le fait en dehors, et que caract est aussi défini pour 2 extensions: les valeurs et les booleans [#15.4]. Voir aussi la fonction F-CARAC [#19.5] et la méthode création [#15.2.1]. La méthode comport-erreur de caract est double: en phase de création d'univers, elle crée une facette-user, alors qu'en temps normal elle imprime le message d'erreur: "... est une

caractéristique incorrecte".

N.B. La remarque faite pour `init-slot` et `init-facet` est valable pour `caract`. Sa présence en facette non standard d'un slot figurant lui-même en facette non standard dans l'univers en cours donne des résultats imprévisibles.

12.3.5.4. Créer-slot.

C'est la méthode de "création" des clés de 1-er niveau. Comme `init-slot` avec `init-facet`, elle fait appel à `creer-facet` sur chacune des clés de 2-ième niveau. Pour le lien `est-un` elle déclenche les réflexes `si-ajout` sur les facettes `value` et `sauf`.

12.3.5.5. Créer-facet.

C'est la méthode de "création" des clés de 2-ième niveau. Si la facette est non standard, il y a activation des réflexes `si-ajout` sur le triplet `slot / facette non standard du slot / facette standard` de cette dernière, de même que pour les réflexes.

12.4. Les CLES comme des MOTS-CLES.

La création d'univers ainsi définie permet d'utiliser les clés (slots ou facettes) comme des mots-clés, leur rôle pouvant alors se limiter à cette phase de "compilation". Les chapitres suivants (en particulier [015 à 18], montreront de nombreux exemples d'une telle utilisation.

12.5. Insertion dans un univers.

Il peut être nécessaire d'élargir la définition d'un univers existant, en y rajoutant de nouveaux slots ou facettes. La fonction `INSERT-UNIVERS`, avec comme argument l'univers à compléter, a le même effet que `BIG-BANG`: initialiser une phase de création d'univers sur le canal d'entrée courant. C'est le mécanisme de base des autoloads (partie suivante).

En cas d'insertion, les fonctions de création d'univers ne sont déchargées que si elles n'étaient pas chargées auparavant.

12.6. La hiérarchie des univers.

Le nom du dernier univers créé, ou en cours de création / insertion, est la valeur de la variable globale #:YAF00L:IDEAL. Chaque nouvel univers est une instance de #:YAF00L:IDEAL: ils sont donc tous imbriqués. Si l'on veut modifier cette hiérarchie, il faut redéfinir #:YAF00L:IDEAL avant l'appel de BIG-BANG., en lui donnant le nom d'un autre univers et/ou donner explicitement d'autres modèles à l'univers créé. La figure 15 montre une séquence de création d'univers, et le graphe résultant.

```

? #:univers:ideal
= objet-ideal
? (big-bang univers1)
...
+
= univers1
? #:univers:ideal
= univers1
? (setq #:univers:ideal objet-ideal)
= objet-ideal
? (big-bang univers2)
...
+
= univers2
? (big-bang univers3
  (est-un (value univers1)))
...
+
= univers3

```

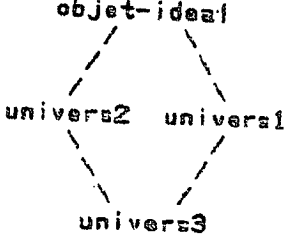


Fig. 15

12.6.1. Univers-contenus et univers-contenant.

Ces 2 relations inverses décrivent la hiérarchie des univers: univers-contenant est à valeur unique et pointe sur la valeur de #:YAF00L:IDEAL avant le BIG-BANG. Univers-contenus est multiple.

13. MACROS ET APPLICATEURS D'ACCES.

L'idée de ce chapitre est d'automatiser et de simplifier les accès aux objets de façon performante: on en trouvera l'application la plus intéressante dans les extensions temporelles [16]. L'idée de départ repose sur les applicateurs de MERING [Ferber 83]: définir tous les accès élémentaires (lecture, écriture, effacement etc..) comme des comportements attachés au slot et / ou à l'objet.

13.1. Généralités.

On retrouve ce problème dans la plupart des LOO, où les accès aux attributs ne se font pas par appel de primitive mais par envoi de message. Il est ainsi possible de définir des accès personnalisés pour les slots et/ou les objets.

Le principal intérêt de cette démarche est de permettre d'"oublier" toutes les primitives et de manipuler des accès d'un niveau plus évolué: syntaxe allégée et uniforme pour une sémantique plus puissante.

Malheureusement, le prix est élevé s'il faut, à chaque appel, faire une coûteuse recherche dans la hiérarchie du slot et/ou de l'objet.

13.2. Macros.

LISP offre une technique éprouvée pour résoudre ces problèmes d'allègement de la syntaxe couplée avec des performances aussi bonnes que sans cet allègement: les macros.

Une macro est une fonction LISP qui présente 2 particularités remarquables: l'évaluateur LISP réévalue la valeur retournée par la macro (et ce tant qu'il est retourné une valeur dont le CAR est une macro), et la liste des paramètres de la macro est liée à la liste d'appel de la fonction, et non pas à la liste de ses arguments, qui ne sont d'ailleurs pas évalués. Le corps de la macro peut donc modifier son propre appel, en remplaçant (DISPLACE) la forme d'origine par son expansion, c'est à dire la valeur retournée par la fonction.

Pour les non-initiés, voir [Winston 84] et [Chailloux 85].

13.3. Obligation de consistance.

Le principe de l'expansion des macros est tel que l'on peut utiliser une même macro avec des expansions différentes en différents endroits d'un programme, mais qu'il est indispensable que pour une expansion donnée, tous les appels futurs de la forme expansée doivent donner le même résultat qu'avec la forme d'origine.

13.4. Principes d'implémentation.

13.4.1. Slots, macros et applicateurs.

L'idée de base de ce chapitre est la suivante: tout élément du dual a une valeur fonctionnelle de macro. Cette macro, utilisée dans son évaluation des applicateurs dont l'un des arguments est la forme d'origine

de l'appel ce qui leur permet de la modifier à volonté.

13.4.2. Macro-caractère ":".

Tous les applicateurs actuellement implémentés ont un nom commençant par ":" suivi d'un caractère non alphabétique. Le macro-caractère de package ":" a donc été redéfini pour tenir compte de ce fait. Si le caractère suivant est alphabétique, il a le rôle que lui assigne habituellement Le_Lisp [Chailloux 85]. Vu cette convention de nom, on les appellera des :applicateurs.

13.4.3. Syntaxe générale.

Elle est la suivante: (:applie slot frame [facet] . args).

13.4.3.1. Ordre d'évaluation.

On note que l'ordre habituel du triplet frame-slot-facet est ici inversé au profit de slot-frame-facet. C'est du au fait que dans l'accès le plus courant, le :applicateur est omis et remplacé par le slot. L'ordre d'évaluation n'est donc pas le même que celui de l'écriture de la macro.

13.4.4. Expansion avec évaluation.

L'expansion de ces macros se fait avec évaluation: il se pose donc un certain nombre de problèmes, en ce qui concerne l'éventuelle double évaluation et la compilation.

13.4.4.1. Valeur déplacée et valeur retournée.

Si la macro retourne la forme qu'elle DISPLACE, ses arguments seraient évalués 2 fois: une fois pour l'expansion et une seconde pour l'évaluation de la forme déplacée.

Il est donc nécessaire que ces macros retournent une valeur autre que celle qui est déplacée: la même, avec remplacement des arguments par leur valeur.

13.4.4.2. Problèmes de compilation.

A la compilation, il est possible que l'un des arguments ne soit pas connu, ou bien que la forme d'expansion ne soit prévisible (par ex. pour :?, si le slot est un attribut: voir plus loin). Dans ce cas, l'expansion se fait avec un appel à l'évaluateur, ce qui repousse le problème à l'évaluation.

N.B. Deux fonctions de compilation, MY-COMPILE-ALL-IN-CORE et MY-COMPILEFILES [#20.5.1], ainsi qu'un indicateur de compilation #:YAFUOL:COMPILE-FLAG [#20.9] sont documentées en annexe..

13.4.5. Elision de FRAME+.

D'une façon générale, tous les :applicateurs considèrent que l'absence d'un argument doit être compensée par l'insertion de FRAME+ en 2-ième position.

Seules les exceptions ou restrictions à cette règle seront notées au passage.

13.5. Les réflexes des :applicateurs.

Le paragraphe [7.3] soulignait la nécessité des réflexes pour justifier la présence d'applicateurs.

Ce chapitre semble démentir cette assertion puisqu'il n'y a pas de réflexe de :applicateur dans le noyau. On en trouvera dans les extensions.

13.6. Les différents :applicateurs.

Ce paragraphe va définir tous les :applicateurs de l'implémentation actuelle: il s'agit donc de choix éminemment arbitraires et redéfinissables.

13.6.1. La macro générale.

Tout slot, à l'exception des :applicateurs, a pour valeur fonctionnelle une macro (SLOT-MACRO), dont le seul rôle consiste à faire appel au :applicateur ::. Cette macro rajoute seulement :: en tête de la forme appelante.

13.6.2. Les macros des :applicateurs.

Leur macros sont légèrement plus compliquées, et s'expansent sous la forme:

```
(eval (fappl objet slot :applie cell . args)).
```

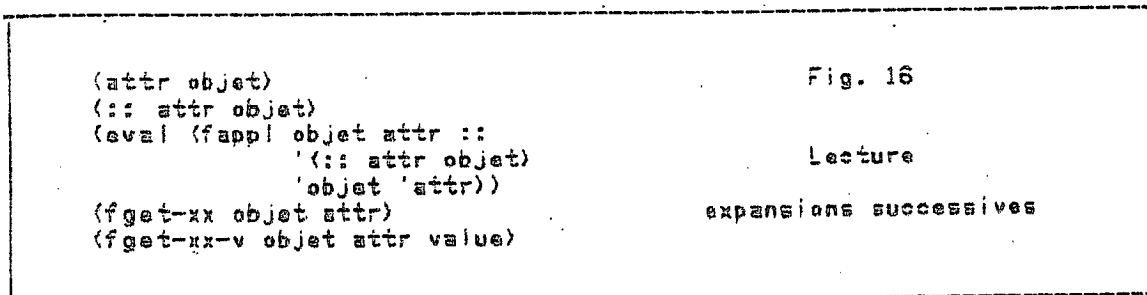
Où objet et slot sont les paramètres d'appel de l'accès, :applie le nom de l'applicateur, cell la liste d'appel (qui sera modifiée) et args le reste des arguments.

13.6.3. Le :applicateur ::, lecture et déclenchement.

C'est l'applicateur de lecture des attributs et de déclenchement des comportements.

13.6.3.1. Lecture des attributs.

Si attr est un attribut, l'expansion due à :: provoque un appel à FGET-xx où -xx est la valeur de l'attribut type-de-recherche de l'attribut.



13.6.3.2. Déclenchement des comportements.

L'expansion est simple et consiste à générer un appel à FMETH ou FAPPL, suivant que le slot d'appel est une méthode ou un applicateur.

L'élision de **FRAME*** ne marche que s'il n'y a aucun argument (pour les méthodes), ou un seul (pour les applicateurs).

13.6.4. Le :applicateur := , affectation.

Pour l'affectation, := s'expande avec **FPUT+**, **FPUT-L** ou **FPUT-H**, suivant qu'il s'agit d'un slot quelconque, d'un lien ou de **est-un**.

13.6.5. Le :applicateur :- , retrait et effacement.

Il fait une expansion avec **FREM+** (**FREM-H** pour **est-un**), en suppléant l'argument de valeur **T** s'il manque un argument. L'élision de **FRAME*** ne s'applique qu'après.

(:- attr objet val) (frem+ objet attr val)	Retrait
(:- attr objet) (frem+ objet attr t)	Effacement
(:- attr) (frem+ frem+ attr t)	Fig. 17

N.B. Pour les attribut2, :- utilise en fait **FREM+**: il faut donc se méfier si l'attribut est à valeur liste de liste.

13.6.6. Le :applicateur :+ , ajout.

Ce :applicateur fait un ajout (**FADD++** ou **FADD-H**) sur tout élément du dual, à l'exception des attribut1 pour lesquels il déclenche un message d'erreur.

N.B. :+ utilisant **FADD++**: il faut se méfier si l'attribut est à valeur liste de liste.

13.6.7. Le :applicateur :? , vérification.

C'est le :applicateur de vérification de valeur: il s'expande à partir de du :applicateur :=, avec un **EQ** ou **MEMQ** suivant qu'il s'agit d'un attribut1 ou d'un attribut2.

(:? attr1 objet val) (when (eq val (:= attr1 objet)) val)	Vérification.
(:? attr2 val) (memq val (:= attr2))	
	Fig. 18

Si le slot est un attribut (ou relation) mais n'est ni un attribut1, ni un attribut2, ce qui signifie (sauf erreur) qu'il n'a pas encore reçu de valeur, l'expansion est **NIL** et ne fait pas de **DISPLACE**.

13.6.8. Le :applicateur :< , fermeture transitive des liens.

L'expansion se fait avec FGET-H pour est-un, FGET-L pour les relation2, FGET-H1 pour relation1.

13.6.9. Les :applicateur :> et :>>, super-comportement.

L'expansion se fait avec un appel à FMETH> ou FAPPL>, avec comme argument de hiérarchie biaisée (cadr herité) pour le premier et (fget-h0 (cadr herité) est-un) pour le second [7.4.1].

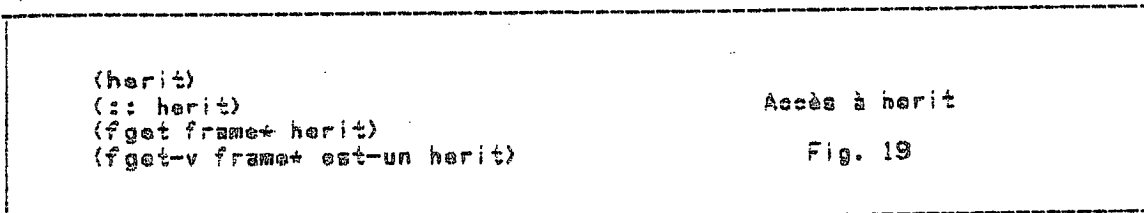
L'élimination de FRAME+ est sans objet.

13.7. Pseudo :applicateurs: :/ , :% et :?? .

Ce sont 3 macros qui s'expandent avec FRPLAC+, FMOD+ et pour la dernière comme :? pour attribut2, en remplaçant :: par :<. C'est donc un test d'appartenance à la fermeture transitive d'un lien.

13.8. Accès à des facettes: le slot standard.

Lorsqu'une facette est définie dans un univers, il lui est associé par l'attribut liste-frame [14.2], la liste des slots dans laquelle cette facette était présente. Lorsque cette liste est réduite à un seul élément, le slot peut être éliminé dans tous les accès à cette facette. Cette élimination est valable aussi pour les primitives (cf [4.2.2.1], le suffixe "-V"). Le slot est alors appelé slot standard de la facette. Dans le noyau, cette situation est vraie pour herit et pour toutes les facettes spécifiques de univ-stat.



Cette possibilité s'étend à tous les accès à un triplet frame-slot-facette.

14. L'UNIVERS NOYAU.

Ce chapitre décrit les objets minimum du noyau existant, tels qu'ils apparaissent dans les chapitres précédents. Le reste est défini comme des extensions.

Le nom de l'univers noyau est `objet-ideal`. Un objet particulier, `dual-ideal` regroupe tous les éléments du dual.

L'univers, tel qu'il est décrit ci-dessous est cohérent, mais il ne tient pas compte de son auto-définition: ainsi, `caract`, `init-slot` et `init-facet` n'ont pas de `comport-error`, qui sont en facette non standard, et donc inconnues pendant les 3 premières passes de la création. En fait, ce sont aussi ces 3 méthodes qui sont inconnues dans ces 3 premières passes, quand elles sont en facettes non standard.

De même l'attribut `univ-stat-std` n'est pas indiqué (il figure dans toutes les catégories du dual auxquelles sont associées une facette de `univ-stat` (voir univers).

Les objets ne seront pas commentés outre mesure, on se référera donc au chapitres précédents.

14.1. Objet-ideal.

C'est l'objet suprême, l'abstraction fondamentale, la racine du graphe d'héritage, bien qu'il soit lui-même un univers.

```
(big-bang objet-ideal
  (est-un (si-ajout ((demon+isa)))
    (si-enleve ((demon-isa)))
    (type-de-recherche . "")
    (herit)
    (:= . appl=isa)
    (:+ . appl+isa)
    (:- . appl-isa)
    (< . appl<isa)
    (lien-inverse . instantiation)
    (creer-slot . meth-slot+value)
    (init-facet . meth-init-isa))
  (instantiation (si-ajout ((demon+inst)))
    (si-enleve ((demon-inst)))
    (lien-inverse . est-un))
  (init-facet (methode . meth-init-non-std)
    (comport-error . meth-init-new-non-std))
  (:loaded-from-file (value . #.#:system:loaded-from-file))
  (in-lib (value . #.:yaf-dir))
  (univ-dual (value . dual-ideal))
  (ici-univers)
  (liste-slot (type-de-recherche . "-a")))
```

On y voit la définition de `est-un` et de son lien inverse: `instantiation`. Leurs réflexes d'écriture (`DEMON.ISA` et `DEMON.INST`) réciproques leur

assurent une stricte symétrie.

Liste-slot contient la liste des slots nouveaux (du nouvel univers) définis dans un modèle.

L'attribut univ-dual associe à tout univers la racine duale de l'univers, c'est-à-dire un objet qui est l'ancêtre de tous les duaux de l'univers.

Pour les autres univers que le noyau, son nom est formé par la concaténation de "dual-" et du nom de l'univers, et cet objet est une instance de la racine de l'univers et de dual-ideal. Il est créé automatiquement.

14.2. Le dual.

C'est l'objet dual le plus abstrait: il comporte la définition des applicateurs les plus généraux.

```
(objet-ideal dual-ideal
  (init-slot (methode . meth-init-dual)
             (comport-error . meth-init-error))
  (creer-slot (methode . meth-slot-dual))
  (creer-facet (methode . meth-facet-non-standard))
  (liste-frame (lien-inverse . liste-slot)
               (si-ajout ((demon+des-liens))))
  (si-besoin (si-ajout demon-ajout-besoin))
  (:? (applic . appl?attr1))
  (:= (applic . appl=dual)
      (reflex-applic . mapc))
  (:: (applic . appl.dual)
      (reflex-applic . mapc))
  (:- (applic . appl-dual)
      (reflex-applic . any))
  (:+ (applic . appl+dual)
      (reflex-applic . any)))
```

Liste-frame assure l'inverse de liste-slot: c'est la liste des modèles dans lesquels un nouveau slot est défini, et la liste des slots dans lesquels une nouvelle facette est définie.

14.2.1. Les réflexes.

```
(dual-ideal reflexe
  (creer-facet (methode . meth-facet-reflex))
  (caract (methode . meth-caract-rlx)
          (comport-error . meth-caract-error)))

(reflexe si-besoin)
```

Si-besoin n'étant pas utilisé dans le noyau, il faut le "déclarer" explicitement.

14.2.2. Les attributs.

```
(dual-ideal attribut
  (facette-standard (value . value))
  (type-de-recherche)
  (lien-inverse (si-ajout demon-auto-inverse demon-lien-1))
  (:? (applic . meth-nil))
  (:= (applic . appl=attr))
  (multiplicite (methode . meth-mult-attr))
  (init-slot (methode . meth-inits-attr))
  (caract (methode . meth-caract-attr))
```

14.2.2.1. Multiplicité des attributs.

```
(attribut attribut1
  (:? (applic . appl?attr1))
  (:< (applic . appl<rel1))
  (:+ (applic . appl+attr1))
  (multiplicite (methode . meth-nil))
  (init-slot (methode . meth-inits-dual)))

(attribut attribut2
  (:? (applic . appl?attr2))
  (:< (applic . appl<rel2))
  (:- (applic . appl-attr2))
  (multiplicite (methode . meth-nil))
  (init-slot (methode . meth-inits-dual)))
```

La méthode multiplicité est définie pour différencier attribut1 et attribut2 ...

N.B. Le :applicateur + déclenche une erreur pour attribut1.

14.2.2.2. Relation.

... et relation1 et relation2.

Sur-lien et sous-lien sont 2 relation2s des relations qui dénotent des implications entre 2 relations: si A est un sur-lien de B, (B est un sous-lien de A), alors, pour tout objet X, toute valeur de B est valeur de A. En d'autres termes, B définit un sous-graphe du graphe défini par A. Les relations inverses et réflexes d'écriture correspondants sont générés automatiquement. Une petite bibliothèque de réflexes pour les relations inverses est donnée en annexe [§20.8].

N.B. On notera que relation1 (resp. relation2) est-un attribut1 (resp. attribut2) avant d'être une relation.

```

(attribut relation
  (sur-lien
    (lien-inverse . sous-lien)
    (si-ajout
      ((demon+des-liens)
        (fmapc caract si-enleve value+ (ici-univers)
          (cirlist
            `(((:- ,frame+ frame+ value+))))))
        (caract si-ajout frame+ (ici-univers)
          (ncons
            (fmapcar list :+ value+ 'value+))))))
    (sous-lien (si-ajout ((demon+des-liens)
      (:+ sous-lien (lien-inverse)
        (fmapcar lien-inverse value+))))))
  (multiplicite (methode . meth-mult-rel))
  (init-slot (methode . meth-init-rel))
  (lien-inverse (lien-inverse . lien-inverse)))

(attribut1 relation1
  (est-un (value relation)))

(attribut2 relation2
  (est-un (value relation))
  (:= (methode . appl=rel)))

```

14.2.2.3. Type-de-recherche.

L'attribut type-de-recherche définit le suffixe de la fonction FSET à utiliser, pour un slot donné, dans le :applicateur ::. S'il possède une valeur chaîne de caractères (en minuscules) définie par l'utilisateur, celle-ci prime tout. Sinon, il prend la valeur -Z en cas de présence de réflexes si-besoin sur le slot et sinon, -I en cas de présence de facette value. Dans tous les autres cas, l'accès se fait sans héritage ni réflexe.

14.2.3. Comportements.

```
(dual-ideal methode
  (facette-standard (value . methode))
  (init-facet (methode . meth-init-comport))
  (creer-facet (methode . meth-facet-comport))
  (:= (applic . appl.meth))
  (:> (applic . appl>meth))
  (comport-error (methode . meth-meth-error))
  (caract (methode . meth-caract-meth))

(dual-ideal applic
  (facette-standard (value . applic))
  (init-facet (methode . meth-init-comport))
  (creer-facet (methode . meth-facet-comport))
  (:= (applic . appl.applic))
  (:> (applic . appl>applic))
  (comport-error (methode . meth-appl-error))
  (caract (methode . meth-caract-appl))
  (reflex-applic))
```

La définition de `comport-error` dans `methode` assure la bonne implémentation des primitives `FMETH` et `FAPPL` [§7.2.1].

14.2.4. Facettes.

```
(dual-ideal facette
  (init-facet (methode . meth-nil))
  (creer-facet (methode . meth-nil))
  (?: (applic . appl?facette)))

(facette facette-user
  (creer-facet (methode . meth-facet-user)))

(facette value
  (init-facet (methode . meth-init-attr))
  (creer-facet (methode . meth-facet-value)))

(facette sauf
  (init-facet (methode . meth-init-sauf)))

(facette herit)
```

14.3. Univers.

Outre son attribut de statistique univ-stat, il possède un attribut in-lib indiquant pour les slot-autoload la directory dans laquelle se trouve les fichiers à charger (voir chapitre suivant) et les relations univers-contenus et -contenant qui redoublent la hiérarchie des univers.

```
(objet-ideal univers
  (univers-contenant (si-ajout ((demon+inverse)))
    (lien-inverse . univers-contenus))
  (univers-contenus)
  (univ-stat (reflexes)
    (attributs)
    (attribut1s)
    (attribut2s)
    (relations)
    (relation1s)
    (relation2s)
    (methodes)
    (facettes)
    (facette-users)
    (applica)
    (objets)
    (slot-autoloads))
  (in-lib))
```

Il faut noter que toutes les facettes de univ-stat l'ont comme slot standard: un accès à l'une de ces facettes sans mentionner le slot est donc licite [11.5.1].

QUATRIEME PARTIE

LES EXTENSIONS

YAF00L 2.1

15. LES EXTENSIONS.

Chaque extension est décrite avec sa définition YAFODL exacte. Les fonctions associées ne seront décrites que par leurs effets. L'extension temporelle, par sa taille, est repoussée au chapitre suivant.

15.1. Extension de l'héritage.

Ce paragraphe décrit 5 facettes, `like`, `init`, `init-eval`, `value-of`, `herit-from` et `herit-by`, qui permettent d'étendre les mécanismes de partage de la connaissance de façon assez simple. Elles sont toutes basées, à part la première, sur l'écriture de réflexes, et sont présentées essentiellement en tant qu'exemples, les variations sur ce thème étant innombrables.

Toutes ces extensions sont trop petites pour qu'il soit intéressant de les mettre en autoload.

On peut considérer que ces facettes sont une bonne solution de remplacement pour la facette défaut dont nous avons exclu l'implémentation en [#5.2].

M.B. Il faut noter que l'utilisation de ces nouvelles facettes n'a de sens qu'en phase de création d'univers: les facettes sont alors "traduites" par leurs méthodes `init-facet` et `arer-facet`. Ce sont des `meta-objets` [#12.4]. Les 3 premières sont exclusives, mais les 3 dernières peuvent coexister l'ordre d'héritage étant alors celui de leur apparition dans le slot.

15.1.1. La facette LIKE.

Pour les nostalgiques de COBOL: elle FCLAMPe le slot de l'objet en cours de définition sur le couple slot-frame de la valeur de la facette `like`. La syntaxe de `LIKE` est `(like [slot] [. objet])`, les défauts des 2 paramètres étant respectivement le slot et l'objet courant. Les 2 couples slot-objets sont donc identiques, y compris dans leurs modifications ultérieures. IL faut noter que cette identification se fait sur les objets en cause et non sur leurs futures instances. Pour ce dernier usage, voir l'exemple des axes en [#17.3.1.1]. La présence de `like` exclut toute autre facette.

```
(facette like
  (init-facet (methode . meth-init-like)))
```

15.1.2. Les facettes INIT et INIT-EVAL.

Pour résoudre le problème de l'écriture avec héritage: elles recopient la valeur à la création de l'instance (par la méthode `CREATION`, voir plus loin), sans évaluation pour `INIT` et avec évaluation pour `INIT-EVAL`. Dans ce dernier cas, la forme à évaluer est le `CAR` de la valeur. L'implémentation se fait par réflexe `si-ajout` sur le lien `instancie-par` (voir la méthode `CREATION`) du modèle en cours de création.

```
(facette init
  (init-facet (like . value))
  (creer-facet (methode . meth-facet-init)))

(facette init-eval
  (creer-facet (methode . meth-facet-init-eval)))
```

15.1.3. Les facettes HERIT-FROM et HERIT-BY.

La facette herit-from permet d'élargir l'héritage pour un attribut donné, en citant le (ou les) objets dont l'objet peut hériter, pour cet attribut à uniquement. Herit-by a un rôle similaire, mais indique non pas un objet dont on doit hériter, mais le(s) lien(s) qui permet(tent) d'obtenir la valeur de cet (ces) objet(s).

```
(facette herit-from
  (creer-facet (methode . meth-facet-herit-from)))

(facette herit-by
  (creer-facet (methode . meth-facet-herit-by)))
```

Elles se traduisent, toutes deux, par un réflexe si-besoin pour le couple objet-slot.

15.1.4. La facette VALUE-OF.

La facette value-of permet d'élargir l'héritage pour un attribut donné, à la valeur d'un autre attribut du même objet.

```
(facette value-of
  (init-facet (methode . meth-init-value-of))
  (creer-facet (methode . meth-facet-value-of)))
```

Comme pour les facettes précédentes, elle se traduit par un réflexe si-besoin dans l'objet et le slot.

N.B. Les 3 facettes précédentes sont à valeurs multiples, elles peuvent être présentes ensemble, chacune plus d'une fois et dans un ordre quelconque. Les réflexes si-besoin résultants sont dans le même ordre, mais tous après les réflexes si-besoin explicites de l'attribut.

15.2. Commentaires.

Il est possible d'attacher à un objet ou à un slot des commentaires avec le slot / facette bla-bla.

```
(dual-ideal bla-bla
  (init-slot (methode . meth-nil))
  (init-facet (methode . meth-nil))
  (creer-slot (methode . meth-nil))
  (creer-facet (methode . meth-nil)))
```

Comme le montre la figure 20, bla-bla est le seul slot à échapper à la structure frame-slot-facette. Il n'est évidemment plus accessible comme un slot normal. En tant que slot ou facette, il peut être suivi par n'importe quoi.

```
(bla-bla
  (bla-bla "pour m'auto-commenter"
    moi-même (personnellement))
  (est-un (valeur dual-ideal)
    (bla-bla "un objet sans est-un"
      est-un (est-un ?)
      jour sans soleil)))
```

Fig. 20

N.B. Ne pas oublier cependant, que de tels commentaires sont lus et restent en mémoire, ainsi que tous leurs symboles et chaînes de caractères, et contrairement aux commentaires généraux de LISP (par ";"). Mais l'utilisateur peut toujours modifier le comportement de BLA-BLA, pour que sont contenu soit effacé à la lecture: 2 modes différents sont possibles, à volonté.

15.3. Constantes.

C'est un objet un peu spécial, permettant d'associer une valeur d'objet à une variable globale. Ce sont les seuls objets dont les symboles ne sont pas autovalués. Les constantes sont définies, par la fonction F-PARAM.

```
(dual-ideal constante
  (univ-stat-std (valeur . constantes))
  (multiplicite (methode . meth-nil)))
```

Les constantes sont des objets motivés par l'éditeur de règles: comme il ne connaît que des objets, il était nécessaire de définir ainsi tous les paramètres des applications.

15.4. Création et destruction d'instances.

15.4.1. Méthode création.

C'est une méthode de création d'instances, avec foliage des liens est-un de toutes les instances de même hiérarchie (gain de place et de calcul de la hiérarchie), et initialisation de certains slots.

Deux relations sont associées à cette méthode: instance-de et instancier-par, resp. sous-liens de est-un et de instanciation.

Ces relations permettent de différencier les classes et instances [63.1].

```

(objet-ideal
  (instance-de (lien-inverse . instancier-par)
    (sur-lien est-un)
    (si-ajout ((demon+des-liens)))
    (si-enleve ((demon-des-liens))))
  (instancier-par
    (type-de-recherche . "-a-inv")
    (si-ajout ((demon+des-liens)))
    (si-enleve ((demon-des-liens))))
  (creation (methode . meth-creation))
  (efface (methode . meth-efface)
    (comport-error . meth-nil)))

(dual-ideal (efface (methode . meth-efface-dual)))

(attribut1 (efface-partie (methode (u) (efface (fget u frame+))))))

(attribut2 (efface-partie
  (methode (u) (fmapc efface (fget u frame+))))))

(attribut (efface-partie (methode . meth-nil)))

(univers (efface (methode . meth-efface-univ)))

```

La syntaxe de CREATION est:

(creation modèle nouvel-objet initialisation . autres-modèles), dans lequel MODELE est l'objet à instancier, NOUVEL-OBJET le nom de l'objet à créer, AUTRES-MODELES ses autres liens est-un et INITIALISATION une liste d'initialisation dont les éléments sont, soit des listes slot-valeur, soit des atomes. La fonction F-CARAC est appliquée à NOUVEL-OBJET et INITIALISATION ce qui a pour effet d'appliquer la méthode caract aux slots de la liste, avec pour argument la valeur (premier cas), ou aux atomes, sans argument (deuxième cas). Dans le premier cas, il y a généralement affectation de la valeur, dans la facette standard du slot de NOUVEL-OBJET. Dans le deuxième cas, à moins d'élargir la définition de caract, il faut que l'atome soit un booléen ou une valeur (voir plus loin ces deux extensions).

15.4.2. Méthode efface.

C'est une méthode qui détruit un objet, ainsi que toutes ses instanciations (récursivement), en effaçant le lien est-un des objets détruits pour déclencher tous les réflexes associés. Efface fait appel à

efface-partie pour effacer toutes ses "parties" qui sont référencées par les arguments passés à **EFFACE**.

C'est la seule méthode propre pour détruire un objet, en laissant le graphe d'héritage et d'instanciation dans un état correct.

15.5. Les autoloads.

Ce sont les slots ou facettes dont la définition consiste à décrire un comportement d'autoload. Les méthodes **init-slot** et **init-facet**, ainsi que tous les applicateurs de ces objets ont pour effet de charger le (ou les) fichier de l'attribut **in-file**, recherchés dans la directory **in-lfb** de l'univers du slot-autoload, et de redéclencher le message.

```
(dual-ideal slot-autoload
  (in-file (init-facet . meth-init-in-file)
           (creer-slot . meth-slot-in-file)
           (si-enleve demon-ient-autoload))
  (creer-slot (methode . meth-nil))
  (creer-facet (methode . meth-nil))
  (init-slot (methode . meth-autoload))
  (init-facet (methode . meth-autoload))
  (:= (applic . applic-autoload))
  (+ (applic . applic-autoload))
  (- (applic . applic-autoload))
  (:? (applic . applic-autoload))
  (< (applic . applic-autoload))
  (> (applic . applic-autoload))
  (: (applic . applic-autoload)))
```

15.5.1. 2 catégories d'autoloads.

15.5.1.1. Slot autoload.

Ce sont soit des instances explicites de **slot-autoload**, soit des slots (clé de 1-er niveau) qui possèdent la facette non standard **in-file** (clé de 2-ième niveau: méthode **init-facet**). Le chargement est provoqué par l'appel d'un applicateur ou des méthodes de création d'univers sur le slot (et donc par sa présence en clé de 1-er ou 2-ième niveau en création d'univers).

15.5.1.2. Objet autoload.

La présence de **in-file** en clé de 1-er niveau plante dans l'objet un réflexe **si-ajout** sur son lien instanciation, dont l'activation déclenche l'autoload. C'est donc l'instanciation de l'objet qui provoque le chargement.

L'utilisation de tout autre accès à un **slot-autoload** (directement par une primitive) a des résultats imprévisibles.

15.5.2. Fichiers d'autoload.

Toutes les extensions en autoload sont des créations d'univers en insertion, et utilisent **INSERT-UNIVERS** [#12.4]. Ces insertions se font

dans l'univers racine objet-ideal, pour celles qui sont présentées ici, mais il est bien sûr possible de les faire dans n'importe quel univers.

Il y a 3 catégories de fichiers à charger pour un autoloading: les fichiers de suffixe ".kb" définissent les objets (INSERT-UNIVERS), ceux de suffixe ".univ" les méthodes de création d'univers. Enfin un fichier de suffixe ".ll" (version interprétée) ou ".cp" (compilée) contient les utilitaires de l'extension. Tous ces fichiers sont dans la directory in-lib de l'univers de définition du slot-autoloading.

Les fichiers présents parmi ces 3, sont chargés dans l'ordre: ".univ", ".ll" (ou ".cp") puis ".kb"; l'absence de l'un d'entre eux est acceptée, mais un message d'erreur signale l'absence de tous (1).

15.5.3. Remise en autoloading.

Le fichier de suffixe ".univ" peut être déchargé (REMFN) à la fin de la création avec les autres fonctions de création d'univers, et rechargé au premier besoin, à condition de rajouter l'une quelconque des fonctions de ce fichier dans la variable globale #:YAF00L:FN-FILES: voir cette variable [#19.4.1] et la fonction REMOB-FILE-UNIV [#19.5]. Les slots autoloading pour lesquels existent de telles fonctions sont remis en "autoloading" pour leurs méthodes d'univers par REMOB-FILE-UNIV.

M.B. Toutes les extensions précédentes sont présentes dans le noyau du système. Les suivantes sont toutes en autoloading.

15.6. Boolean.

C'est un cas particulier d'attribut1, utilisé essentiellement à valeur vrai/faux. On lui associe une valeur fautive autre que NIL, pour éviter la confusion signalée [#4.4]: faux*.

```

(univers (univ-stat (boolean)))
(objet-ideal
  (boolean (si-:: demon-bool-ideal)))
(boolean
  (type-de-recherche (si-ajout demon-Z-bool))
  (:= (applic . appl=bool))
  (caract (methode . meth-caract-bool)))

```

Un réflexe si-besoin est implémenté dans objet-ideal pour tout boolean, afin d'affecter cette valeur fautive en cas de recherche infructueuse. Ce réflexe se définit par:

(progn (fput frame* slot* value 'faux*) (—))
la pseudo-continuation servant à désactiver les mécanismes habituels des réflexes si-besoin (d'où aussi le FPUT explicite).

(1) L'existence du fichier est actuellement testée par la fonction Le_Lisp #ROSEFILE qui envoie malheureusement le message "Le_Lisp: infile: no such file or directory", sans permettre de le détourner sur un autre canal. Aucune erreur n'est néanmoins déclenchée.

4/7/85

On a aussi ici un exemple de réflexe d'applicateur. Pour un booléen bool, on a l'expansion:

```

<bool>
  <fget frame+ bool>                retourné par l'applicateur
  <(let ((g238 (fget frame+ bool)))  modifié par le réflexe
    (and (neq 'faux+ g238) g238))
  
```

Le :applicateur := supplée l'absence de valeur par T.

N.B. L'élision de FRAME+ avec := ne marche donc qu'en l'absence de tout argument.

```

(:= 'bool)
(fput+ frame+ bool t)
  
```

La méthode caract d'un booléen n'a pas besoin d'argument de valeur: T est supplée:
 (caract bool frame) a pour effet d'affecter au BOOL de FRAME la valeur T, sous la facette value.

15.7. Domain.

C'est un attribut particulier, qui indique le domaine de la valeur d'un slot. Il a 2 valeurs par défaut, réflexe si-besoin, pour les attributs et les relations.

```

(attribut (domain (si-besoin (frame+))))
(relation (domain (si-besoin ((ici-univers))))))
  
```

Cet attribut est particulièrement utilisé par l'application d'éditeur de règles. Il pourrait aussi servir avec les réflexes a priori (si-possible), s'ils doivent être implémentés.

15.8. Valeur et range.

Range est un attribut qui donne la liste de toutes les valeurs possibles d'un attribut: chacune de ces valeurs est une valeur, de domaine son slot.

```

(univers (univ-stat (valeurs)))
(attribut (range (creer-facet . meth-facet-range)
                (init-facet . meth-init-range)))
(valeur (domain (bla-bla pour autoloader DOMAIN))
        (:= (applic . appl=valeur))
        (:= (applic . appl.valeur))
        (caract (methode . meth-caract-val)))

```

Il est possible d'accéder directement à une valeur, sans citer son slot: si VAL est une valeur déclarée dans le range de SLOT, la figure 21 donne les expansions des accès en lecture/écriture.

```

(val)
(when (eq val (:= (domain val))) val)
(:= val)
(:= (domain val) val)

```

Fig. 21
Valeurs

Le range n'induit aucune vérification particulière de validité d'écriture: celle-ci est immédiate si l'on utilise := comme ci-dessus.

Enfin, la méthode caract des valeurs n'a pas d'argument de valeur: (caract val frame) a pour effet d'affecter au SLOT de FRAME la valeur VAL, sous la facette value.

N.B. Il y a une obligation de consistance totale pour les valeurs: elles ont un domaine unique.

15.9. Trace de slots.

Les méthodes trace-slot et detrace-slot permettent de tracer les attributs, comportements et facettes, et ce sélectivement dans tel ou tel objet.

Leurs arguments représentent la liste des objets pour lesquels (pour les instances desquels) le slot est à tracer.

15.9.1. Trace des attributs.

Elle se fait par implantation de réflexe si-ajout et si-enleve dans les objets à tracer, par défaut objet-ideal. Ne sont donc tracés que les accès déclenchant des réflexes d'écriture.

15.9.2. Trace des facettes.

Elle concerne la trace des attributs: seules sont tracées les écritures sur des facettes tracées.


```

(dual-ideal (trace-frame (est-un attribut2)))

(attribut (trace-slot (methode . meth-trace-attr))
          (detrace-slot (methode . meth-detrace-attr)))

(methode (trace-slot (methode . meth-trace-comport))
         (detrace-slot (methode . meth-detrace-comport)))

(applic (trace-slot (methode . meth-trace-comport))
        (detrace-slot (methode . meth-detrace-comport)))

(facette (trace-flag (est-un boolean)
                    (value . faux*))
         (trace-slot (methode . meth-trace-fac))
         (detrace-slot (methode . meth-detrace-fac)))

(facette-user (trace-flag (value . t)))

(value (trace-flag (value . t)))

(szuf (trace-flag (value . t)))

```

15.9.3. Trace des comportements.

Pour tous les objets à tracer, par défaut la valeur de l'attribut `liste-frame` du slot, et leurs instances (récursivement), les valeurs fonctionnelles du comportement à tracer sont remplacées (FPUSH) par des λ -expressions formées de la même manière que par le package de trace de `Le_Lisp`, avec édition de `FRAME+`, du nom de la méthode, du nom de sa valeur fonctionnelle si c'est un symbole, du nom de l'objet qui détient la valeur fonctionnelle, (car `herit+`) et enfin des arguments.

N.B. La fonction `TRACE` de `Le_Lisp` a été modifiée de façon à ce que les fonctions tracées dont le nom commence par `REFLEX`, `METH` ou `DEMON` fassent une édition des variables du quadruplet d'environnement.

15.10. Edition.

Les méthodes d'édition sont `editer` pour un objet, `editer-slot` pour un slot et `editer-fac` pour une facette. De façon générale, la première appelle la seconde, qui appelle la dernière.

`Editer` prend comme argument optionnel un nom d'univers: la signification est que l'édition ne doit concerner que les slots définis postérieurement à cet univers. Par défaut, c'est l'univers de définition de l'objet à éditer. Si c'est `t`, c'est l'édition de tous les slots.

`Editer-slot` (resp. `editer-fac`) prennent pour argument la paire du slot (resp. de la facette) dans la λ -liste de l'objet (resp. du slot).

```

(objet-ideal (editer (methode . meth-editer))
             (editer-slot (methode . meth-editer-slot))
             (editer-fac (methode . meth-nil)
                          (comport-error . meth-nil)))

(reflexe (editer-slot (methode . meth-editer-reflex)))
(attribut (editer-slot (methode . meth-editer-facet)))
(methode (editer-slot (methode . meth-editer-comport)))
(applic (editer-slot (methode . meth-editer-comport)))
(facette (editer-fac (methode . meth-nil)))
(facette-user (editer-fac (methode . meth-edt-facet)))
(value (editer-fac (methode . meth-edt-value)))
(sauf (editer-fac (methode . meth-edt-sauf)))
(ble-bla (editer-fac (methode . meth-edt-facet)
                    (editer-slot (methode . meth-edt-facet)))

```

? ^West-un

```

EST-UN : est-un = relation2
         type-de-recherche =
         liste-frame = objet-ideal dual-ideal slot-autoload reflexe
                       si-besoin attribut attribut1 attribut2
                       relation relation1 relation2 methode applic
                       facette facette-user value herit like sauf
                       init caract init-slot init-facet bla-bla
                       constante univers valeur booleen trace-slot
                       detrace-slot compose-de temporel range pjea
                       domain editer
         lien-inverse = instantiation
         liste-slot = si-ajout si-enleve type-de-recherche herit :=
                     :+ :- :< lien-inverse creer-slot init-facet

* si-ajout objet-ideal (())
                      (when ...)
                      univers demon-sauf-univers
* si-enleve objet-ideal (())
                      (when ...)

* creer-slot meth-slot+value
* init-facet meth-init-isa
* := appl=isa
* :- appl-isa
* :+ appl+isa

= est-un

```

Fig. 22

est-un

Comme le montre la figure 22, l'édition standard est déparenthésée, tout au moins jusqu'au niveau des valeurs, et l'édition des attributs a lieu avant celle des réflexes et comportements (qui sont retardées). Enfin ne sont éditées que les facettes pour lesquelles editor-fac n'est pas METH-NIL.

La macro-caractère ^M est une abréviation pour éditer avec T comme argument. Rappelons ici que le macro-caractère ^V fait appel à FPRETTY et fait donc un pretty-print de l'objet.

16. UNIVERS TEMPORELS.

Il s'agit de représenter des objets dont l'état évolue dans le temps, que ce soit un temps "réel", linéaire et continu, ou un temps métaphorique des changements d'état des objets. Cette évolution doit être discrète, en ce sens que les objets ne doivent pas changer d'état à tout instant: en cas de continuité, il faut donc discrétiser le temps, par utilisation de "quanta de temps".

16.1. Objets temporels et atemporels.

La représentation de tels processus temporels nécessite donc, pour chaque objet concerné par le temps, la définition de 2 objets:

- l'un, atemporel (1), représente les propriétés immuables (au moins pendant un certain laps de temps) de l'objet.
- l'autre, temporel, représente son état à un instant t donné.

Le modèle temporel peut représenter l'état de l'objet, soit à chaque changement d'état, soit à chaque quantum de temps.

16.2. Univers temporel.

L'extension temporelle définit deux modèles, les objet-temporels et objet-atemporels, qui doivent être dans la hiérarchie de tout objet temporel ou atemporel, ...

```

(objet-ideal objet-atemporel
  (instamp (si-ajout demon-inst+temporel)
            (si-enleve demon-inst-temporel)
            (editer-slot . meth-nil))
  (temporel (si-enleve demon-temporel)
            (type-de-recherche . "")
            (lien-inverse . atemporel)
            (creer-slot . meth-slot+value)))

(objet-ideal objet-temporel
  (atemporel (si-ajout ((demon+inverse1)))
            (type-de-recherche . "")
            (creer-slot . meth-slot+value)))

```

... 3 liens, temporel, atemporel et instamp ... et une nouvelle catégorie d'attribut, booléen-t.

16.2.1. Liens et instances temporels.

Dans les termes du paradigme modèle/instances [#12.1], on a donc:

- 1 modèle atemporel,
- 1 modèle temporel,

(1) On appellera atemporel un objet pour lequel le temps n'a pas de sens.

- des instances atemporelles et
- pour chacune d'elles, une infinité virtuelle d'instances temporelles.

16.2.2. Liens temporels.

Deux liens inverses à valeur unique sont destinés à la représentation de cette dualité: temporel et atemporel.

Pour les modèles ces liens sont sans ambiguïté: chacun pointe sur l'autre.

Pour les instances, le lien atemporel ne pose toujours pas de problème; par contre le lien temporel est défini comme pointant sur la dernière instance de l'objet: il s'agit donc de son état courant.

16.2.2.1. Liens (a)temporels et est-un.

Enfin, ces liens temporels ne sont pas sans ambiguïté vis-à-vis du lien est-un. En effet, on peut considérer que tout objet "temporel" est son objet "atemporel", puisque ce dernier contient ses propriétés immuables. Inversement, tout objet "atemporel" est son objet "temporel", puisque ce dernier définit son état courant.

On trouve alors un paradoxe: si l'on définit un objet "temporel" comme un objet ayant un lien atemporel, et inversement, un objet "temporel" est "atemporel" puisqu'il hérite des propriétés de son objet "atemporel" qui, lui, a un lien temporel.

N.B. Dans notre implémentation, nous avons choisi que l'objet temporel hérite de l'objet atemporel, par inclusion du lien atemporel dans le lien est-un. Les propriétés de l'instance temporelle ont donc la priorité sur celles de l'objet atemporel: on peut dire que ce dernier est un objet par défaut. Cette inclusion est automatique pour les instances (créées par la méthode création-t) mais pas pour les modèles pour lesquels cette inclusion doit être explicite. Enfin, tout modèle ou instance (créée par création-t) temporelle contient objet-atemporel comme exception à l'héritage.

16.2.3. Modes d'accès particuliers en lecture.

La recherche de la propriété d'un objet (a)temporel peut se faire dans l'un ou l'autre des objets temporels ou atemporels.

Les modes d'accès [#5.1] en T (resp. AT) permettant de consulter les 2 objets, dans l'ordre temporel puis atemporel, pour un objet temporel (resp. atemporel).

N.B. Pour des raisons évidentes [#6.3.2], il n'y a pas d'équivalent de ces modes de lecture pour l'écriture.

16.2.4. Instances temporelles.

L'attribut instamp est utilisé pour lier à une instance atemporelle l'ensemble de ses instances temporelles.

L'instance temporelle y figure comme valeur, sous la facette de son quantum de temps (2).

(2) On a donc ici une exception à la règle comme quoi toute facette est un objet. Il peut aussi y avoir des problèmes d'implémentation dans d'autres dialectes puisque l'accès se fait par EQ.

La fonction **INSTANCE** permet de retrouver, avec 2 arguments l'instance temporelle d'un objet à un instant donné, et avec 1 seul la dernière instance: c'est alors un équivalent de **TEMPOREL** (en tant qu'appel du applicateur ::).

La fonctions **INSTANCES** permet de retrouver la dernière instance temporelle à un moment donné: **INSTANCE** fait un test d'égalité (=), **INSTANCES** d'infériorité (<=). **INSTANCE** et **INSTANCES** retournent la même valeur si et seulement si **INSTANCE** ne retourne pas NIL.

16.3. Booléens temporels.

Les attributs booléens des objets atemporels présentent une caractéristique intéressante: ils peuvent donner une information non plus strictement booléenne, vrai / faux, mais en plus indiquer quand le booléen est devenu vrai.

16.3.1. Booléen-t.

```

(boolean boolean-t
  (univ-stat-std (value . boolean-ts))
  (domain (value . qt))
  (:= (applic . appl=boot))
  (:- (applic . appl=boot)))

(univers (univ-stat (boolean-ts)))

(objet-atemporel
  (liste-bool (type-de-recherche . -A)))

(objet-temporel
  (liste-bool))

```

Ces booléens particuliers sont regroupés dans la sous-catégorie **boolean-t**.

16.3.2. Accès particulier.

L'un des problèmes posés par la gestion du temps est celle de la conservation d'un historique. Pour les objets temporels, il suffit de conserver les instances temporelles "intéressantes" (voir la méthode **nett-histoire**). Pour les objets atemporels, qui ne sont pas censés changer, il peut néanmoins y avoir une évolution. Les **boolean-t** disposent de modes d'accès particuliers qui empiètent les nouvelles valeurs, en cas d'affectation ou d'effacement. Ce mode d'accès est réalisé par les primitives **FPUT-P** et **FREM-P**.

FPUT-P fait les mêmes tests d'écriture que **FPUT**, mais si l'écriture doit se faire, elle empile une nouvelle facette comme **FPUT** (3).

(3) **FPUT** et **FPUT-P** diffèrent uniquement par l'absence de tests d'écriture pour la première: avec **FPUT-P** il ne peut pas y avoir 2 facettes successives avec la même valeur. De même **FREM-P** diffère de **FREM** (avec la valeur NIL) par la présence de tests.

FREM-P fait les mêmes tests d'effacement que FREM, mais si l'effacement se fait, elle empile une nouvelle facette, sans valeur.

Cas 2 primitives existent aussi avec déclenchement de réflexes (FPUT+-P et FREM+-P).

FOECK-P permet une vérification dans l'une quelconque des facettes empilées.

16.3.3. Liste-bool.

Pour chaque objet atemporel, l'attribut liste-bool mémorise tous les booléens: il est ainsi possible de gérer des retours en arrière (voir plus loin la méthode RETOUR-T) de façon très générale.

16.4. Applications aux :applicateurs.

Leur but étant de simplifier les accès, il s'est agi de les automatiser partiellement en ce qui concerne les problèmes temporels.

```
(objet-atemporel
  (boolean-t (si-:= demon-bool=atemp))
  (attribut (si-:= demon-attribut.atemp)
            (si-:= demon-attribut=atemp)
            (si-:= demon-attribut=atemp)
            (si-:= demon-attribut=atemp)))

(objet-temporel
  (boolean-t (si-:= demon-bool=temp))
  (attribut (si-:= demon-attribut=temp)
            (si-:= demon-attribut=temp)
            (si-:= demon-attribut=temp)
            (si-:= demon-attribut=temp)))
```

16.4.1. :applicateurs sur objets temporels.

Pour décrire brièvement le problème, 4 types d'accès sont possibles:

- accès à un objet temporel, à partir de l'objet atemporel et du temps,
- accès à une propriété temporelle (4) à partir de l'objet atemporel,
- accès à une propriété atemporelle, à partir de l'objet temporel,
- enfin, l'accès à une propriété (a)temporelle, à partir de l'objet (a)temporel.

16.4.1.1. Lecture.

On veut donc que pour la lecture, si A est une propriété d'un objet X ou de son objet (a)temporel:

- (A X T) s'expande en (fget (instance X T) A),
- et (A X) en (fget (fget X temporel) A),
- ou (fget (fget X atemporel) A),
- ou (fget X A), suivant les cas (5).

(4) Par généralisation (et abus), on peut étendre aux alets les notions de temporalité, par extension des objets où ils sont définis: un alet peut donc être à la fois temporel et atemporel.

Dans tous ces cas, l'élimination du FRAME* [13.4.5] s'applique avant l'expansion "temporelle".

Ces différentes expansions sont réalisées par les différents réflexes des applicateurs, définis dans objet-temporel et objet-atemporel.

16.4.1.2. Écriture.

Dans le cas de l'écriture, il n'y a d'indirection temporelle sur l'objet que si le slot est uniquement (a)temporel, et contrairement à l'objet.

16.4.2. :applicateur d'écriture booléen.

Dans ce cas, la valeur par défaut, T est remplacée par la valeur du quantum de temps courant TEMPS, la macro

(:= X A) s'expande en (:= X A temps), où TEMPS est l'horloge du système.

Les :applicateurs d'écriture utilisent les primitives FPUT+P et FREN+P définies au paragraphe précédent.

16.4.3. Programmation par métonymie.

On peut envisager un développement intéressant mais aventureux de ce type de programmation: ce que l'on pourrait appeler programmation par métonymie (6).

Dans le cadre d'un réseau sémantique, l'accès à l'attribut X d'un objet A, se ferait alors sur l'objet, "le plus proche" de A dans le réseau, qui peut effectivement posséder l'attribut X.

Ainsi, "boire un verre" se traduirait par "boire le contenu du verre", "contenu" étant un lien pointant sur un objet "buvable" !

Il s'agirait donc, dans un premier temps, de généraliser à tous les liens, ce qui a été fait pour les liens temporels. A terme, cette méthode peut être utilisée récursivement, en suivant dans le réseau un chemin et non pas un arc.

Se poserait évidemment de nombreuses questions en cas de chemins multiples.

16.5. Comportements temporels.

16.5.1. Création.

16.5.1.1. Création d'instances temporelles.

CREATION-T crée de nouvelles instances temporelles d'un objet atemporel, avec FCLAMPage des liens est-un de toutes les instances temporelles d'un même objet.

(5) L'expansion tient bien sûr compte du TYPE-DE-RECHERCHE du slot, en utilisant éventuellement les modes en T ou AT, en cas d'absence.

(6) La métonymie est une figure de rhétorique consistant à nommer la partie pour le tout, le contenant pour le contenu ou un objet pour son voisin. La figure duale de la métonymie est la métaphore: dans la première, la relation de dénotation est syntagmatique (en présence), dans la seconde, elle est paradigmatique (en absence). La programmation "métaphorique" semble totalement hors de question ([Jakobson]).


```

(objet-atemporel
  (nett-histoire (methode . meth-nett-histoire))
  (creation (methode . meth-creation-atemp))
  (creation-t (methode . meth-creation-t))
  (retour-t (methode . meth-retour-t)))

(objet-temporel
  (efface-t (methode . meth-efface-t)))

(boolean
  (retour-t (methode . meth-retour-slot)))

```

Sa syntaxe (`creation-t` `objet-atemp` `instant` `initialisation`) présente un seul point commun avec celle de création: la liste d'INITIALISATION. Le nom de l'instance est formé par concaténation de l'OBJET-ATEMP et de l'INSTANT. L'héritage de l'instance est formé de OBJET-ATEMP et de l'ensemble des liens temporels des propres modèles de celui-ci.

16.5.1.2. Création d'instance atemporelles.

La méthode CREATION provoque pour un objet atemporel, la création d'une première instance temporelle. Tout objet atemporel est ainsi "initialisé" par un objet temporel.

N.B. La création de cette première instance temporelle ne se fait pas par CREATION-T, cette dernière présupposant toujours son existence.

16.5.1.3. Effacement d'instance temporelles.

Ils se font par la méthode efface-t.

16.5.2. Retour en arrière.

RETOUR-T détruit (par EFFACE-T) toutes les instances temporelles d'un objet atemporel, postérieures à son argument. Tous les booléens de l'objet atemporel (dans son attribut liste-bool), dont la valeur est elle-aussi postérieure sont effacés.

Elle existe aussi dans la catégorie duale boolean: son rôle est alors d'effacer, s'il y a lieu, les booléens dont la valeur est supérieure à son argument. La méthode duale est appliquée sur tous les éléments de liste-bool par la méthode de objet-atemporel.

16.5.3. Nettoyage d'historique.

La méthode nett-histoire permet de nettoyer le passé d'un objet des instances qui datent de plus que la valeur de durée-de-conservation, et qui ne présente aucune modification de leurs attributs booléens-t par rapport aux instances précédentes et suivantes.

16.6. Constantes temporelles.

Temps est l'horloge en quantum de temps (QT) du système, échelle-de-temps la valeur d'un quantum de temps, et durée-de-conservation la valeur de la durée minimale (en QT) de conservation d'une instance temporelle.

```
(f-param (temps (domain . qt)
              (value . -1))
         (echelle-de-temps (value . 1.))
         (duree-de-conservation (value . 2)
                                (domain . qt)))
```

16.7. Qt et durée.

Ces 2 objets définissent des objets abstraits utilisés essentiellement comme cible de l'attribut `domain`, pour l'éditeur.

Sous certaines conditions, dire qu'un slot est de `domain qt` en fait un `domain-t`.

```
(objet-ideal qt-et-duree)
(qt-et-duree duree)
(qt-et-duree qt)
```

16.8. Fonctions temporelles.

Un certain nombre de fonctions spécifiques concernant les objets temporels et atemporels.

16.8.1. Fonctions de manipulation d'instances temporelles.

Outre `INSTANCE` et `INSTANCES`, `CREER-INSTANCE`, `DEL-INSTANCE` et `PREC-INSTANCE` permettent de retrouver une instance temporelle, ou la précédente, d'en créer ou d'en détruire.

16.8.2. Recherche des instances (a)temporelles d'un objet.

`TEMPORAUX` et `ATEMPORAUX` permettent de retrouver toutes (7) les vraies instances (8) (a)temporelles d'un objet. `ATEMPORAUX` retourne l'intersection des `INSTANCIE-PAR` (tel que le définit son `:applicateur` ::) de son argument et de `OBJET-ATEMPOREL`: (`instancie-par objet-atemporel`), (`atemporeaux objet-atemporel`) et (`atemporeaux objet-ideal`) sont donc équivalents. `TEMPORAUX` se définit à partir de `ATEMPORAUX`.

(7) c'est-à-dire récursivement.

(8) C'est-à-dire les instances qui ne sont pas des classes: ce type de fonction est, semble-t-il, impossible pour les objets intemporels dans la mesure où la distinction classe / instance devient caduque (chapitre [5.4.1]). Par contre une instance atemporelle se définit comme un représentant d'un objet atemporel qui a des instances temporelles: une classe atemporelle a un lien temporel mais pas d'instance temporelle. Les relations `<instance-dest` (`instancie-par` [5.4.1]) infirment cette note.

16.8.3. Fonctions d'historique.

Ces fonctions permettent une édition expurgée de l'"histoire" d'un objet atemporel et des interrogations sur celle-ci: ce sont HISTOIRE, et CERCHE-QUAND, QUAND, QUAND-NON, QUAND-CHANGE ou QUAND-.

Enfin la fonction EDITION-INSTANT permet d'éditer l'état d'un univers à un instant donné.

N.B. Toutes ces fonctions sont définis au chapitre [19.6].

17. L'EXTENSION VIDEO.

Cette extension définit des objets (a)temporels écrans, ainsi qu'un certain nombre de primitives, slots et facettes permettant de gérer dans ces écrans des champs et des points. Ces derniers s'affichent dans des repères qui possèdent des axes.

N.B. Il s'agit d'écrans alphanumériques, analogues donc au terminal virtuel Le_Lisp. Le cas des bitmap et du multi-fenêtrage est à l'étude.

La plupart des symboles de cette extension étant dans le package VIDEO, nous le sous-entendrons avec la convention, habituelle en Le_Lisp, du ":", ainsi qu'un indicateur d'état :videoop.

17.1. Les écrans.

#:video:ideal est un objet atemporel qui possède des attributs de taille (nombre de caractères) :xmax et :ymax (comptés à partir de 0), un canal de sortie :outchan et des méthodes de prologue et épilogue, ainsi qu'un indicateur d'état :videoop.

```

(objet-ideal #:video:ideal
  (temporel (value . #:video:ideal-t))
  (:tty (value . #. #:tty:name))
  (:xmax (value . #. #:tty:xmax))
  (:ymax (value . #. #:tty:ymax))
  (:videoop (est-un boolean))
  (:outchan)
  (:epilog (methode . meth-video-epilog))
  (:prolog (methode . meth-video-prolog))
  (:histo (methode . meth-histo-video))
  (:creation (methode . meth-creation-video))
  (:creation-t (methode . meth-creation-video-t))
  (:net-histoire (methode . meth-nil)))

(#:video:ideal #:video:ideal-t
  (atemporel (value . #:video:ideal)))

(objet-ideal (:courante (value . #:yafool:video))
  (:repere-courant (value . #:yafool:repere)))

```

17.1.1. Les coordonnées.

Abscisses et ordonnées s'entendent comme pour le terminal virtuel Le_Lisp, comptées à partir de 0 et désignant respectivement, l'indice du caractère dans la ligne, de gauche à droite, et l'indice de la ligne dans la page, de haut en bas.

17.1.2. Ecriture.

Toute "écriture" sur cet écran se fait dans son objet temporel #:video:ideal-t, par l'intermédiaire de champs et de points. Si l'indicateur :videoop est positionné, cette écriture se fait aussi sur le canal :outchan de l'écran.

17.1.3. Prologue et épilogue.

L'indicateur précédent est positionné entre :prelog et :epilog, qui correspondent respectivement aux TYPROLOGUE et TYEPILOGUE Le_Lisp.

17.1.3.1. Prologue.

L'action du :prelog provoque l'appel de la méthode :histo qui envoie sur le canal :outchan toutes les écritures (champs et points) faites précédemment, et mémorisées par le slot :traces des instances temporelles. Il fait auparavant appel à TYPROLOGUE.

17.1.3.2. Epilogue.

Outre un appel de TYEPILOGUE, il positionne le curseur en bas de l'écran. En mode video, toute erreur (SYSERROR) fait appel à :epilog, sur tous les écrans actifs.

17.1.4. Ecran par défaut.

L'attribut :courante désigne l'écran par défaut (d'un objet, d'un univers etc..). Par défaut, c'est #:YAFOOL:VIDEO, instance de #:VIDEO:IDEAL créée au chargement de la video. Ce dernier a lieu sur autoload des attribut et facette #:videopoint et #:videochamp.

17.2. Les champs.

Un champ est la définition, pour un couple objet-attribut, des coordonnées physiques d'édition, dans l'écran :courante de l'objet, de la valeur de ce couple à tout ses changements.

Un champ se définit par la présence, en phase de création d'univers, de la facette #:videochamp (autoload) dans le couple visé, ce qui provoque l'implantation des réflexes d'écriture ad hoc: il s'agit donc d'une trace (au sens de trace-slot pour les attributs) plus sophistiquée.

```
(facette #:videochamp
  (creer-facet (methode . meth-champ-video)))

(objet-ideal
  (:x (creer-slot . meth-slot+value)
    (si-ajout
      ((when (:y)
          (video-champ (:courante)
                       frame+ value+ (:y))))))
  (:y (creer-slot . meth-slot+value)
    (si-ajout
      ((when (:x)
          (video-champ (:courante)
                       frame+ (:x) value+))))))

(#:video:ideal (:temps (value "temps " 1 -2)))

(#:video:ideal-t (:traces))
```

La primitive d'affichage d'un champ est VIDEO-CHAMP.

17.2.1. Syntaxe de #:VIDEO:CHAMP.

La valeur du triplet objet-attribut-#:video:champ doit être constitué de 3 ou 4 éléments ainsi définis:
(#:video:champ texte absc ord [long]).

- une chaîne de caractère, éventuellement vide (""), désignant un titre: c'est une constante, qui sera affichée une seule fois, au moment de l'analyse en cours du champ, et qui restera visible en permanence, sauf en cas de réécriture au même endroit..

- 2 valeurs entières, indiquant les coordonnées de l'édition du champ dans l'écran. Les valeurs négatives sont comptées à partir de la fin (de la ligne ou de la page). Les valeurs NIL sont remplacées par la valeur correspondante de l'attribut :x (resp. :y) de l'objet dont on veut tracer l'attribut. Ce remplacement a lieu à l'exécution, et à chaque appel.

- une quatrième valeur, entière elle aussi, donne la taille (horizontale) du champ dans lequel on veut écrire la valeur. Par défaut, c'est la longueur de la chaîne titre, et l'édition se fait sur la ligne suivante, en-dessous du titre. En cas de présence explicite, l'édition se fait sur la même ligne que le titre, après lui.

L'écriture de toute valeur est centrée dans le champ physique ainsi défini.

N.B. Il n'y a aucune vérification de conformité des valeurs des positions d'un champ lors de son écriture: il peut donc être écrit en-dehors de l'écran, pourvu qu'il soit bien toujours dans le terminal virtuel (1)!

17.2.2. Coordonnées par défaut.

Sont associés aux champs les deux attributs de coordonnées :x et :y permettant de définir par défaut une position (ligne ou colonne) d'édition de la trace d'un objet. La présence simultanée de ces deux attributs, même par héritage de l'un des 2, provoque l'écriture dans le champ ainsi défini du nom de l'objet.

17.2.3. Attributs complémentaires.

La mémorisation de tous les champs écrits se fait par l'attribut :traces de #:videocr:idesat-t.

Enfin, :stamp est un attribut qui désigne le champ d'écriture de l'horloge. Sa syntaxe est identique à celle de :champ.

17.3. Les points.

La notion de point permet de tracer le déplacement d'un objet dans un repère à 2 dimensions. L'attribut (autoload) #:videopoint désigne, en phase de création d'univers, les 2 attributs, à valeur numérique, représentant les "abscisses" et "ordonnées" de l'objet.

(1) Par contre, une écriture en dehors du terminal peut provoquer des erreurs imprédictibles. Dans le même ordre d'idée, toute utilisation de sortie Le_Lisp standard (PRINT) sur le canal d'un écran en service (<videop>) peut provoquer des erreurs fatales.

17.3.1. Les repères.

Comme les écrans, ils sont définis par des objets (a)temporels: `#:video:repere` et `#:video:repere-t`. Les liens `#:video:repere` et `#:video:ecran` assurent la liaison entre un écran et son (ou ses) repère(s).

```
(#:video:ideal
  (efface (methode u (:> efface #:video:repere)))
  (#:video:repere (lien-inverse . #:video:ecran)
    (si-ajout ((demon+inverse21))))))

(objet-ideal
  #:video:repere
  (temporel (value . #:video:repere-t))
  (#:video:ecran)
  (:histo (methode . meth-histo-repere))
  (creation-t (methode . meth-creation-repere-t))
  (:echellex
    (si-besoin ((divide (:long-x)
      (- (:max-x) (:min-x))))))
  (:echelley
    (si-besoin ((divide (:long-y)
      (- (:min-y) (:max-y))))))
  (:trainee (value . #/.))
  (:min-x (si-ajout ((:- :echellex))))
  (:max-x (like :min-x))
  (:long-x (like :min-x))
  (:pos-x)
  (:min-y (si-ajout ((:- :echelley))))
  (:max-y (like :min-y))
  (:long-y (like :min-y))
  (:pos-y)
  (:re-init))

(#:video:repere
  #:video:repere-t
  (atemporel (value . :repere))
  (efface-t (methode ()) (:re-init (atemporel) (:re-init))
    (:> efface-t)))
  (:traces)
  (:re-init))

(objet-ideal
  (#:video:point (creer-slot . meth-point-video))
  (:trainee (herit-by :repere-courant))
  (:car (si-besoin ((chrnth @ frame+))))))
```

Chaque repère est défini par les valeurs minimales (`:min-x` et `:min-y`) et maximales (`:max-x` et `:max-y`) des coordonnées, par les attributs de position dans l'écran du coin supérieur gauche du repère (`:pos-x` et `:pos-y`), et la longueur suivant les 2 axes, en nombre de lignes ou de caractères, (`:long-x` et `:long-y`). Les échelles (`:echellex` et `:echelley`) de représentation suivant les 2 axes sont calculées à partir de ces 3 attributs, et sont maintenues cohérentes par des réflexes `si-ajout` et `si-besoin`.

17.3.1.1. Les axes.

Chaque repère peut posséder un (ou plusieurs) axe(s), reliés à lui par les liens :axes et :reper. Chaque axe possède 2 caractéristiques: :pos, sa position dans le repère (comptée par référence à celui-ci), et :pas qui représente la valeur de la graduation le long de l'axe. Par défaut le repère est centré, les graduations étant d'une unité.

```

(#:video:reper
  (:video:axes (lien-inverse . #:video:reper)
                (sur-lien instanciation)
                (si-ajout ((demon-inverse21))))
  (:prolog (methode ()) (fmapc :prolog (:axes))))

(objet-ideal #:video:axe
  (:video:reper)
  (:pas (value . 1))
  (:pos (value . 0)))

(#:video:axe #:video:axe-x
  (:prolog (methode ()) (video-axe-x (:reper)
                                     (:pas)
                                     (:pos)
                                     (:max-x)
                                     (:min-x)
                                     (:trainee))))

(#:video:axe #:video:axe-y
  (:prolog (methode ()) (video-axe-y (:reper)
                                     (:pas)
                                     (:pos)
                                     (:max-y)
                                     (:min-y)
                                     (:trainee))))

```

Contrairement aux objets précédents, les axes sont intemporels.

Deux grands types d'axes existent: horizontaux (:axe-x) ou verticaux (:axe-y), qui diffèrent par leur méthode :prolog (appelée par celle du repère) d'affichage.

Enfin, il a été prédéfini 4 types d'axes particuliers, pour les bords du repères, dont la position est FCLAMPée sur les bornes de valeurs des coordonnées.

17.3.1.2. Repère courant.

Comme pour les écrans, il existe, pour tout objet, un repère de référence qui est désigné par l'attribut :reper-courant. Par défaut, c'est #:YAFOOL:REPERE.

17.3.2. Trace des déplacements.

La présence de l'attribut #:video:point dans un objet, provoque la modification de sa méthode CREATION avec l'affichage du caractère :car de l'objet (code ascii), par défaut c'est l'initiale du nom de l'objet,


```

(#:video:axe-x
 #:video:axe-haut
 (:reper (si-ajout ((clamp2 frame+ :pos
                    value+ :max-y))))))

(#:video:axe-x
 #:video:axe-bas
 (:reper (si-ajout ((clamp2 frame+ :pos
                    value+ :min-y))))))

(#:video:axe-y
 #:video:axe-droit
 (:reper (si-ajout ((clamp2 frame+ :pos
                    value+ :max-x))))))

(#:video:axe-y
 #:video:axe-gauche
 (:reper (si-ajout ((clamp2 frame+ :pos
                    value+ :min-x))))))

```

dans sa nouvelle position, et l'affichage de :trainee (par défaut un point), dans son ancienne position. Le repère de référence pour cet affichage est le :reper-courant des coordonnées (plus exactement de la première).

Cet affichage n'a lieu effectivement (sur le terminal), que si les coordonnées sont bien comprises dans les fourchettes définies par le repère.

La primitive d'affichage d'un objet (point) est VIDEO-CLIP.

17.3.3. Mémorisation des points affichés.

Tout point affiché est mémorisé dans l'attribut :traees de #:video:reper-t.

Il est possible de réafficher à chaque pas de temps un objet (devenu immobile par exemple): le dernier affichage de cet objet doit alors avoir été fait par la primitive VIDEO-CLIP-REINIT qui mémorise le point affiché dans les attributs :re-init des écrans atemporels et temporels (les 2 pour pouvoir gérer les retours en arrière, comme le montre la méthode EFFACE).

La méthode CREATION-T de #:video:ideal provoque l'affichage de tous les points de :re-init de l'écran atemporel.

17.3.3.1. Changement de repère.

La mémorisation précédente est absolue: il est possible de changer les paramètres du repère en milieu de session, à condition de le faire après un épilogue. Le prologue suivant réaffichera tout les points précédents, dans le nouveau repère. Cette notion n'a évidemment aucun sens pour les champs.

17.4. Activation et désactivation.

La primitive VIDEO-IN active la video en fournissant un canal de sortie standard \$:YAFDOL:FICHIER-TRACE sur le fichier de nom l'argument de VIDEO-IN et de suffixe ".trace". La variable \$:YAFDOL:TRACE prend alors la valeur de ce canal. Un canal \$:YAFDOL:FICHIER-FIGURE est aussi ouvert sur le fichier de suffixe ".fig".

La primitive VIDEO-OUT permet de désactiver ce canal en redonnant à cette dernière variable la valeur NIL.

17.5. Figure.

Il est possible d'éditer une figure, c'est-à-dire l'état courant de l'écran, par appel de la fonction VIDEO-COPIE. Cette édition se fait sur le canal de sortie \$:YAFDOL:FICHIER-FIGURE, qui doit avoir été préalablement ouvert.

18. EXEMPLE: OBJETS COMPOSITES.

Ce chapitre montre un exemple d'implémentation d'extension. Comme l'a montré la description des univers, une extension consiste à définir un sous-univers (modèles, slots et facettes) et les attachements procéduraux associés.

18.1. Objets composites.

Un objet composite est un objet, formé de plusieurs sous-objets (1), mais ayant un comportement d'ensemble.

Les objets composites sont définis par un objet générique, `objet-composite` qui regroupe les propriétés de ces objets, 2 relations, `compose-de` et `partie-de`, inverses l'une de l'autre, et une facette `init-facet`.

Les seuls comportements globaux que nous y ayons définis sont les méthodes création et efface.

```
(insert-univers objet-ideal)

(facette init-partie
 (init-facet (methode . meth-init-ra!))
 (creer-facet (methode . meth-facet-init-partie)))

(objet-ideal objet-composite
 (compose-de (creer-slot . meth-slot-compose)
 (lien-inverse . partie-de)
 (si-ajout ((demon+des-liens21)))
 (si-enleve ((demon-des-liens21)))
 (type-de-recherche . -1))
 (partie-de (si-ajout ((demon+des-liens)))
 (si-enleve ((demon-des-liens))))
 (creation (methode . meth-creation-composite))
 (efface (methode . meth-efface-composite)))
```

Fig. 23

La présence de l'attribut `compose-de` dans un modèle provoque le rajout, dans la hiérarchie du modèle de `objet-composite`. La facette `init-partie` pointe sur l'élément composant l'objet dont l'instance devra initialiser le slot lors de la création d'une instance.

18.2. Exemple.

La figure 24 décrit une voiture munie de 4 roues (qui portent chacune un nom différent), chaque roue ayant un pneu et une jante.

(1) Ce sont les "templates" de LISP (Schwarz 53).

```

(objet-auto voiture
  (compose-de (value roue-avg roue-avd roue-arg roue-ard))
  (ma-ravg (init-partie . roue-avg))
  (ma-ravd (init-partie . roue-avd))
  (ma-rar (init-partie roue-arg roue-ard)))

(objet-auto roue
  (compose-de (value pneu jante))
  (mon-pneu (init-partie . pneu))
  (ma-jante (init-partie . jante)))

(roue roue-av)
(roue roue-ar)
(roue roue-g)
(roue roue-d)

(roue-av roue-avg
  (est-un (value roue-g)))

(roue-av roue-avd
  (est-un (value roue-d)))

(roue-ar roue-arg
  (est-un (value roue-g)))

(roue-ar roue-ard
  (est-un (value roue-d)))

(objet-auto pneu)
(objet-auto jante)

```

Fig. 24

Voiture, roues, pneus et jantes.

Le message (creation voiture 'touf-touf ()) provoque la création d'une voiture avec ses 4 roues, leurs pneus et jantes. Les attributs ma-ravg ... et mon-pneu et ma-jante pointent sur les instances respectives de ces divers éléments.

18.3. Méthodes d'univers.

La présence d'une facette init-partie fait de son slot une relation (méthode init-facet), et se traduit dans le modèle par la pose d'un réflexe si-ajout sur le lien instanciation par lequel, à chaque création d'une instance, la relation est initialisée par la (ou les) dernières instances de la valeur de cette facette (méthode creer-facet) (2).

(2) L'implémentation de la figure 25 présente le cas d'une valeur unique: pour une valeur multiple (cas de MA-RAR), il y a un mapping supplémentaire. La facette INIT est implémentée de façon assez semblable à INIT-PARTIE. Les facettes VALUE-OF, HERIT-FROM et HERIT-BY aussi, mais avec des réflexes SI-BESOIN.

```

(de meth-init-rel (frame slot values)
  (cond ((null values) relation)
        ((consp values) relation2)
        (relation1)))

```

Fig. 25

```

(de meth-facet-init-partie (frame slot values)
  (caract si-ajout
    instantiation
    frame
    `(((:= ,slot value+
           (last-instance
            , (fram frame slot frame+ t))))))

```

```

(de meth-slot-compose (facets)
  (nouvel-ancestre :frame objet-composite)
  (meth-slot+value facets))

```

Enfin la présence d'un attribut `compose-de` dans un modèle en fait une instance (éventuellement indirecte) de `objet-composite`. `METH-SLOT+VALUE` est la méthode générale utilisée pour créer-slot dans le cas où l'on veut déclencher les réflexes si-ajout sur la facette `value` (ou `sauf`) du slot.

18.4. Utilitaires.

La méthode création des objets composite commence par créer les instances de toutes les parties, avant de créer l'objet lui-même et de lui affecter ses parties.

```

(de meth-creation-composite (objet liste)
  (let ((parties (mapcar (lambda (u) (creation u (fgensym u) ()))
                        (compose-de))))
    (:> creation objet liste)
    (:= compose-de objet parties))
  objet)

```

Fig. 26

```

(de meth-efface-composite ()
  (fmapc efface (:= compose-de))
  (:> efface))

```

La méthode `efface` détruit les parties, avant de détruire l'objet.

18.5. Classes, instances, parties et héritage.

Quelques remarques sont nécessaires pour limiter l'utilisation de ces objets composites tels qu'ils sont implémentés ici: c'est une implémentation "jouet" qui ne résiste pas à une utilisation abusive.

On remarquera que le graphe d'héritage des roues est assez complexe, et que

la "composition" des roues s'est faite à travers cet héritage.

Vouloir instancier récursivement `teuf-teuf` ne causerait pas d'erreur mais manque sans doute de signification: toutes les parties de la nouvelle instance seraient des instances des parties de `teuf-teuf`.

Par contre, toutes les instances et toutes les parties d'un objet composite doivent avoir le même statut dans le paradigme classe/instance [§3.1]. Les parties d'un modèle composite ne doivent pas avoir d'instance, au moment de la création de l'univers: la définition de sous-catégories de `pneu`, par ex. `pneu-lisse` et `pneu-clou` aurait produit des résultats aberrants: les réflexes `si-ajout` correspondant à la facette `init-partie` de `roue-ideal` sont en effet activés par la création des sous-catégories de `roue-ideal`. Par contre, toute partie peut être elle-même composite.

Ainsi, les attributs `non-pneu` et `ma-jante` des sous-catégories sont absents (NIL) uniquement parce qu'ils sont à valeur unique (LAST-INSTANCE retourne alors NIL, il n'y a pas d'écriture). Si ces attributs avaient été à valeurs multiples, LAST-INSTANCE aurait retourné une liste de NIL et il y aurait eu écriture.

10.6. Les fichiers de l'autoload.

Les figures 23, 25 et 26 constituent respectivement les fichiers `composite.kb`, `composite.univ` et `composite.ll`. Pour assurer la remise en `autoload` de `compose-de`, il faut ajouter dans le fichier `composite.univ` la forme

```
(NEW #:YAFUOL:FW-FILES 'NETH-SLOT-COMPOSE).
```

Enfin, `compose-de` est défini dans le noyau par:

```
(slot-autoload compose-de
  (in-file (value . composite)))
```

CINQUIEME PARTIE
MANUEL D'UTILISATION

Life is a tale,
full of sound and fury,
said by A FOOL,
and signifying nothing.

[Shakespeare]

19. PROGRAMMER EN YAF00L.

Ce chapitre reprend l'ensemble du langage au niveau de l'utilisateur, autour d'un exemple qui est disponible dans le fichier `astarix.kb`, dont nous allons décrire ici la genèse.

Il nécessite, outre une connaissance minimale de LISP, la lecture de la première partie de ce manuel [1, 2 et 3]. Des renvois seront faits dans le cours du texte, aux autres chapitres [4 et suivants] auxquels le lecteur pourra se reporter pour plus de détails.

19.1. Un univers d'objets.

Le but de YAF00L est de faciliter la description d'objets symboliques capables de s'envoyer des messages. Les objets à décrire sont rassemblés dans un univers [12], dont la création commence par l'appel de la fonction `big-bang` [12.3]: toutes les formes lues sont alors des descriptions d'objets, avec la syntaxe définie en [12.3.1 et 20.4], jusqu'à la première forme atomique rencontrée.

L'univers que nous allons décrire se situe un peu en arrière dans le temps, plus précisément aux alentours de 52 avant Jésus-Christ. Mais laissons le soin au slot `ble-bla` [15.2] de nous raconter la suite, lors de la création de l'univers.

```
(big-bang avant-JC
  (ble-bla toute la gaule est occupee...
    (toute? non car un petit village...
      d'apres goscinnny et uderzo))
```

`avant-JC` est alors la racine de l'univers de même nom, et donc instance des objets univers et de l'univers courant, sans doute le noyau du système. Quant à `ble-bla`, il permet d'introduire des commentaires [15.2].

19.2. Des lieux et des personnages.

Notre univers se compose de 2 types principaux d'objets: personnage et lieu.

```
(avant-JC personnage
  (lieu-actuel (lien-inverse . presents)
    (si-ajout ((demon+des-liens12)))
    (si-enleve ((demon-des-liens12))))
  (berceil))
```

Ce sont alors des instances (ou sous-classes) de l'objet `avant-JC`.


```
(avant-JC lieu
  (presente (si-ajout ((demon+des-liens21)))
    (si-enleve ((demon-des-liens21))))))
```

Ces 2 concepts sont reliés par 3 attributs: lieu-actuel, bercail et présente: tout personnage est quelque part, son lieu-actuel et possède un bercail, son lieu habituel. Tout lieu connaît les personnages qui y sont présents. Ces 3 attributs sont des liens [#3.1.5], c'est à dire que leur valeur est un autre objet de l'univers.

Présente et lieu-actuel sont clairement inverses l'un de l'autre, ce qui se traduit par la présence de la facette (voir [#13.3.2 et 14.2.2.2]) lien-inverse et de réflexes si-ajout et si-enleve (1) qui assurent que toute modification de l'un entrainera une modification de l'autre.

19.2.1. Instances et accès.

Si la définition de notre univers s'arrête là, nous pouvons déjà décrire des instances de ces 2 modèles, par utilisation de la méthode prédéfinie création [#15.4.1].

```
? (creation lieu 'rome ())
= rome
? (creation personnage 'cesar '((bercail rome)))
= cesar
```

On a là un lieu, rome, et un personnage, cesar, dont le bercail est rome. On peut regarder le contenu de ces 2 nouveaux objets (2):

```
? ^Vcesar
(cesar (est-un (value personnage)
              (herit personnage avant-JC objet-ideal))
 (instance-de (value personnage)
              (bercail (value . rome)))
= cesar
```

(1) Ces réflexes sont ici des macros prédéfinies [#28.8].

(2) L'édit ion obtenue par ^V (voir PRETTY, [#28.5]) est ici "expurgée" pour simplifier le contenu des objets.

Il est aussi possible de lire l'attribut d'un objet, ou de lui donner une valeur.

```
? (bercaill cesar)
= rome
? (:= lieu-actuel cesar rome)
= rome
```

A l'issue de cette affectation, les réflexes ont bien fonctionné:

```
? (lieu-actuel cesar)
= rome
? (présents rome)
= (cesar)
```

On peut aussi vérifier la présence de quelqu'un:

```
? (:? lieu-actuel cesar 'bolegna)
= ( )
? (:? présents rome cesar)
= (cesar)
? (:? lieu-actuel cesar rome)
= rome
```

Enfin, l'effacement se fait s'il est possible, accompagné de ses réflexes.

```
? (:- présents rome 'neron)
= ( )
? (:- présents rome)
= (cesar)
? (lieu-actuel cesar)
= ( )
```

Mais on peut toujours rajouter ce qui n'y est pas:

```

? (:+ presents rome cesar)
= cesar
? (:+ presents rome '(neron cesar))
= (neron)
? (presents rome)
= (cesar neron)

```

19.2.2. Comportement des personnages.

Nous reprenons maintenant où nous en étions dans la création de notre univers en introduisant un lieu privilégié, un no man's land: la forêt, lieu des rencontres les plus folles,

```

(lieu foret
  (presents
    (si-ajout ((saluer (car value)
                      (un-des (presents foret)
                              (car value)))))))

```

et 2 comportements pour les personnages: aller-en-foret et saluer, qui sont des méthodes, s'est à dire des fonctions génériques dépendant de l'objet (du personnage) auquel on les applique [57.2].

La méthode aller-en-foret consiste à déplacer le personnage, s'il est au bercail, vers la forêt. La méthode saluer sera définie pour chaque type de personnage.

```

(personnage (aller-en-foret
             (methode ()) (when (:? lieu-actuel (bercail))
                               (:= lieu-actuel foret))))
           (saluer (methode)))

```

Dans la forêt, un réflexe si-ajout sur les presents impose au nouvel arrivant de saluer l'un-des (3) personnages présents.

De tels comportements sont appelés comme des fonctions LISP, le premier argument étant l'objet sur lequel a été défini le comportement.

(3) UN-DES est une fonction qui choisit aléatoirement un élément de son premier argument, à l'exclusion de ses autres arguments: ici, il s'agit de choisir l'un quelconque des anciens présents. C'est une fonction de l'exemple.

```

? (aller-en-foret cesar)
il n y a pas de methode saluer pour cesar avec (())
= forêt
? (resents rome)
= (neron)
? (resents forêt)
= (cesar)
? (aller-en-foret neron)
il n y a pas de methode saluer pour neron avec (cesar)
= forêt
? (resents forêt)
= (cesar neron)
? (resents rome)
= ()

```

Comme nous n'avons pas encore défini la méthode saluer, la méthode `comport-errer` [§7.2.1] est appelée et imprime le message.

Mais la valeur renvoyée par la méthode `aller-en-foret` est, comme pour tout `pregn` (implicite ici), la valeur de la dernière forme évaluée, c'est à dire (`:= lieu-actuel forêt`).

19.2.3. Classes de personnages et instances de lieux.

Nous allons définir quelques classes de personnages, des romains et des gaulois, ces derniers étant eux-même subdivisés en galle-romains, irréductibles et druides (ce n'est pas une partition).

A ces classes de personnages sont associés des lieux: `camp-romain`, `villa-galle-romaine` et `village-gaulois`. Les comportements (en particulier `saluer`) vont être précisés pour chaque classe de personnage.

19.3. Les gaulois.

Nous avons déjà vu pour les personnages de notre univers 2 comportements: les méthodes `aller-en-foret` et `saluer`. Ce dernier comportement n'étant pas encore défini, nous allons le préciser pour un type de personnage: les gaulois.

Il prend un argument `x` en plus du récepteur du message `frame*` (4) [§13.4.5]. Il va consister à imprimer un message, puis à demander à l'autre (`x`) de dire bonjour, au moyen d'un troisième comportement, la méthode `bonjour`.

Ce dernier comportement a la même syntaxe que la méthode `saluer`, et consiste pour les 2 intervenants à se taper sur le ventre (et à la faire savoir par un message imprimé), puis à assommer chacun un sanglier.

Les sangliers ne sont pas des objets de l'univers, mais simplement des symboles engendrés au moyen de la fonction `fgensym`. Ces sangliers sont rangés, au moyen de l'opérateur `:=`, dans l'attribut à valeur multiple que possède chaque gaulois `sangliers-assommés`.

(4) prononcer "Frais estaire" (et non Singer Roger)

```

(personnage gaulois
  (saluer (methode (x)
    (print "      " frame+ " dit bonjour a " x)
    (bonjour x frame+)))
  (bonjour (methode (x)
    (print "      " x " et " frame+
      " se tapent le ventre.")
    (:+ sangliers-assommes x (fgensym 'sanglier))
    (:+ sangliers-assommes (fgensym 'sanglier))))
  (sangliers-assommes
    (si-ajout ((when (cdr (sangliers-assommes))
      (:= lieu-actuel (berceil)))))))

```

Comme cela apparait dans la definition de l'objet gaulois, un retour au berceil a lieu lorsque le personnage a assomme au moins 2 sangliers. Ceci est fait à l'aide d'un reflexe si-ajout sur l'attribut.

19.3.1. Les comportements des gaulois.

Reprenons les instances créées au paragraphe précédent, en y ajoutant quelques gaulois.

```

? (prezents forêt)
= (cesar neron)
? (creation lieu 'gergovia ())
= gergovia
? (creation gaulois 'vercingetorix '((berceil gergovia)))
= vercingetorix
? (:= lieu-actuel vercingetorix gergovia)
= gergovia
? ~vercingetorix
(vercingetorix (est-un (value gaulois)
  (herit gaulois personnage avant-JC
    objet-ideal))
  (instance-de (value gaulois))
  (berceil (value . gergovia))
  (lieu-actuel (value . gergovia)))
= vercingetorix

```

Regardons maintenant ce qui se passe si nous envoyons ce gaulois en forêt, lieu où se trouvent déjà cesar et neron.

```
? (aller-en-foret vercingetorix)
vercingetorix dit bonjour a cesar
il n y a pas de methode bonjour pour cesar avec (vercingetorix)
= forêt
? (presents forêt)
= (cesar neron vercingetorix)
```

Cette fois, ce n'est pas la méthode saluer qui a déclenché l'appel à ~~comport-er~~er, mais la méthode bonjour, qui a été appelée sur l'objet cesar où elle n'est pas définie.

Nous reviendrons sur ce comportement après avoir défini d'autres classes de personnages.

19.3.2. Les druides.

Chez les gaulois, les druides sont de vénérables sages qui ont toujours l'esprit dans les hautes sphères de la science, ce qui se traduit par une certaine distraction quand ils saluent quelqu'un.

```
(gaulois druide
  (saluer
    (methode (x)
      (print " bonjour madame " x
            " - oh druide " frame+))))
```

Si nous rajoutons un druide aux objets déjà créés, nous pourrions constater que la méthode que nous venons de définir masque la méthode saluer qui a été précédemment définie pour les gaulois en général.

```
? (creation druide 'docteurjekoux ((bercaill gergovie)
?                                     (lieu-actuel gergovie)))
= docteurjekoux
? (saluer docteurjekoux vercingetorix)
bonjour madame vercingetorix - oh druide docteurjekoux
= docteurjekoux
? (saluer vercingetorix docteurjekoux)
vercingetorix dit bonjour a docteurjekoux
vercingetorix et docteurjekoux se tapent le ventre.
= sanglier.2
? (sangliers-assommes vercingetorix)
= (sanglier.1)
? (sangliers-assommes docteurjekoux)
= (sanglier.2)
```

Cependant, les druides ne vont pas en forêt pour le plaisir de chasser le sanglier ou de saluer les autres, mais pour cueillir du gui, base universelle de la médecine gauloise. Bien sûr, lorsqu'un druide a besoin de gui et n'en a plus, il doit aller en forêt pour en cueillir.

Pour aller en forêt, un druide fera donc comme tout le monde, mais, en plus, rentrera au bercail rapidement en ramenant du gui. Il ne s'agit donc plus de masquer une méthode par une autre, mais de la compléter. Ceci est réalisé par l'applicateur :> [#13.6.9], qui va chercher le super-comportement [#7.4], c'est à dire la méthode suivante dans l'héritage [#9] de l'objet appelant.

```
(druide (aller-en-foret (methode ()
                        (and (:> aller-en-foret)
                             (:= gui (makelist 5 'bouquet))
                             (:= lieu-actuel (bercail))))))
      (gui (si-besoin ((aller-en-foret))))))
```

Notons que la valeur renvoyée par la méthode aller-en-foret ne sera pas écrite dans l'attribut gui après l'évaluation du si-besoin, puisque une autre valeur a déjà été écrite durant cette évaluation, et que c'est la première valeur qui est gardée [#5.4.4].

```
? (présents forêt)
= (cesar neron vercingetorix)
? (aller-en-foret docteurjekouix)
bonjour madame cesar - oh druide docteurjekouix
= gergovie
? (présents forêt)
= (cesar neron vercingetorix)
? ~docteurjekouix
(docteurjekouix (est-un (valeur druide)
                       (hérit druide gaulois personnage
                          avant-JC objet-ideal))
               (instance-de (valeur druide)
                             (bercail (valeur . gergovie))
                             (lieu-actuel (valeur . gergovie))
                             (gui (valeur bouquet bouquet bouquet bouquet bouquet)))
               = docteurjekouix
```

13.4. Les irréductibles.

Un second type de gaulois est irréductible, pour lequel la valeur (facette valeur) de l'attribut bercail est village-gaulois.

Ils se distinguent par quelques attributs supplémentaires (bien connus des connaisseurs) qui sont potien (magique?), sangliers-manges et romaine-essences.

```

(gaulois irreductible
  (aller-en-foret
    (methode ()
      (and (potion)
            (:> aller-en-foret)
            (:+ sangliers-assommes (fgenaym 'sanglier))))))
(bercail (value .village-gaulois))
(potion)
(sangliers-manges)
(romains-assommes))

```

Mais ils se distinguent aussi par leur manière d'aller en forêt: ils commencent par s'assurer qu'ils ont de la potion, puis font comme tout le monde, c'est à dire comme auraient fait des représentants du reste de leur héritage, et terminent en assommant encore un sanglier. Là encore, nous utilisons l'applicateur :> pour définir ce comportement.

19.4.1. Le chef (courageux et embrageux) et le druide.

Nous allons maintenant définir deux irréductibles qui ont un rôle important pour fournir de la potion aux autres irréductibles: le chef abracadourcix et le druide panoramix. Ils ont tous les deux un comportement spécial fournir-potion.

Définissons tout d'abord un troisième type d'objet, dont l'utilisation se précisera dans la suite: marmite, avec un attribut contenu.

Le principe de fourniture de la potion est le suivant: tout irréductible ayant besoin de potion demande au chef de lui en fournir. S'il en reste dans la marmite, le chef en donne une dose au demandeur. Sinon, il demande au druide d'en fournir.

```

(irreductible (potion (si-besoin ((fournir-potion abracadourcix))))))
(avant-JC marmite (contenu))
(irreductible abracadourcix
  (fournir-potion (methode ()
    (or (:- contenu marmite 'dose)
        (fournir-potion panoramix))))))

```

Nous voyons qu'ici, la méthode fournir-potion ne connaît pas le demandeur: c'est donc la valeur renvoyée dose qui sera mise dans l'attribut potion du demandeur.

Quant au druide, il fabrique de la potion à partir du gui: il vérifie qu'il a du gui, en prend un bouquet, et fait de la potion, qu'il laisse dans la marmite (20 doses), non sans avoir rempli la gourde d'asterix (qui sera définie dans le paragraphe suivant).


```

(druide panoramix
  (est-un (value irreductible))
  (fournir-potion
    (methode ()
      (when (gui)
        (:- gui frame+ 'bouquet)
        (:= contenu marmite (makelist 20 'dose))
        (:= gourde asterix (makelist 5 'dose))
        'dose))))))

```

Notons que panoramix a un double héritage: c'est un druide et un irreductible. Que va-t-il donc se passer s'il va en forêt (S)? L'héritage [#3] de Panoramix est en effet le suivant: (druide irreductible gauleis personnage avant-JC objet-ideal).

La première méthode aller-en-forêt rencontrée est donc celle définie pour tout druide.

Mais cette méthode commence par un appel au super-comportement, qui va être dans ce cas la méthode suivante trouvée dans l'héritage de panoramix, c'est à dire celle de tout irreductible.

Cette dernière méthode commence par une vérification de l'attribut potion, ce qui, par le réflexe si-besoin, va appeler la méthode fournir-potion du chef abracadouré. S'il n'y a pas de potion dans la marmite, celui-ci va demander à panoramix d'en fabriquer.

Si panoramix, à ce moment là, n'a plus de gui, le réflexe si-besoin de tout druide va se déclencher, qui consiste pour le druide à (aller-en-forêt): ça tombe bien, c'est justement ce qu'il était en train de faire (S)... Yafool détecte ce genre de bouclage, et les si-besoin déclenchés vont renvoyer NIL [#5.4.3]. Oufffff!!!

Mais cela ne résout pas le problème des irreductibles, qui est de pouvoir se fournir en potion. En effet, faute de potion, le druide, comme tout irreductible, ne pourra pas aller en forêt, et ne pourra donc pas ramener de gui pour faire de la potion.

Panoramix sera donc doté d'une méthode spécifique pour aller-en-forêt, qui consistera à envoyer d'abord en forêt un irreductible présent au village et ayant de la potion, puis à aller lui-même en forêt, comme s'il avait de la potion. Ceci peut être fait, par liaison dynamique de la valeur de l'attribut potion, au moyen de la primitive fwith [#8.2].

(S) Question à 20 doses!

(O) Erreur fatale : pile pleine etc

```

(panoramix
  (aller-en-foret
    (methode ()
      (aller-en-foret
        (any (lambda (x) (and (not (x frame+)) (potion x) x))
              (presente village-gaulois)))
        (fwith ((frame+ potion t))
                (:> aller-en-foret))))

```

19.4.2. Des individualités assez fortes.

Pour que la méthode `aller-en-foret` définie pour `panoramix` soit efficace, il faut que certains irréductibles aient toujours de la potion. Et c'est là que nous définissons les héros de notre histoire: `asterix`, qui a toujours sur lui une gourde pouvant contenir de la potion, et `obelix`, qui est tombé dans la potion quand il était petit, et qui a un réflexe bien à lui (7) quand il assomme un romain.

```

(irréductible asterix
  (potion (si-besoin ((:- gourde 'dose))))
  (gourde))

(irréductible obelix
  (potion (value . tombe-dedans-petit))
  (romains-assomme
    (si-ajout ((and (:- casque (car value+))
                    (:% casques-pis '1+))))
    (casques-pis (value . 0)))

```

Voyons ce qui se passe, si la création de notre univers s'arrête là, lorsque `asterix` est envoyé en forêt, sa gourde étant vide.

Le réflexe `si-besoin` de la gourde d'`asterix` ayant renvoyé `NIL`, c'est le réflexe plus général de tout irréductible qui s'est appliqué, et `panoramix`, n'ayant plus de `gui`, a dû `aller-en-foret` en chercher et fabriquer de la potion.

Quant au fait que tout le monde soit rentré au bercail, nous laissons au lecteur le soin de compter les sangliers (8)... Nous donnerons un exemple complet à la fin de ce chapitre, mais il nous faut encore définir quelques entités.

(7) voir "Asterix Gladiateur"

(8) ce change des marions en cas d'innocence!

```

? (creation lieu 'village-gaulois
? '(resents asterix obelix panoramix))
= village-gaulois
? (aller-en-foret asterix)
boucle enrayee asterix potion
obelix dit bonjour a ()
il n y a pas de methode bonjour pour () avec (obelix)
bonjour madame obelix - oh druide panoramix
asterix dit bonjour a obelix
asterix et obelix se tapent le ventre
= sanglier.5
? (resents foret)
= ()
? (resents village-gaulois)
= (panoramix obelix asterix)
? (sangliers-assommes panoramix)
= (sanglier.2)
? (gui panoramix)
= (bouquet bouquet bouquet bouquet)
? (length (contenu marmite))
= 19
? (sangliers-assommes obelix)
= (sanglier.1 sanglier.4)
? (sangliers-assommes asterix)
= (sanglier.3 sanglier.5)
? (gourde asterix)
= (dose dose dose dose dose)

```

19.5. Les romains et galle-romains.

Définissons pour les personnages un autre attribut qui préfigure ce qui va se passer dans la suite: `prend-sur-la-tete`. La valeur de l'attribut `prend-sur-la-tete` est un autre personnage, qui arrive (9) sur la tête du premier, depuis un autre lieu.

Lorsqu'un personnage en prend un autre sur la tête, le second doit, pour la cohérence de l'ensemble, être mis dans le même lieu que le premier. Ceci peut être fait par un réflexe `si-ajout` [#6.3], qui va utiliser `value+` [#6.3.2] pour le déplacer.

Ce réflexe utilise l'opérateur d'écriture `:=` [#13.6.4] et l'émission de l'objet récepteur du message, `frame+` [#13.4.5] dans les accès en lecture.

```

(personnage (prend-sur-la-tete
              (si-ajout ((:= lieu-actuel value+ (lieu-actuel))))))

```

Les romains sont définis de façon similaire aux gaulois, leur berceuil étant le `camp-romain`. Ils sont munis d'une méthode `ave`, et leur méthode `saluer` consiste à imprimer un message et à demander à l'autre de dire `ave`.

(9) en général par voie sérielle:

```
(personnage romain
  (berceuil (value . camp-romain))
  (saluer (methode (x)
    (print " " frame+ " dit ave a " x)
    (ave x frame+)))
  (ave (methode (x)
    (print " " ave " x " - ave " frame+)))
  (casque (est-un boolean)
    (si-ajout
      ((print "et votre casque " frame+ "?")))))
```

Remarquons l'attribut casque et son réflexe si-ajout, qui prendra tout son sel quand le romain passera devant son centurion après être passé trop près d'Obélix.

Comme les gaulois, les romains ont un chef, le centurion Caligulaminus, qui a une méthode de chef pour aller-en-foret: envoyer en forêt 2 romains présents au camp. Et il fait cela chaque fois qu'un personnage lui atterrit sur la tête (10)! Ceci est réalisé par un réflexe si-ajout.

```
(romain caligulaminus
  (aller-en-foret
    (methode ()
      (repeat 2 (aller-en-foret
        (un-des (présents camp-romain))))))
  (prend-sur-la-tete (si-ajout ((aller-en-foret))))))
```

La dernière classe de personnage est un sous type de gaulois, qui est le "collabo" (11): le gallo-romain.

Ses caractéristiques sont d'avoir une méthode saluer beaucoup plus diplomatique que les autres personnages, et, en plus de la méthode bonjour héritée des gaulois, une méthode ave comme les romains.

```
(gaulois gallo-romain
  (saluer (methode (x)
    (if (=? est-un x gaulois) (bonjour x frame+)
        (ave x frame+))))
  (ave (methode (x)
    (print " " ave, venera " x " - ave " frame+)))
  (berceuil (value . ville-gallo-romaine))
  (prend-sur-la-tete (si-ajout ((:= lieu-actuel (berceuil))))))
```

(10) ce qui dénote un certain entêtement, que nous verrons à l'œuvre par la suite...
 (11) du célèbre album "Le Combat des Chefs"

Et tout gaullo-romain qui prend sur la tête quelqu'un d'autre rentre au bercail...

19.6. Les lieux.

Nous avons déjà défini le lieu forêt. Trois autres lieux existent dans notre univers, qui sont les valeurs de l'attribut bercail des différents personnages.

Le lecteur étant maintenant familiarisé avec les primitives de YAFOOL, nous le laisserons savourer la définition du village-gaulois.

```

(lieu village-gaulois
 (chef-de-village (value . abraracourcix))
 (reserve-sangliers
  (si-ajout ((when (> (length (reserve-sangliers))
                       (length (instanciation irreductible))))
             (:= banquet))))))
 (banquet (est-un boolean)
  (si-ajout
   ((terpri)
    (print "——BANQUET AU VILLAGE——")
    (terpri)
    (:+ presents
     (:- presents foret
      (instanciation irreductible))))
   (terpri)
   (fmapc + sangliers-manges
    (instanciation irreductible)
    (reserve-sangliers)
    (:- banquet))))
 (presents (si-ajout
  (:+ reserve-sangliers
   (fmapcan :- sangliers-assommes value+))))))
(irreductible
 (sangliers-manges
  (si-ajout
   (:- reserve-sangliers village-gaulois value+))))

```

Notez l'utilisation des fonctions de mapping [20.5.2] fmapc et fmapcan, ainsi que la définition et l'utilisation de l'attribut boolean [15.6] banquet, destiné à éviter les empilements de banquets en cas de retour massif au village (réflexe si-ajout de reserve-sangliers).

La définition du camp-romain est, elle, plus simple, et consiste en des réflexes sur l'attribut presents.

```

(lieu camp-romain
 (centurion (value . caligulaminus))
 (presente (si-besoin ((terpri)
 (print "—RASSEMBLEMENT AU CAMP—")
 (terpri)
 (:= presente foret
 (instanciation romain))
 (terpri))))
 (si-ajout ((fmapc := casque value@))))))

```

Terminons par la ville-gallo-romaine qui est un lieu quelconque, sans caractéristique spéciale.

```

(lieu ville-gallo-romaine)

```

Le fonctionnement de ces lieux sera illustré par la suite.

19.7. De l'art de se faire des politesses.

Nous avons vu que les gaulois avaient une méthode `bonjour` et les romains une méthode `ave`.

Etant donné la définition de la méthode `saluer` pour les différents personnages, une absence de méthode se produira chaque fois qu'un gaulois (non gallo-romain) saluera un romain ou vice-versa, déclenchant un appel de la méthode `comport-error` [§7.2.1].

On peut donc redéfinir la méthode `comport-error` de `bonjour` de façon à mettre un peu d'animation dans cette morne forêt...

```

(methode bonjour
 (comport-error
 (methode (x li)
 (prin "
 (ifn x (print "Je suis un pauvre "
 (car li) " solitaire")
 (print "TTCHHAKKKK " (car li) " assomme " x)
 (:+ romains-assommes (car li) x)
 (:= potion (car li) 'dose)
 (:= prend-sur-la-tete
 (or (un-des (faubset :? est-un
 (presente foret)
 gallo-romain))
 (centurion camp-romain))
 x))))))

```

Notez qu'il y a une Justice, car tout gallo-romain présent en forêt peut prendre sur la tête un romain préalablement assommé par un irréductible.

La définition de la méthode comport-error de ave est du type cessard (12), puisqu'elle consiste à appeler la méthode bonjour de telle façon qu'elle déclenche un appel à comport-error.

```
(methode ave
  (comport-error
    (methode (x li)
      (if x (bonjour (car li) x)
        (print "I am a poor lonesome "
          (car li)))))))
```

Le décor est maintenant planté, il ne reste qu'à ajouter des personnages.

19.8. Les comparses.

Il y a 2 manières de créer des instances: pendant la création de l'univers, au moyen des modèles existants, ou bien en dehors, au moyen de la méthode creation [§15.4.1]. Afin que tous nos personnages aient le même statut, nous utiliserons la première solution.

```
(irreductible cedezix)      (irreductible cetautomatix)
(irreductible seyix)       (irreductible colorix)
(irreductible ordralphabetix) (irreductible smecix)

(romain brutus)      (romain comurus) (romain diafoirus)
(romain droldagus)  (romain langesus) (romain nautilus)
(romain nempeuplus) (romain perclus)

(druide amnesix (est-un (value gallo-romain)))

(gallo-romain aplusbegalix) (gallo-romain hemailletix)
(gallo-romain ibmrix)       (gallo-romain multix)
(gallo-romain soixantuiuidix) (gallo-romain unix)
```

Nous pouvons maintenant commencer les festivités, en envoyant en forêt les différents personnages de notre univers.

19.9. Exemple de session.

L'initialisation du système consiste à mettre tous les personnages à leur place, c'est à dire au bercail, dans l'attribut presents, sauf les chefs, qui ne sont pas comptés dans les presents car cela produirait des

(12) personnage ayant un attribut (booléen) poil-dans-la-main. NDLR.

bouclages dans les tirages au hasard.

```
(:+ presents village-gaulois (instanciation irreductible))
(:- presents village-gaulois abraracourcix)

(:+ presents ville-galle-romaine (instanciation galle-romain))
(:+ presents camp-romain (instanciation romain))
(:- presents camp-romain caligulaminus)
```

Puis nous en enverrons en forêt un certain nombre, les uns après les autres, en éditant la situation à chaque fois, au moyen de la méthode `editer`, et en traçant les modifications des valeurs de certains attributs, au moyen de la méthode `trace-slot`.

```
(fmapc trace-slot
  '(prend-sur-la-tete sangliers-assommes potion contenu
    presents sangliers-manges))
```

Nous avons défini une fonction `aide` (sans argument) qui édite les différents lieux (sauf la forêt), puis demande à l'utilisateur qui il désire envoyer en forêt, jusqu'à ce que la réponse soit `NIL`.

```
VILLAGE-GAULOIS : presents = panoramix asterix obelix cetautomatix
                  ordralphabetix ceyix smecix colorix
                  cedouzix
                  chef-de-village = abraracourcix
```

```
VILLE-SALLO-ROMAINE : presents = amnesix aplusbagalix multix unix
                               soixantunmildix ibmrix hemailletix
```

```
CAMP-ROMAIN : presents = perclus langelus diafoirus nautilus brutus
                       comunrus droldagus nempeuplus
                       centurion = caligulaminus
```

qui va en forêt? perclus

```
(- camp-romain presents (perclus))
(+ forêt presents (perclus))
  perclus dit ave a ()
  I am a poor lonesome perclus
```


VILLAGE-GAULOIS : presents = panoramix asterix obelix cetautomatix
 ordralphabetix coyix smecix colorix
 sedeuzix
 chef-de-village = abraracourcix

VILLE-GALLO-ROMAINE : presents = amnesix aplusbegalix multix unix
 soixantuwimildix ibmrix hemailletix

CAMP-ROMAIN : presents = langelus diafoirus nautilus brutus comunrus
 droldegus nempplus
 centurion = caligulaminus

qui va en foret? asterix

- (- village-gaulois presents (obelix))
- (+ foret presents (obelix))
 - obelix dit bonjour a perclus
 - TTCHHAKKKK obelix assomme perclus
- (+ caligulaminus prend-sur-la-tete perclus)
- (- camp-romain presents (brutus))
- (+ foret presents (brutus))
 - brutus dit ave a obelix
 - TTCHHAKKKK obelix assomme brutus
- (+ caligulaminus prend-sur-la-tete brutus)
- (- camp-romain presents (nautilus))
- (+ foret presents (nautilus))
 - nautilus dit ave a brutus
 - ave nautilus - ave brutus
- (- camp-romain presents (comunrus))
- (+ foret presents (comunrus))
 - comunrus dit ave a obelix
 - TTCHHAKKKK obelix assomme comunrus
- (+ caligulaminus prend-sur-la-tete comunrus)
- (- camp-romain presents (droldegus))
- (+ foret presents (droldegus))
 - droldegus dit ave a obelix
 - TTCHHAKKKK obelix assomme droldegus
- (+ caligulaminus prend-sur-la-tete droldegus)
- (- camp-romain presents (langelus))
- (+ foret presents (langelus))
 - langelus dit ave a obelix
 - TTCHHAKKKK obelix assomme langelus
- (+ caligulaminus prend-sur-la-tete langelus)
- (- camp-romain presents (nempplus))
- (+ foret presents (nempplus))
 - nempplus dit ave a nautilus
 - ave nempplus - ave nautilus
- (- camp-romain presents (diafoirus))
- (+ foret presents (diafoirus))
 - diafoirus dit ave a comunrus
 - ave diafoirus - ave comunrus

4/7/86

Y A F O O L 2.1

---RASSEMBLEMENT AU CAMP---

(- foret presents (perclus langelus diafoirus nautilus brutus
comunrus droidegus nempeuplus))
(+ camp-romain presents (perclus langelus diafoirus nautilus brutus
comunrus droidegus nempeuplus))
et votre casque perclus?
et votre casque langelus?
et votre casque brutus?
et votre casque comunrus?
et votre casque droidegus?

(- camp-romain presents (perclus))
(+ foret presents (perclus))
perclus dit ave a obelix
TTCHHAKKKK obelix assomme perclus
(+ caligulaminus prend-sur-la-tete perclus)
(- camp-romain presents (comunrus))
(+ foret presents (comunrus))
comunrus dit ave a obelix
TTCHHAKKKK obelix assomme comunrus
(+ caligulaminus prend-sur-la-tete comunrus)
(- camp-romain presents (brutus))
(+ foret presents (brutus))
brutus dit ave a perclus
ave brutus - ave perclus
(- camp-romain presents (nempeuplus))
(+ foret presents (nempeuplus))
nempeuplus dit ave a comunrus
ave nempeuplus - ave comunrus
(- camp-romain presents (nautilus))
(+ foret presents (nautilus))
nautilus dit ave a brutus
ave nautilus - ave brutus
(- camp-romain presents (droidegus))
(+ foret presents (droidegus))
droidegus dit ave a obelix
TTCHHAKKKK obelix assomme droidegus
(+ caligulaminus prend-sur-la-tete droidegus)
(- camp-romain presents (diafoirus))
(+ foret presents (diafoirus))
diafoirus dit ave a nempeuplus
ave diafoirus - ave nempeuplus
(- camp-romain presents (langelus))
(+ foret presents (langelus))
langelus dit ave a nautilus
ave langelus - ave nautilus

---RASSEMBLEMENT AU CAMP---

(- foret presents (perclus langelus diafoirus nautilus brutus
comunrus droidegus nempeuplus))
(+ camp-romain presents (perclus langelus diafoirus nautilus brutus
comunrus droidegus nempeuplus))

(- camp-romain presents (diafoirus))
(+ foret presents (diafoirus))
diafoirus dit ave a obelix
TTCHHAKKKK obelix assomme diafoirus
(+ caligulaminus prend-sur-la-tete diafoirus)
(- camp-romain presents (langelus))

```

(+ foret presents (langelus))
  langelus dit ave a diafoirus
  ave langelus - ave diafoirus
(- camp-romain presents (nautilus))
(+ foret presents (nautilus))
  nautilus dit ave a obelix
  TTCHHAKKKK obelix assomme nautilus
(+ caligulaminus prend-sur-la-tete nautilus)
(- camp-romain presents (perclus))
(+ foret presents (perclus))
  perclus dit ave a langelus
  ave perclus - ave langelus
(- camp-romain presents (comunrus))
(+ foret presents (comunrus))
  comunrus dit ave a diafoirus
  ave comunrus - ave diafoirus
(+ obelix sangliers-assommes (sanglier.1))
(- village-gaulois presents (panoramix))
(+ foret presents (panoramix))
  bonjour madame nautilus - oh druide panoramix
(+ panoramix sangliers-assommes (sanglier.2))
(- foret presents (panoramix))
(+ village-gaulois presents (panoramix))
(- panoramix sangliers-assommes (sanglier.2))
(+ marmite contenu (dose dose dose dose dose dose dose dose dose dose))
(+ asterix potion dose)
(- village-gaulois presents (asterix))
(+ foret presents (asterix))
  asterix dit bonjour a perclus
  TTCHHAKKKK asterix assomme perclus
(- asterix potion dose)
(+ caligulaminus prend-sur-la-tete perclus)
(- camp-romain presents (nempoplus))
(+ foret presents (nempoplus))
  nempoplus dit ave a comunrus
  ave nempoplus - ave comunrus
(- camp-romain presents (brutus))
(+ foret presents (brutus))
  brutus dit ave a nempoplus
  ave brutus - ave nempoplus
(+ asterix sangliers-assommes (sanglier.3))

```

4/7/86

Y A F O O L 2.1

VILLAGE-GAULOIS : presents = cetautomatix ordralphabetix ceyix smecix
 colorix cedeuzix panoramix
 reserve-sangliers = sanglier.2
 chef-de-village = abrarcourcix

VILLE-GALLO-ROMAINE : presents = amnesix aplusbegalix multix unix
 soixantuimildix ibarix hemaitletix

CAMP-ROMAIN : presents = droidegus
 centurion = caligulaminus

qui va en foret? unix

(- ville-gallo-romaine presents (unix))
 (+ foret presents (unix))
 asterix et unix se tapent le ventre.
 (+ asterix sangliers-assoimes (sanglier.4))
 (- foret presents (asterix))
 (+ village-gaulois presents (asterix))
 (- asterix sangliers-assoimes (sanglier.3 sanglier.4))
 (+ unix sangliers-assoimes (sanglier.5))

VILLAGE-GAULOIS : reserve-sangliers = sanglier.2 sanglier.3
 sanglier.4
 presents = cetautomatix ordralphabetix ceyix smecix
 colorix cedeuzix panoramix asterix
 chef-de-village = abrarcourcix

VILLE-GALLO-ROMAINE : presents = amnesix aplusbegalix multix
 soixantuimildix ibarix hemaitletix

CAMP-ROMAIN : presents = droidegus
 centurion = caligulaminus

qui va en foret? multix

(- ville-gallo-romaine presents (multix))
 (+ foret presents (multix))
 ave multix - ave langelus

VILLAGE-GAULOIS : reserve-sangliers = sanglier.2 sanglier.3
 sanglier.4
 presents = cetautomatix ordralphabetix ceix smecix
 colorix cedouzix panoramix asterix
 chef-de-village = abraracoursix

VILLE-GALLO-ROMAINE : presents = amnesix aplusbegalix soixantuiildix
 ibmrix hemailletix

CAMP-ROMAIN : presents = droidegus
 centurion = caligulaminus

qui va en foret? smecix

- (- marmite contenu (dose))
- (+ smecix potion dose)
- (- village-gaulois presents (smecix))
- (+ foret presents (smecix))
 - smecix dit bonjour a comunrus
 - TTCHHAKKKK smecix assomme comunrus
- (- smecix potion dose)
- (+ unix prend-sur-la-tete comunrus)
- (- foret presents (unix))
- (+ ville-gallo-romaine presents (unix))
- (+ smecix sangliers-assomes (sanglier.S))

4/7/86

Y A F O O L 2.1

VILLAGE-GAULOIS : presents = cetautomatix ordralphabetix ceyix
colorix cadeuzix panoramix asterix
reserve-sangliere = sanglier.2.sanglier.3
sanglier.4
chef-de-village = abrarcourcix

VILLE-GALLO-ROMAINE : presents = amnesix apusbegalix soixantunmildix
ibmrix hamilletix unix

CAMP-ROMAIN : presents = droidegus
centurion = caligulaminus

qui va en foret? ceyix

(- marmite contenu (dose))
(+ ceyix potion dose)
(- village-gaulois presents (ceyix))
(+ foret presents (ceyix))
ceyix dit bonjour a langelus
TTCHHAKKKK ceyix asseomme langelus
(- ceyix potion dose)
(+ multix prend-sur-la-tete langelus)
(- foret presents (multix))
(+ ville-gallo-romaine presents (multix))
(+ ceyix sangliers-asseomes (sanglier.7))

VILLAGE-SAULOIS : presents = cetautomatix ordralphabetix colorix
 cedezix panoramix asterix
 reserve-sangliers = sanglier.2 sanglier.3
 sanglier.4
 chef-de-village = abrareacourcix

VILLE-GALLO-ROMAINE : presents = amnesix aplusbegelix soixantumildix
 ibmrix hemailletix unix multix

CAMP-ROMAIN : presents = droldegus
 centurion = caligulaminus

qui va en foret? caligulaminus

(- camp-romain presents (droldegus))
 (+ foret presents (droldegus))
 droldegus dit ave a smecix
 TTCHHAKKKK smecix assomme droldegus
 (+ caligulaminus prend-sur-la-tete droldegus)

---RASSEMBLEMENT AU CAMP---

(- foret presents (perclus langelus diafoirus nautilus brutus
 comunrus droldegus nempeuplus))
 (+ camp-romain presents (perclus langelus diafoirus nautilus brutus
 comunrus droldegus nempeuplus))
 et votre casque diafoirus?
 et votre casque nautilus?

(- camp-romain presents (nautilus))
 (+ foret presents (nautilus))
 nautilus dit ave a ceyix
 TTCHHAKKKK ceyix assomme nautilus
 (+ caligulaminus prend-sur-la-tete nautilus)

(- camp-romain presents (droldegus))
 (+ foret presents (droldegus))
 droldegus dit ave a ceyix
 TTCHHAKKKK ceyix assomme droldegus
 (+ caligulaminus prend-sur-la-tete droldegus)

(- camp-romain presents (nempeuplus))
 (+ foret presents (nempeuplus))
 nempeuplus dit ave a obelix
 TTCHHAKKKK obelix assomme nempeuplus
 (+ caligulaminus prend-sur-la-tete nempeuplus)

(- camp-romain presents (comunrus))
 (+ foret presents (comunrus))
 comunrus dit ave a droldegus
 ave comunrus - ave droldegus

(- camp-romain presents (brutus))
 (+ foret presents (brutus))
 brutus dit ave a nautilus
 ave brutus - ave nautilus
 (- camp-romain presents (diafoirus))

(+ foret presents (diafoirus))
 diafoirus dit ave a smecix
 TTCHHAKKKK smecix assomme diafoirus
 (+ caligulaminus prend-sur-la-tete diafoirus)
 (- camp-romain presents (perclus))
 (+ foret presents (perclus))
 perclus dit ave a nempeuplus
 ave perclus - ave nempeuplus
 (- camp-romain presents (langelus))
 (+ foret presents (langelus))
 langelus dit ave a nempeuplus
 ave langelus - ave nempeuplus

---RASSEMBLEMENT AU CAMP---

(- foret presents (perclus langelus diafoirus nauutilus brutus
 comunrus droldegus nempeuplus))
 (+ camp-romain presents (perclus langelus diafoirus nauutilus brutus
 comunrus droldegus nempeuplus))
 et votre casque nempeuplus?

(- camp-romain presents (diafoirus))
 (+ foret presents (diafoirus))
 diafoirus dit ave a obelix
 TTCHHAKKKK obelix assomme diafoirus
 (- camp-romain presents (langelus))
 (+ foret presents (langelus))
 langelus dit ave a diafoirus
 ave langelus - ave diafoirus
 (- camp-romain presents (brutus))
 (+ foret presents (brutus))
 brutus dit ave a obelix
 TTCHHAKKKK obelix assomme brutus
 (+ caligulaminus prend-sur-la-tete brutus)
 (- camp-romain presents (perclus))
 (+ foret presents (perclus))
 perclus dit ave a obelix
 TTCHHAKKKK obelix assomme perclus
 (+ caligulaminus prend-sur-la-tete perclus)
 (- camp-romain presents (nempeuplus))
 (+ foret presents (nempeuplus))
 nempeuplus dit ave a perclus
 ave nempeuplus - ave perclus
 (- camp-romain presents (droldegus))
 (+ foret presents (droldegus))
 droldegus dit ave a perclus
 ave droldegus - ave perclus
 (- camp-romain presents (comunrus))
 (+ foret presents (comunrus))
 comunrus dit ave a nempeuplus
 ave comunrus - ave nempeuplus

VILLAGE-GAULOIS : presents = cetautomatix ordralphabetix colorix
 cedeuzix panoramix asterix
 reserve-sangliers = sanglier.2 sanglier.3
 sanglier.4
 chef-de-village = abraracourcix

VILLE-SALLO-ROMAINE : presents = amnesix aplusbegalix soixantumildix
 ibmrix hemmailletix unix multix

CAMP-ROMAIN : presents = neutilus
 centurion = caligulaminus

qui va en foret? cedeuzix

- (- marmite contenu (dose))
- (+ cedeuzix potion dose)
- (- village-gaulois presents (cedeuzix))
- (+ foret presents (cedeuzix))
 - cedeuzix dit bonjour a obelix
 - cedeuzix et obelix se tapent le ventre.
- (+ cedeuzix sangliers-assommes (sanglier.8))
- (+ obelix sangliers-assommes (sanglier.9))
- (- foret presents (obelix))
- (+ village-gaulois presents (obelix))
- (- obelix sangliers-assommes (sanglier.1 sanglier.9))
- (+ cedeuzix sangliers-assommes (sanglier.10))
- (- foret presents (cedeuzix))
- (+ village-gaulois presents (cedeuzix))
- (- cedeuzix sangliers-assommes (sanglier.8 sanglier.10))

VILLAGE-GAULOIS : reserve-sangliers = sanglier.2 sanglier.3
sanglier.4 sanglier.1
sanglier.9 sanglier.8
sanglier.10
presents = setautomatix ordralphabetix colorix
panoramix asterix obelix cedeuzix
chef-de-village = abrarcoureix

VILLE-GALLO-ROMAINE : presents = amnesix apiusbegalix soixantumildix
ibnrix hemmilletix unix multix

CAMP-ROMAIN : presents = nautilus
centurion = caligulaminus

qui va en foret? unix

(- ville-gallo-romaine presents (unix))
(+ foret presents (unix))
eve unix - eve comurus

```
VILLAGE-GAULOIS : reserve-sangliers = sanglier.2 sanglier.3
                    sanglier.4 sanglier.1
                    sanglier.9 sanglier.8
                    sanglier.10
presents = setautomatix ordralphabetix calerix
           panoramix asterix obelix cedeuzix
chef-de-village = abrarecourcix
```

```
VILLE-GALLO-ROMAINE : presents = amnesix aplusbegalix soixantunmildix
                           ibarix hemaillatix multix
```

```
CAMP-ROMAIN : presents = nautilus
                centurion = caligulaminus
```

```
qui va en foret? asterix
```

```
(+ asterix potion dose)
(- village-gaulois presents (asterix))
(+ foret presents (asterix))
    asterix dit bonjour a nempouplus
    TTCHHAKKKK asterix assomme nempouplus
(- asterix potion dose)
(+ unix prend-sur-le-tate nempouplus)
(- foret presents (unix))
(+ ville-gallo-romaine presents (unix))
(+ asterix sangliers-assommees (sanglier.11))
```

VILLAGE-GAULOIS : presents = cetautomatix ordralphabetix colorix
panoramix obelix cedeuzix
reserve-sangliers = sanglier.2 sanglier.3
sanglier.4 sanglier.1
sanglier.9 sanglier.6
sanglier.10
chef-de-village = abraracourcix

VILLE-SALLO-ROMAINE : presents = amnesix plusbegalix soixantunmildix
ibmrix hezbollahix multix unix

CAMP-ROMAIN : presents = nautilus
centurion = caligulaminus

qui va en foret? unix

(- ville-gallo-romaine presents (unix))
(+ foret presents (unix))
ave unix - ave perclus

VILLAGE-GAULOIS : presents = cetautomatix ordralphabetix colorix
 panoramix obelix cedeuzix
 reserve-sangliers = sanglier.2 sanglier.3
 sanglier.4 sanglier.1
 sanglier.9 sanglier.6
 sanglier.10
 chef-de-village = abrarascourcix

VILLE-GALLO-ROMAINE : presents = amnagix aplusbagatix soixantuinidix
 ibarix hemaillotix multix

CAMP-ROMAIN : presents = nautilus
 centurion = caliguleminux

qui va en foret? cedeuzix

- (- village-gaulois presents (cedeuzix))
- (+ foret presents (cedeuzix))
 cedeuzix dit bonjour a ceyix
 cedeuzix et ceyix se tapent le ventre.
- (+ cedeuzix sangliers-assommes (sanglier.12))
- (+ ceyix sangliers-assommes (sanglier.13))
- (- foret presents (ceyix))
- (+ village-gaulois presents (ceyix))
- (- ceyix sangliers-assommes (sanglier.7 sanglier.13))
- (+ cedeuzix sangliers-assommes (sanglier.14))
- (- foret presents (cedeuzix))
- (+ village-gaulois presents (cedeuzix))
- (- cedeuzix sangliers-assommes (sanglier.12 sanglier.14))

—BANQUET AU VILLAGE—

- (- foret presents (asterix smecix))
- (+ village-gaulois presents (asterix smecix))
- (- asterix sangliers-assommes (sanglier.11))
- (- smecix sangliers-assommes (sanglier.6))
- (+ abrarascourcix sangliers-manges (sanglier.2))
- (+ panoramix sangliers-manges (sanglier.3))
- (+ obelix sangliers-manges (sanglier.4))
- (+ cetautomatix sangliers-manges (sanglier.1))
- (+ ordralphabetix sangliers-manges (sanglier.9))
- (+ ceyix sangliers-manges (sanglier.6))
- (+ smecix sangliers-manges (sanglier.10))
- (+ colorix sangliers-manges (sanglier.7))
- (+ cedeuzix sangliers-manges (sanglier.13))
- (+ cedeuzix sangliers-manges (sanglier.12))

```
VILLAGE-GAULOIS : reserve-sangliers = sanglier.14 sanglier.11
                                     sanglier.6
                 presents = catsautomatix ordralphabetix colorix
                           panoramix obelix cayix cedeuzix asterix
                           smecix
                 chef-de-village = abraracourcix
```

```
VILLE-GALLO-ROMAINE : presents = amnesix aplusbegalix soixantunmildix
                                     ibmrix homailletix multix
```

```
CAMP-ROMAIN : presents = nautilus
                 centurion = caligulaminius
```

Bien sûr, comme un tirage aléatoire (13) intervient dans les échanges de messages, une telle session n'est pas complètement reproductible. Mais elle donne une idée du comportement de notre univers, et, bien sûr, des améliorations à y apporter.

Cet exemple de session peut paraître un peu long, mais nous avons voulu qu'il se termine par un banquet au village gaulois...

(13) et l'accumulateur est initialisé par les soins de YAF00L, comme vous le remarquerez par la variété des messages de bienvenue!

ANNEXES

YAF00L 2.1

20. TOUTES LES FONCTIONS.

20.1. Syntaxe.

La syntaxe adoptée est la suivante:

- nom de la fonction
- liste des arguments (des crochets indiquent un argument optionnel), et la liste décrit l'arbre des paramètres (cf. la liaison arguments-paramètres en `Le_Lisp`),
- un `*` indique la présence d'un tag-continuation, et donc le déclenchement d'attachements procéduraux, généralement avec liaison du quadruplet d'environnement,
- le type de la fonction, si ce n'est pas un `EXPR`.

La valeur retournée est décrite dans les commentaires.

Les mots en gras sont des objets système, et les mots en MAJUSCULES des fonctions ou variables.

20.1.1. Facette standard.

Toutes les fonctions avec un argument facette optionnel (`[facet]`) sont en fait des macros s'expansant avec la même fonction suffixée par `-V`: le mécanisme de ces macros a été décrit en [04.2.2.1 et 011.5.1]. A noter qu'il est possible d'avoir des arguments optionnels (`FSEN` ou `FPUSH`), ou en nombre indéterminé (`FMOD`), simultanément avec cette facette automatique. A noter aussi que c'est le slot qui peut manquer, et non la facette [011.5.1].

20.1.2. Plan.

On trouvera d'abord toutes les primitives des frames, c'est-à-dire les fonctions qui accèdent en lecture ou écriture à un triplet frame-slot-facette, puis les primitives de structures de contrôle, avec les variables globales du package `CONTINU`.

Suivent les fonctions de création d'univers, divers utilitaires et les fonctions temporelles et de vidéo. Le chapitre se termine avec les variables globales du package `YAFOOL` et une boîte à outil de quelques réflexes et méthodes utilisables très généralement.

20.2. Primitives des frames.

FADD (frame slot [facet] val)

Fonction d'ajout de valeur, dans un slot-facette à valeur multiple. Retourne VAL s'il y a modification effective.

FADD+ (frame slot [facet] val)

Fonction d'ajout de valeur, dans un slot-facette à valeur multiple. Si VAL est atomique, c'est FADD. Sinon fait un ajout de chacun des éléments de VAL. Dans ce cas retourne la sous-liste de VAL des ajouts effectifs.

FADD++ (frame slot [facet] val)

Combiné de FADD+ et FADD+.

FADD+ (frame slot [facet] val)

Comme FADD avec liaison d'environnement et activation de réflexes si-ajout. Même retour que FADD.

FADD-H (frame slot [facet] val)

FADD++ pour le lien est-un: activation des réflexes si-ajout sur la nouvelle hiérarchie et effacement de facette hérit. Même retour que FADD.

FAPPL (frame slot appl . args)

Activation de l'applicateur APPL lié à SLOT sur ARGS, avec activation des réflexes SI-APPL. La valeur retournée est la valeur retournée par l'applicateur (dans VALUE+), éventuellement modifiée par les réflexes. En cas d'absence d'applicateur, le message comport-error est envoyé à APPL, avec SLOT et ARGS comme arguments.

FAPPL> (heritage appl . args)

Déclenchement d'applicateur avec héritage biaisé: le premier applicateur APPL trouvé dans HERITAGE est appliqué, FRAME+ et SLOT+ ne sont pas modifiés. Par contre HERIT+ l'est.

FCHECK (frame slot [facet] val)

Vérification unique sans activation de réflexe si-besoin ni recherche hiérarchique. Si le test est positif, retourne VAL.

FCHECK-I (frame slot [facet] val)

Vérification unique sans activation de réflexe si-besoin mais avec recherche hiérarchique: démarche en I. Si le test est positif, retourne VAL.

FCHECK-N (frame slot val)

Vérification unique avec activation de réflexe si-besoin et recherche hiérarchique: démarche en N.

FCHECK-P (frame slot [facet] val)

Vérification unique sans activation de réflexe si-besoin ni recherche hiérarchique mais avec recherche de toutes les FACET éventuellement empilées, jusqu'à trouver la valeur FCHECKée: pour les attributs besoins-t d'objets atemporels. Si le test est positif, retourne VAL.

FCHECK-Z (frame slot val) *

Vérification unique avec activation de réflexe si-besoin et recherche hiérarchique: démarche en Z. Si le test est positif, retourne VAL.

FCONC (frame slot [facet] val)

Comme FPUT sans aucun test: affectation pure et simple. Retourne VAL.

FGEN (frame slot [facet] [val])

C'est la variable-fonction associée aux valeurs: avec 3 arguments c'est FGET, avec 4 c'est FCONC ou FREM, suivant que le dernier est non-NIL ou NIL. Est même compatible avec FREM-P.

FGET (frame slot [facet])

Lecture sans activation de réflexe si-besoin ni recherche hiérarchique: retourne la valeur lue.

FGET-A (frame slot [facet]) *

Lecture de toutes les occurrences dans la hiérarchie de la facette cherchée, pour des valeurs multiples.

FGET-A1 (frame slot [facet]) *

Lecture de toutes les occurrences dans la hiérarchie de la facette cherchée, pour des valeurs uniques.

FGET-AT (objet-atemp slot [facet])

OBJET-ATEMP est un objet atemporel: ce mode de lecture se fait par une lecture (FGET) dans l'objet temporel de OBJET-ATEMP, puis dans ce dernier.

FGET-H (frame lien)

Lecture de l'héritage pour le lien est-un, avec facette herit. Comme toutes les primitives de calcul de la fermeture transitive d'un lien, FGET-H utilise le marquage d'objets. FGET-H provoque une erreur en cas de présence dans la hiérarchie de FRAME d'un objet sans LIEN (facette value).

FGET-HB (frame lien)

Comme FGET-H, mais FRAME exclus.

FGET-I (frame slot [facet])

Lecture sans activation de réflexe si-besoin mais avec recherche hiérarchique: démarche en I.

FGET-L (frame lien)

Lecture de la fermeture transitive ordonnée pour un lien multiple; FGET-H sans facette hérité.

FGET-LE (frame lien)

Comme FGET-L, mais FRAME exclus.

FGET-LI (frame lien)

Lecture de l'héritage pour un lien unique.

FGET-N (frame slot)

Lecture avec activation de réflexe si-besoin et recherche hiérarchique; démarche en N.

FGET-T (objet-temp slot [facet])

Symétrique de FGET-AT. OBJET-TEMP est un objet temporel; ce mode de lecture se fait par une lecture (FGET) dans OBJET-TEMP, puis dans son objet atemporel.

FGET-Z (frame slot [facet])

Lecture avec activation de réflexe si-besoin et recherche hiérarchique; démarche en Z. Le traitement de [facet] est particulier: si FACET est fournie, différente de value, c'est FGET-I.

FMEM (frame slot [facet] val)

Vérification multiple sans activation de réflexe si-besoin ni recherche hiérarchique. Retourne la sous-liste de valeurs commençant par VAL.

FMEM-A (frame slot [facet] val)

Vérification dans toutes les occurrences dans la hiérarchie de la facette cherchée, pour des valeurs multiples. Retourne la sous-liste de valeurs commençant par VAL.

FMEM-A1 (frame slot [facet] val)

Vérification dans toutes les occurrences dans la hiérarchie de la facette cherchée, pour des valeurs uniques. Retourne la sous-liste de valeurs commençant par VAL.

FMEM-H (frame lien val)

Comme FMEM-L pour le lien est-un (avec facette hérité).

FMEM-I (frame slot [facet] val)

Vérification multiple sans activation de réflexe si-besoin mais avec recherche hiérarchique; démarche en I. Retourne la sous-liste de valeurs commençant par VAL.

- FRSL** (frame lien val)
 Vérification de l'héritage pour un lien multiple. Retourne la sous-liste de valeurs commençant par VAL.
- FRSL1** (frame lien val)
 Vérification de l'héritage pour un lien unique. Retourne la sous-liste de valeurs commençant par VAL.
- FRSE-N** (frame slot val) *
 Vérification multiple avec activation de réflexe si-besoin et recherche hiérarchique: démarche en N. Retourne la sous-liste de valeurs commençant par VAL.
- FRSE-Z** (frame slot val) *
 Vérification multiple avec activation de réflexe si-besoin et recherche hiérarchique: démarche en Z. Retourne la sous-liste de valeurs commençant par VAL.
- FRSEQ** (frame slot [facet] vals)
 Recherche de la valeur de la FACET du SLOT de FRAME dans la liste VALS. Retourne la sous-liste de VALS commençant par la valeur.
- FRETH** (frame meth . args) *
 Activation de la méthode METH lié à FRAME sur les arguments ARGS. Retourne la valeur retournée par la méthode.
- FRETH>** (heritage meth . args)
 Déclenchement de méthode avec héritage biaisé: la première méthode METH trouvé dans HERITAGE est appliquée, FRAME+ n'est pas modifié. Par contre HERIT+ l'est.
- FMOD** (frame slot [facet] fet . args)
 Si la FACET du SLOT de FRAME a une valeur non-NIL, la remplace par le résultat de l'application de FCT sur cette valeur et les ARGS. Retourne la nouvelle valeur.
- FMOD+** (frame slot [facet] fet . args) *
 Comme FMOD, avec déclenchement de si-ajout.
- FNEW** (frame slot [facet] val)
 Comme FCONC mais pour les ajouts: FNEW / FADD = FCONC / FPUT. Retourne VAL.
- FNEW+** (frame slot [facet] val) *
 Comme FNEW mais avec activation de réflexes: FNEW+ / FADD+ = FNEW / FADD. Retourne VAL.

FNEXT (frame slot [facet])

Enleve la première valeur du triplet FRAME-SLOT-FACET. La retourne.

FNEXT+ (frame slot [facet]) *

Comme FNEXT, avec déclenchement de réflexes si-enleve.

FPUSH (frame slot [facet] [val])

Empile en tête de slot une nouvelle paire facette-valeur, où la valeur est VAL s'il est présent, sinon une copie (COPYLIST) de l'ancienne valeur. Retourne cette valeur.

FPUT (frame slot [facet] val)

Fonction d'affectation de valeur. S'il y a modification, retourne VAL.

FPUT+ (frame slot [facet] val) *

Comme FPUT avec liaison d'environnement et activation de réflexes si-ajout. Même retour que FPUT.

FPUT-H (frame slot [facet] val) *

FPUT+ pour le lien est-un: activation des réflexes si-ajout sur la nouvelle hiérarchie et si-enleve sur l'ancienne, et effacement de la facette herit. Même retour que FPUT.

FPUT-L (frame slot [facet] val) *

Comme FPUT-H, mais pour tous les liens, uniques ou multiples, sans facette herit.

FPUT-P (frame slot [facet] val)

FPUT pour les attributs booléens-t d'objets atemporels: en cas de succès, empile une nouvelle FACET, de valeur VAL, sauf si la précédente valeur était NIL. Même retour que FPUT.

FPUT+P (frame slot [facet] val) *

Idem, avec réflexes.

FREM (frame slot [facet] val)

Fonction d'effacement de valeur: retourne la valeur effacée. Si VAL EQ la valeur, l'efface. Si VAL appartient (MEMQ) à la valeur, l'enlève. Si VAL=T, efface. Si la facette n'a plus de valeur, l'enleve du slot et les slot de l'objet ...

FREMI (frame slot [facet] val)

Comme FREM, mais uniquement pour les valeurs uniques: pour qu'il y ait effet, VAL doit être T, ou bien EQ à la valeur du triplet FRAME SLOT FACET.

FREM1+ (frame slot [facet] val) *

FREM1 avec déclenchement de réflexe si-enleve..

FREM+ (frame slot [facet] val)

Si VAL est atomique, c'est FREM. Sinon enlève tous les éléments de VAL de la valeur. Retourne la liste des éléments effectivement enlevés.

FREM++ (frame slot [facet] val) *

Comme FREM+ avec activation de réflexes si-enleve.

FREM* (frame slot [facet] val) *

Comme FREM avec liaison d'environnement et activation de réflexes si-enleve. Même retour que FREM.

FREM-H (frame slot [facet] val) *

FREM++ pour le lien est-un: activation des réflexes si-enleve sur l'ancienne hiérarchie et effacement de la facette hérit. Même retour que FREM.

FREM-P (frame slot [facet] val)

FREM pour les attributs booléens-t d'objets atemporels: en cas de succès, empile une nouvelle FACET, sans valeur. Même retour que FREM.

FREM*-P (frame slot [facet] val) *

Idem, avec réflexes.

FRPLAC (frame slot [facet] . vals)

VALS est une liste de longueur paire (VAL1 VAL2 ... VAL-2i-1 VAL-2i ...). Remplace le premier VAL-2i rencontré, par VAL-2i-1. Retourne ce dernier.

FRPLAC+ (frame slot [facet] . vals) *

Comme FRPLAC, avec activation de réflexes si-enleve sur VAL-2i, puis si-ajout sur VAL-2i-1 et enlèvement de facette hérit (pour est-un).

20.3. Structures de contrôle.

FAVEC (lobj . liste-de-formes) + dmacro
 Comme FWITH mais avec AVEC (sans PROTECT) au lieu de WITH. Retourne la valeur de la dernière forme (comme un PROGN).

FLETF (lobj . liste-de-formes) + dmacro
 C'est le LET pour les objets. LOBJ est une liste de couples objet-ancêtres. FLETF est implémenté avec WITH. Même valeur qu'un PROGN.

FSET (lobj . liste-de-formes) + dmacro
 Comme FLETF avec AVEC (sans PROTECT) au lieu de WITH.

FWITH (lobj . liste-de-formes) + dmacro
 LOBJ est une liste de quadruplet FSCFJV. FWITH est un WITH utilisant comme fonction-variable FGEN sur les quadruplets.

<= liste-de-formes fexpr
 Déclenche un retardement externe: plus précisément fait une copie au premier niveau (APPEND) de LISTE-DE-FORMES et la NCONC à #:CONTINU:MESSAGE=. Retourne NIL.

<=> liste-de-formes + dmacro
 Déclenche une continuation dialogante: c'est une double continuation, avec mémorisation des quadruplets FSCFJV, et ré-empilement de ceux-ci en sens inverse, avant l'évaluation de LISTE-DE-FORMES. <=> utilise =LET=. Retourne la dernière forme évaluée, qui devient alors la valeur du TAG correspondant à l'échappement.

<=LET= (lvar . liste-de-formes) + fexpr
 Sauvegarde l'environnement provisoire créé par la liste LVAR (LET parallèle), et fait un retardement (<=>) où LISTE-DE-FORMES sera évalué dans cet environnement sauvegardé. Retourne NIL.

=LET=> (lvar . liste-de-formes) + fexpr
 Comme =LET=> pour les continuations dialogantes.

<<= liste-de-formes fexpr
 Déclenche un retardement global externe: retardement récursif jusqu'au plus haut niveau des messages. Retourne NIL.

<<=LET= (lvar . liste-de-formes) + fexpr
 Comme <=let=, pour les retardements globaux.

<- liste-de-formes fexpr
 Déclenche un retardement interne: plus précisément fait une copie au premier niveau (APPEND) de LISTE-DE-FORMES et la NCONC à #:CONTINU:MESSAGE-. Retourne NIL.

~~LET~~ (lvar . liste-de-formes) * fexpr

Comme ~~let~~, pour les retards internes.

~~=>~~ liste-de-formes ~~macro~~

Déclenche une continuation: plus précisément fait un ~~UNEXIT~~ CONTINUATION LISTE-DE-FORMES), où UNEXIT fait un échappement de nom CONTINUATION avec évaluation après l'échappement.

~~=>>~~ liste-de-formes fexpr

Déclenche une continuation globale externe: continuation récursive jusqu'au plus haut niveau des messages. Retourne la dernière évaluation de LISTE-DE-FORMES.

~~LET=>~~ (lvar . liste-de-formes) * fexpr

Sauvegarde l'environnement provisoire créé par la liste LVAR (LET parallèle), et fait une continuation (~~=>>~~) en évaluant LISTE-DE-FORMES dans cet environnement sauvegardé. Retourne la dernière forme évaluée, qui devient alors la valeur du TAG correspondant à l'échappement.

~~LET=>>~~ (lvar . liste-de-formes) * fexpr

Comme ~~let=>~~, pour les continuations globales.

~~=>~~ liste-de-formes ~~macro~~

Comme ~~=>~~, mais en utilisant EXIT: évaluation avant l'échappement. C'est une pseudo-continuation, servant juste à interrompre le fonctionnement normal d'une primitive. Retourne la dernière forme évaluée, qui devient alors la valeur du TAG correspondant à l'échappement.

~~=>>~~ liste-de-formes fexpr

Comme ~~=>>~~, pour les continuations internes.

~~LET=>>~~ (lvar . liste-de-formes) * fexpr

Comme ~~let=>>~~, pour les continuations globales internes.

20.3.1. Le package \$:CONTINU

Il contient les variables globales du tag-continuation qui sont utilisées à chaque déclenchement d'attachement procédural par le système.

\$:CONTINU:CONT-LISTE

variable

\$:CONTINU:CONT-LISTE

variable

Ce sont respectivement les piles d'environnement gérées par les continuations, respectivement internes et externes, avec liaison de λ -variables.

\$:CONTINU:MESSAGE-

variable

\$:CONTINU:MESSAGE-

variable

Ce sont respectivement les piles de messages retardements, respectivement internes et externes, gérées par les

20.4. Primitives d'univers.

Ce sont toutes les fonctions de top-niveau à utiliser en phase de création d'univers.

BIG-BANG

(univ . slots)

fexpr

Initialise les variables du package YAFOOL (cf ci-dessous) et définit un nouveau TOPLEVEL sur le fichier (terminal) en cours de lecture. Seule la lecture d'un atome, ou bien une fin de fichier sur le canal d'entrée courant provoque la sortie de cette boucle.

Si le CAR de la forme lue n'est pas un objet (OBJVAL = NIL), elle est évaluée. Sinon, les formes lues peuvent avoir deux structures (cf. figure de [12.3]):

(modèle nouveau-modèle . slots)

ou bien
(modèle . slots). Dans les 2 cas modèle est un objet pré-existant, éventuellement défini dans le même univers, nouveau-modèle est une nouvelle instance de modèle et slots représente la valeur d'objet initiale de nouveau-modèle (premier cas), ou bien la liste des slots à rajouter, facette par facette (deuxième cas).

Dans la syntaxe de BIG-BANG, UNIV représente l'univers à créer (s'il existe déjà, il est détruit par la méthode efface) et SLOTS a la même signification que plus haut. UNIV est une instance de #:YAFOOL:IDEAL, qui prend alors la valeur UNIV.

Si l'atome de fin de lecture est NIL, il y a déchargement des fonctions de création d'univers et remise en autoloade de BIG-BANG et INSERT-UNIVERS.

F-PARAM

liste-constante

fexpr

La syntaxe de LISTE-CONSTANTE est identique à celle du paramètre SLOTS de BIG-BANG. Pour chaque slot de la liste, F-PARAM crée, non pas un ATTRIBUT, mais une constante. Toutes les facettes doivent être non-standard, sauf la facette value, dont la valeur sera celle du slot: les constantes sont ainsi les seuls objets à ne pas être autoévalués.

INSERT-UNIVERS

(univ . slots)

fexpr

Permet de revenir en phase de création d'univers pour définir dans l'univers UNIV précédemment défini les objets lus en entrée: même principe que BIG-BANG, mais en ajout et non en création pour univ. Peut être utilisé récursivement au cours de la création d'un autre univers.

Le déchargement n'a lieu que si les fonctions n'étaient pas chargées auparavant.

REMOB-FILE-UNIV

()

Comme REMOB-FILE [19.5], pour les fonctions de #:YAFOOL:FN-FILES (fonctions de création d'univers) qui est remis à NIL. Tous les slots autoloade qui possédaient des fonctions de création d'univers marquées dans #:YAFOOL:FN-FILES sont remis en autoloade pour les méthodes de création d'univers et pour le fichier de suffixe ".univ". Les variables globales #:YAFOOL:SPECIAL-FIN et #:YAFOOL:SPECIAL-DEBUT sont remises à NIL.

20.5. Fonctions diverses.

Ce paragraphe regroupe un certain nombre de fonction YAFOOL, puis des utilitaires divers, fonctions de compilation, chargement et mapping.

DEL-ALL-INSTANCES (frame)

Détruit (EFFACE) toutes les instances (récursivement) de frame, à l'exception de celles qui datent de la création de l'univers ayant défini frame.

FCLAMP (frame1 frame2 slot)

Identification physique du SLOT de FRAME1 sur celui de FRAME2. Retourne l'ensemble du slot. Pas d'action si FRAME2 est NIL.

FCLAMP+ (frame1 frame2 slot) *

FCLAMP avec activation de réflexes si-enleve et si-ajout sur les facettes value et sauf, et effacement de facette herit (pour le lien est-un).

FCLAMP2 (frame1 slot1 frame2 slot2)

Comme FCLAMP, pour des slots différents dans les 2 objets: si les 2 slots sont vides, ils sont initialisés par la facette standard de slot1, sans valeur.

FUNCLAMP (frame slot)

C'est la fonction inverse de FCLAMP et FCLAMP+: le SLOT de FRAME est remplacé par sa propre copie (COPYN de profondeur 2): il est alors possible de le modifier sans toucher aux objets dont le SLOT est FCLAMPé sur celui de FRAME.

FCOPY (frame1 frame2)

Récopie (COPYN de profondeur 3) FRAME1 dans FRAME2, avec remplacement de toutes les occurrences de FRAME1 par FRAME2. FRAME2 est autovalué. Les liens est-un sont FCLAMPés. Retourne FRAME2.

N.B. FCOPY et FUNCLAMP utilisent COPYN, fonction comparable à COPYLIST, mais dont un second argument, entier, permet de préciser la profondeur de copie. S'il vaut 0, c'est IDENTITY, 1 c'est APPEND.

GENSYM ([arg])

C'est un équivalent de GENSYM, qui permet d'éviter de mélanger les symboles de l'utilisateur de ceux que génèrent le système, en particulier dans l'expansion des macros. Les variables globales GENSYM-STRING et GENSYM-COUNTER sont dans le package YAFOOL, initialisées à "yaf" et @ respectivement. Un "." est inséré entre la chaîne et le nombre.

La présence d'un argument modifie la première, s'il est numérique, et utilise l'argument à la place de GENSYM-STRING sinon (un peu comme en MACLISP).

- FBET-IM** (lframe lien)
Comme FBET-L, mais LFRAME est une liste d'objets: donne la fermeture transitive du LIEN, pour un objet abstrait dont le LIEN serait LFRAME.
- FPJCA** (liste-de-frame)
Retourne l'un des "plus jeunes communs ancêtres" (PJCA) de LISTE-DE-FRAME.
- FPRETTY** lframe
Fait un pretty-print des valeurs objets des éléments de LFRAME. Abrégé en ^V.
- FPRETTY-ALL** (out [univ]) fexpr
Edite (méthode EDITER) sur le fichier OUT, toutes les frames de l'univers UNIV (OBJET-IDEAL par défaut), et ce récursivement, par édition d'un objet, puis de ses instances triées par ordre alphabétique. En autoloading.
- FPRINT-FOR-READ** (frame)
Edite frame sous une forme telle qu'elle puisse être relue dans une phase de création d'univers. Les liens est-un et instantiation sont modifiés dynamiquement (FWITH) en conséquence. En autoloading.
- FPRINT-ALL** (out [univ]) fexpr
Comme FPRETTY-ALL, mais utilise FPRINT-FOR-READ à la place de la méthode EDITER.
- FREM-21** (frame lien)
Cette fonction efface les facettes sauf et value de LIEN en activant les réflexes si-enleve. Nécessaire à FREMOB et à la méthode EFFACE pour effacer proprement le lien est-un de l'objet avec un déclenchement correct des réflexes.
- FREMOB** (frame)
Détruit récursivement (FREMOB) toutes les instances de FRAME. FUNCLAMP puis efface le lien est-un de FRAME, avec activation de réflexes, avant de le détruire (REMOB). Il est préférable d'utiliser la méthode efface [§15.4.2].
- F-CARAC** (frame liste)
Applique (MAPC) la méthode CARACT, avec FRAME comme argument, à tous les éléments de LISTE, ou à leur CAR, suivant qu'il s'agit d'un atome ou d'un CONS. Une telle LISTE sera appelée une liste de valeurs ou d'attributs-valeurs. F-CARAC est utilisé par les méthodes de création (CREATION et CREATION-T) pour initialisation. Retourne NIL (c'est un MAPC).

NOUVEL-ANCIETRE (frame val)

FUTUR-NOUVEL-ANCIETRE (frame val)

Ces deux fonctions rajoutent un lien est-un, VAL, à l'objet FRAME, si celui-ci ne le contient pas déjà dans sa hiérarchie.

La seconde (FADD+) ne déclenche pas les réflexes, contrairement à la première (FADD-H): elle est destinée à être utilisée en phase de création d'univers.

HARA-KIRI (fun)

fexpr

Cette fonction permet à un réflexe de s'enlever de l'objet où il est implanté au moment de son déclenchement: doit être appelé depuis un réflexe.

LAST-INSTANCE (arg)

dsacro

Retourne la dernière instance de ARG. "Dernier" est ici à entendre au sens du déroulement du programme, et non pas d'un point de vue temporel.

20.5.1. Utilitaires de chargement et compilation.

FUN-AUTOLOAD (file dir . lfn)

fexpr

Comme MY-AUTOLOAD, mais pour des fonctions appelées par FUNCALL ou APPLY.

LELISP ([[core] dir])

Permet de recharger (RESTORE-CORE) une image mémoire Le_Lisp. Sans argument, c'est l'image mémoire standard. Sinon, c'est l'image mémoire du fichier CORE de suffixe #:SYSTEM:CORE-EXTENSION (en général: ".core"), de la directorie DIR (directorie courante par défaut).

MY-AUTOLOAD (file dir . lfn)

fexpr

Cette FEXPR, où seul DIR est évalué, permet de faire des autoloads des fonctions de la liste LFN, dans le fichier FILE de directorie DIR, en mémorisant LFN comme propriété du fichier, ce qui autorise une remise en autoload après REMOS-FILE. Se sert de MY-LOAD.

MY-COMPILE-ALL-IN-CORE ()

Equivalente à COMPILE-ALL-IN-CORE, mais place l'indicateur #:YAFDOL:COMPILE-FLAG pour la compilation des macros. Commence par un REMOS-FILE-UNIV.

MY-COMPILEFILES (dir . lfil)

fexpr

Equivalente à COMPILEFILES, mais place l'indicateur #:YAFDOL:COMPILE-FLAG pour la compilation des macros. La fonction DE est redéfinie pour que ce soit l'ancienne définition de la fonction qui soit compilée et non la nouvelle: il est ainsi possible de compiler des fichiers de fonctions, après l'expansion des macros, ce qui donne un code plus performant. DIR est la directorie, LFIL la liste des fichiers (de

suffixe ".ll") à compiler, les sorties se font dans les fichiers de
suffixe ".cp".

MY-LOAD (dir . lfil) fexpr

FEXPR où seul DIR est évalué: chargement des fichiers LFIL de la
directory DIR. Se sert de MY-PATHNAME et de PROBEFILE. MY-LOAD
retourne la sous liste des fichiers effectivement chargés.

MY-PATHNAME (dir fil [suff])

Calcule le nom du fichier FIL, de suffixe SUFF, dans la directory DIR.
Si SUFF est NIL, il est pris par défaut ".ll" ou ".cp", suivant
l'indicateur #:YAFOOL:SYSTEME-COMPILE. Si DIR est NIL, la valeur de
#:YAFOOL:HOME-DIR ("home directory") lui est substituée. Si DIR est une
chaîne de longueur vide, c'est la directory courante.

REMOB-FILE lfn

LFN est une liste de fonctions: REMOB-FILE détruit (REMFN) les
définitions de fonctions de toutes les fonctions définies dans les
fichiers qui contenaient les diverses fonctions de LFN. Cette fonction
remet les autoloade préalablement définis par MY-AUTOLOAD ou FUN-
AUTOLOAD.

YAFOOL ()

Permet de recharger l'image mémoire YAFOOL, connu par la variable
#:YAFOOL:CORE: comme LELISP mais sans argument.

20.5.2. Fonctions de mapping.

Aux classique fonctions de mapping de Le_Lisp (`mapc`, `map`, `mapcan`, `maplist`, `mapcar`, `any` et `every`) ont été rajoutées:

`ANYL` (fct . liste) dmacro

`EVERYL` (fct . liste) dmacro

Sont à `ANY` et `EVERY` ce que `MAP` est à `MAPC`.

`SUBSET` (fct listel . liste) dmacro

Retourne le sous-ensemble de `LISTEL` pour lequel `FCT` retourne une valeur non `NIL`.

`SUBCAR` (fct . liste) dmacro

Comme `SUBSET`, mais ce sont les valeurs de `FCT` et non ses premiers arguments qui sont constituées en liste.

Pour ces 4 macros, si `FCT` est une λ -expression, l'expansion en tient compte et génère une seule λ -expression et non 2 imbriquées l'une dans l'autre. D'autre part, si `FCT` est une forme quotée, l'expansion se fait par la macro suivante (`Fxxx`) correspondant, en supprimant la quote.

```
(subset (lambda (u v) (> u v)) '(1 2 3 4)
        '(3 1 3 2))
```

s'expansse en

```
(mapcan (lambda (u v) (when (> u v) (list u))) '(1 2 3 4)
        '(3 1 3 2))
```

comme l'auraient fait

```
(subset '> '(1 2 3 4) '(3 1 3 2))
```

et

```
(fsubset > '(1 2 3 4) '(3 1 3 2))
```

Dans ces 2 derniers cas, l'expansion utiliserait évidemment des `GENSYM`, au lieu des variables `U` et `V`.

FANY	(fet . liste)	dmacro
FANYL	(fet . liste)	dmacro
FEVERY	(fet . liste)	dmacro
FEVERYL	(fet . liste)	dmacro
FMAP	(fet . liste)	dmacro
FMAPC	(fet . liste)	dmacro
FMAPCAN	(fet . liste)	dmacro
FMAPCAR	(fet . liste)	dmacro
FMAPLIST	(fet . liste)	dmacro
FSUBCAR	(fet . liste)	dmacro
FSUBSET	(fet listel . liste)	dmacro

Ces 11 macros sont identiques à leurs quasi-homonymes, sauf que fet n'est pas évalué: si FCT est une macro son expansion n'a lieu qu'une seule fois ce qui n'est pas le cas avec les fonctions classiques.

FIRST-WITH (fet listel . liste) dmacro

Cette macro est à FANY ce que SUBSET est à SUBCAR. Elle retourne le premier élément de LISTEL pour lequel FCT est vrai (non-NIL).

FMAP+ (fet . liste) dmacro

Fait la somme de toutes les valeurs non-NIL du mapping de FCT sur LISTE. Retourne NIL s'il n'y en a aucune. (fmap+ fet . liste) est donc équivalent à (apply '+ (fmapcar fet . liste)), si FCT a au moins une valeur non-NIL.

20.5.2.1. Syntaxe de ces macros.

Ces 17 macros ont un comportement particulier par rapport aux "constantes" du mapping, c'est-à-dire aux arguments de la fonction mappée qui ne changent pas pendant la durée du mapping:

- si un argument (non évalué) est NIL, il est pris comme argument de la fonction mappée;
- si l'argument (non évalué) est une liste circulaire (CIRLIST) d'un atome ou d'une forme quotée, ce dernier est pris comme argument de la fonction mappée;
- si la valeur d'un argument est une liste (non-NIL), il est pris comme argument du mapping, de la façon habituelle;
- si la valeur d'un argument est un atome (non-NIL), il est pris comme constante de la fonction mappée, et:
 - si l'argument (non évalué) est un atome ou une forme quotée, il est pris comme argument de la fonction mappée;
 - sinon la liste circulaire (CIRLIST) de cet argument est prise comme argument de la fonction de mapping.

On a ainsi l'expansion de:

```
(fmap fct '(1 2 3)
      'a
      (cirlist 'b)
      ()
      (print 'c)
      (cirlist (print 'd)))
on
(map (lambda (g123 g124 g125)
      (fct g123 'a 'b () g124 g125))
      '(1 2 3)
      (cirlist (print 'c))
      (cirlist (print 'd)))
```

N.B. Toutes les remarques du chapitre [13.4.4] sont évidemment valables pour la compilation de ces macros dont l'expansion dépend de l'évaluation des arguments.

20.6. Fonctions temporelles.

20.6.1. Fonctions temporelles générales.

ATEMPORAUX (objet)

Elle retourne la liste des "vraies" instances (récursivement) atemporelles de OBJET. C'est l'intersection de INSTANCIE-PAR de L'OBJET et de OBJET-ATEMPOREL.

CRE-INSTANCE ([[objet-atemporel] [instant]]) macro

Crée une nouvelle instance temporelle, de nom OBJET-ATEMPOREL-INSTANT, sous la facette INSTANT du lien instanciación de OBJET-ATEMPOREL, en tête de slot. Par défaut, TEMPS est pris pour INSTANT, et FRAME+ pour OBJET-ATEMPOREL.

DEL-INSTANCE (objet-atemporel instant) macro

Destruction de l'instance temporelle de facette INSTANT de OBJET-ATEMPOREL: cette destruction s'accompagne de l'effacement et de la correction de différents liens: est-un, instanciación, temporel et atemporel.

INSTANCE ([objet-atemporel [instant]]) macro

Avec un seul argument (FRAME+ est pris par défaut), retourne la dernière instance de OBJET-ATEMPOREL. Avec 2, c'est l'instance de facette INSTANT.

INSTANCES (objet-atemporel instant)

C'est la dernière instance précédant INSTANT (facette inférieure ou égale à INSTANT).

PREC-INSTANCE ([[objet] instant]) macro

Si INSTANT est présent, recherche l'instance temporelle de OBJET, précédant immédiatement celle de facette INSTANT (OBJET doit être atemporel).

Sans argument, FRAME+ est pris comme OBJET.

Sans INSTANT, si OBJET est atemporel, recherche de l'instance temporelle précédant celle de TEMPS. Si OBJET est temporel, il y a recherche de l'instance temporelle qui le précède.

Toutes ces recherches se font avec INSTANCES.

RETOUR-EN ([instant])

Provoque un retour du système dans l'état où il était avant INSTANT, par défaut 0, par appel de la méthode RETOUR-T sur tous les objets temporels du système: (INSTANCIE-PAR OBJET-ATEMPOREL)..

TEMPORAUX (objet)

C'est la version temporelle de ATEMPORAUX: elle retourne la liste des dernières instances (récursive) temporelles de OBJET.
(temporaux objet) est toujours équivalent à
(temporaux instance (atemporaux objet)).

20.6.2. Fonctions d'édition de l'historique.

HISTOIRE (objet-atemporel [ind])

Cette fonction édite l'historique de OBJET-ATEMPOREL, c'est-à-dire l'ensemble de ses instances temporelles: IND est un indicateur qui, s'il vaut NIL, entraîne une édition minimale, seuls les attributs ayant changé étant édités.

EDITION-INSTANT (instant [univ])

Edite une "coupe" temporelle en INSTANT, c'est-à-dire l'ensemble des instances temporelles de tous les objets atemporels de l'univers UNIV (par défaut \$:YAFUOL:IDEAL), précédant immédiatement INSTANT.

20.6.3. Fonction de recherche dans l'historique.

CHERCHE-QUAND (predicat . liste-de-frames)

Cette fonction permet la recherche, à partir de la fonction LISP PREDICAT et de la liste d'objets LISTE-DE-FRAMES, du dernier instant auquel la fonction appliquée aux instances temporelles de chacun des objets de la liste était vraie, c'est à dire avait une valeur différente de NIL.

La fonction LISP peut être donnée sous la forme d'une λ -expression ou simplement par son nom. Ce dernier cas est surtout intéressant pour les fonctions associées au nom de chaque attribut, qui permettant une recherche simple du dernier instant où l'attribut était vrai.

La valeur renvoyée par la fonction CHERCHE-QUAND est l'instant correspondant, exprimé en quantum de temps, ou NIL s'il n'y a pas de solution.

Les fonctions suivantes, à la syntaxe plus ergonomique, complètent cette primitive générale.

QUAND (slot frame)

Cette fonction permet la recherche, à partir d'un attribut et d'un objet, du dernier instant auquel l'attribut était vrai, c'est à dire avait une valeur différente de NIL, sur l'instance temporelle de l'objet.

Cette fonction est donc équivalente à CHERCHE-QUAND pour les attributs, mais est plus rapide, car elle utilise systématiquement la primitive d'accès FGET au lieu de celle associée à l'attribut. Elle effectue donc une recherche directe, sans déclenchement de réflexes.

QUAND-NON (slot frame)

Cette fonction permet, à partir d'un attribut et d'un objet, de rechercher le dernier instant auquel l'attribut était faux, c'est à dire avait pour valeur NIL, sur l'instance temporelle de l'objet.

4/7/86

QUAD-CHASE

(alet frame)

Cette fonction permet la recherche, à partir d'un attribut et d'un objet, du dernier instant auquel la valeur de l'attribut a changé entre une instance temporelle de l'objet et la suivante.

QUAD=

(alet frame1 frame2)

Cette fonction permet, à partir d'un attribut et de deux objets, de rechercher le dernier instant auquel l'attribut avait la même valeur sur les instances temporelles de chacun des 2 objets.

20.7. Fonctions de video.

VIDEO-BEEP (Evideo) [n])

Fait klaxonner N (par défaut 1) fois le terminal (§:VIDEO:OUTCHAN) de l'écran VIDEO (par défaut la §:VIDEO:COURANTE de l'univers courant, §:YAFOOL:IDEAL).

VIDEO-CHAMP (video string x y [long])

Affiche sur l'écran VIDEO la chaîne STRING aux coordonnées X et Y (valeurs entières), sur une longueur LONG. L'affichage réel n'a lieu que si §:VIDEO:VIDEOP est positionné pour VIDEO.

VIDEO-CLIP (video objet x y)

Affiche sur l'écran VIDEO le caractère §:VIDEO:CAR de OBJET, aux coordonnées absolues (indépendantes du repère) des valeurs des attributs X et Y de OBJET. L'affichage réel n'a lieu que si §:VIDEO:VIDEOP est positionné pour VIDEO, et si les coordonnées figurent bien à l'intérieur du repère.

VIDEO-CLIP-REINIT (video objet x y)

Comme VIDEO-CLIP, mais le point est mémorisé dans §:VIDEO:RE-INIT de VIDEO, pour réaffichage à chaque pas de temps.

VIDEO-COPIE (video)

Provoque l'édition de l'état courant de l'écran dans le fichier de canal §:YAFOOL:FICHER-FIGURE.

VIDEO-IN (Out])

Avec argument provoque l'ouverture des fichiers de nom OUT et de suffixe ".trace" et ".fig", de canal respectif §:YAFOOL:FICHER-TRACE et §:YAFOOL:FICHER-FIGURE. §:YAFOOL:TRACE, qui devait être NIL, reçoit la valeur du premier.

VIDEO-OUT ()

Remet à NIL §:YAFOOL:TRACE.

VIDEO-X (Evideo) [min max])

Permet de consulter la fourchette d'abscisse du repère de VIDEO (par défaut §:VIDEO:COURANTE de l'univers courant, §:YAFOOL:IDEAL) en cas d'absence des autres arguments, ou de la modifier s'ils sont présents.

VIDEO-Y (Evideo) [min max])

Comme VIDEO-X pour l'ordonnée du repère.

20.8. Boite à outil.

Ce paragraphe présente quelques attachements procéduraux généraux, en particulier des comportements de création d'univers [#12.3.5] et des réflexes associés aux relations inverses [#14.2.2.2]. La syntaxe de présentation est modifiée: plus d'arguments, inutiles, mais pour les méthodes le nom générique de la méthode, et pour les réflexes il est indiqué si ce sont des macros qui doivent alors être appelées par un réflexe.

20.8.1. Comportements.

METH-NIL

comportement

C'est le comportement "nul", à utiliser pour tout comportement, quelque soit son nombre d'arguments: retourne NIL, sans effet de bord.

METH-SLOT+REL

creer-slot

C'est la méthode CREER-SLOT pour déclencher les réflexes si-ajout sur la valeur des facettes value et sauf du slot (c'est une relation) de l'objet considéré.

METH-SLOT+VALIE

creer-slot

Comme la précédente, mais seulement pour la facette value.

20.8.2. Réflexes de relation.

Ils ont presque tous pour rôle d'établir l'équivalence des liens inverses, et se différencient par la multiplicité des 2 liens considérés. La présence d'un "+" indique un si-ajout et sinon c'est un si-enleve.

DEMON+DES-LIENS

si-ajout macro

DEMON-DES-LIENS

si-enleve macro

Pour les relations multiples dont l'inverse est multiple.

DEMON+DES-LIENS1

si-ajout macro

DEMON-DES-LIENS1

si-enleve macro

Pour les relations uniques dont l'inverse est unique.

DEMON+DES-LIENS12

si-ajout macro

DEMON-DES-LIENS12

si-enleve macro

Pour les relations uniques dont l'inverse est multiple.

DEMON+DES-LIENS21

si-ajout macro

DEMON-DES-LIENS21

si-enleve macro

Pour les relations multiples dont l'inverse est unique.

DEMON-AUTO-INVERSE

si-ajout

Pour les relations symétriques.

20.9. Le package #:YAFDOL

Il contient un certain nombre de variables globales qui donnent diverses informations sur l'univers en cours de création, ou dernier créé, ainsi que sur l'état général du système, ainsi que des canaux de sortie en cas d'utilisation de la video.

#:YAFDOL:COMPILE-FLAG

variable

Drapeau indiquant, pour l'expansion des macros (mapping ou applicateurs), si le système est en cours de compilation ou non.

#:YAFDOL:CORE

variable

Contient le nom du fichier, de suffixe #:LELISP:CORE-EXTENSION (en général ".core"), de l'image mémoire YAFDOL, dans la directory #:YAFDOL:YAF-DIR.

#:YAFDOL:CREATION

variable

Drapeau qui vaut NIL en dehors de la phase de création, c'est-à-dire hors de DEBUT-D-UNIVERS.

#:YAFDOL:DUAL-IDEAL

variable

Nom de la racine duale de l'univers en cours de création.

#:YAFDOL:EST-UN

variable

Initialisé à est-un, c'est le lien de hiérarchie d'héritage, que l'on peut donc modifier.

#:YAFDOL:FICHER-FIGURE

variable

C'est le canal d'édition des figures (voir la fonction VIDEO-COPIE).

#:YAFDOL:FICHER-TRACE

variable

C'est le canal de sortie standard (PRINT) en cas d'utilisation de la video.

#:YAFDOL:FN-FILES

variable

Contient la liste des fonctions d'univers à éventuellement décharger, une par fichier.

#:YAFDOL:FRAME

variable

C'est le nom de la frame en cours de création: les fonctions attachées à des slots y ont ainsi accès.

#:YAFDOL:GENSYM-COUNTER

variable

#:YAFDOL:GENSYM-STRINGS

variable

Contiennent les valeurs des chaînes de caractères et compteur de FGENSYM.

`@:YAFOOL:IDEAL`

variable

Nom de l'univers en cours de création, ou du dernier univers créé.

`@:YAFOOL:LOADED-FROM-FILE`

slot

Indique pour tout objet, éventuellement par défaut, le fichier d'où il a été chargé. Le nom de ce fichier est obtenu par la variable globale `@:SYSTEM:LOADED-FROM-FILE` qui est gérée par la fonction Le_Lisp `LOAD`.

`@:YAFOOL:NEUS`

variable

Liste des "concepts" (modèles ou duaux) nouveaux de l'univers en cours de création.

`@:YAFOOL:PREC`

variable

Nom de l'univers précédent, dont l'univers en cours est un objet.

`@:YAFOOL:REPERE`

variable

C'est le repère par défaut de l'écran `@:YAFOOL:VIDEO`.

`@:YAFOOL:VIDEO`

variable

C'est l'écran par défaut, correspondant au terminal (canal ()), défini avec le chargement de la video.

`@:YAFOOL:SPECIAL-DEBUT`

variable

`@:YAFOOL:SPECIAL-FIN`

variable

Ces 2 variables contiennent des listes de formes à évaluer en prologue ou épilogue de toute création d'univers. Initialisées à NIL, elles sont incrémentées par tout chargement d'univers (`BIG-BANG` ou `INSERT-UNIVERS`) qui le nécessite (essentiellement les autoload), et remises à NIL par `REMOB-FILE-UNIV`.

`@:YAFOOL:SYSTEME-COMPILE`

variable

Drapeau indiquant si le système YAFOOL est compilé ou non.

`@:YAFOOL:TRACE`

variable

C'est le canal de sortie standard (`PRINT`) à tout instant: il vaut NIL ou `@:YAFOOL:FICHIER-TRACE`.

21. UTILISATION ET CHARGEMENT.

21.1. Caractéristiques de la réalisation.

YAFOOL occupe, en configuration maximale — tous les autoload chargés — de l'ordre de 10 K-CONS, 500 symboles et 40 K-octets de zone code.

Ecrit en Le_Lisp, il est disponible sur toute machine sur laquelle ce dernier a été porté. Mais il est illusoire de vouloir s'en servir sur une machine de taille mémoire insuffisante. A titre d'exemple, une version minimale a été portée sur Macintosh 512 K, ce qui le sature à peu près, alors que le MacPlus (1 M-octets) autorise un travail raisonnable.

Mais 800 K-octets pour l'image mémoire Le_Lisp semble être un minimum de confort: le paramètre d'appel de yafeel doit alors être 7 ou 8.

21.2. Installation (UNIX).

La cartouche de livraison doit être copiée dans /usr/local/yafeel. L'image mémoire YAFOOL est alors générée par l'appel de la commande lelisp.coryafeel. La commande yafeel doit être copiée dans /usr/bin. La commande INSTALL, appelée sur la cartouche réalise l'ensemble de cette installation.

M.B. La zone code de l'image mémoire Le_Lisp doit être, pour YAFOOL, supérieure au standard adopté généralement: une reconfiguration est alors nécessaire au moment de l'installation.

21.3. , Chargement du système.

Il s'effectue par chargement d'une image mémoire Le_Lisp. D'une façon générale, sous un système UNIX, une commande yafeel est fournie: son unique argument, numérique représente la taille de la zone CONS de l'image mémoire Le_Lisp à charger. Par défaut c'est 4.

Cette commande a la signification:
lelisp 01 -r /usr/local/yafeel/yafeel.cora.

A la fin de ce chargement le système charge le premier fichier .yafeel trouvé, soit dans le directorie courante, soit dans le "home directory" de l'utilisateur.

On est alors sous le TOPLEVEL de l'interprète Le_Lisp, avec le noyau de YAFOOL chargé.

4/7/86

Y A F O O L 2.1

22. PERSPECTIVES.

Ce chapitre dévoile brièvement, en forme de programme, les (futures) nouveautés de la version 3, dont on peut prévoir une première apparition (partielle) pour la fin de l'année 1986.

22.1. YAFOOL et la programmation logique.

C'est évidemment le plus gros morceau: moteur d'inférence en logique d'ordre 1, avec chaînage avant et / ou arrière. Sa conception sera très modulaire et extensible: il sera écrit en YAFOOL, pour une base de faits YAFOOL.

22.2. Nouvelle version des applicateurs.

Ils seront simplifiés, avec la disparition des réflexes et l'implémentation de véritables comportements à 2 dimensions.

La perte d'un mécanisme général de réflexe sera compensé par ces comportements beaucoup plus maniables. L'attribut reflex-applic sera peut-être conservé. En son absence, c'est les super-comportements qui devraient tenir son rôle.

22.3. Réflexes a priori: si-possible.

Des réflexes a priori d'écriture sont prévus, ainsi que toutes les extensions de facettes de contrainte, et bien sûr des comportements ou réflexes de gestion des erreurs.

22.4. Environnement de programmation.

22.4.1. Aide à la mise au point.

Il est constitué par celui de Le Lisp (très rudimentaire) et de quelques fonctionnalités YAFOOL comme trace-slot. Il doit être étendu, en utilisant les possibilités de multi-fenêtrage, en particulier pour la définition des objets.

22.4.2. Multi-fenêtrage et graphique.

Développements basés sur les primitives de multi-fenêtrage de Le Lisp, version 15.2, ainsi que sur l'embryon de video présenté dans YAFOOL 2.1.

22.4.3. Editeur d'objets.

La phase de création d'univers en YAFOOL manque d'interactivité. En particulier, sa complexité oblige à tout recommencer à zéro, en cas d'erreur. Il est envisagé de développer un éditeur d'objets qui permettrait une définition plus interactive.

22.4.4. Editeur de règles.

Le point précédent et l'intégration de la programmation logique amèneront naturellement à spécialiser cet éditeur pour les règles.

22.4.5. Gestion de bases d'objets et de règles.

Il s'agit de permettre la maintenance de grosses bases de connaissances, dans des versions multiples (suivant l'instant et les hommes).

22.5. Le_Lisp, version 15.2.

La version de YAFOOL décrite ici est celle de la version 15.1 de Le_Lisp. Le portage sur la version 15.2 est en cours. Il se traduira notamment par de meilleures performances, grâce au nouveau compilateur, et par l'amélioration de certains traits du langage, notamment les continuations et retardements.

La compatibilité avec la version 15.1 devrait être totale.

22.6. Le_Lisp et Common Lisp.

Outre le suivi de l'évolution de Le_Lisp — version 15.2, éventuelle convergence (européenne ?) vers Common Lisp etc., —, il est envisagé de porter YAFOOL en Common Lisp. Des modifications d'implémentation sont prévisibles.

4/7/86

Y A F O O L 2.1

23. BIBLIOGRAPHIE.

- [Albert 83] Albert P.: "KODL: représentation des connaissances", BIGRE No 37, Octobre 1983.
- [Bigre 86] Actes des Journées Afcet-Informatique Langages Orientés Objets, BIGRE+GLOBULE No 48.
- [Bobrow 83] Bobrow D. S. et Stefik M.: "The LOOPS Manual", Xerox Corporation, Décembre 1983.
- [Bourgault 83] Bourgault S., Dinobas M. et Le Pape J.P.: "Manuel LISLOG", CNET 1983.
- [Brachman 83] Brachman R.-J.: "What IS-A is and isn't: an analysis of taxonomic links in semantic networks". Computer, Vol 16 No 10, Octobre 1983, pp 30-36.
- [Carre 84] Carre F. et Salle P.: "Acteurs et logique", Congrès Reconnaissance des formes et Intelligence Artificielle, AFCET, Janvier 1984, pp313-319.
- [Chailloux 85] Chailloux J.: "Le-Lisp Version 15 Manuel de référence", INRIA, Février 1985.
- [Chailloux 86] Chailloux J.: "La machine LLM3" INRIA, janvier 1986.
- [Chouraqui 81] Chouraqui E.: "Contribution à l'étude théorique de la représentation des connaissances. Le système symbolique ARCHES". Thèse d'Etat. Institut national polytechnique de Lorraine (1981).
- [Cointe 83] Cointe P.: "Comprendre SMALLTALK, Penser SMALLTALK", BIGRE No 37, Décembre 1983.
- [Ducournau 86] Ducournau R.: "MAX: Système expert et Langage Objet pour la Simulation et le Maquettage" ENSIMAG, Grenoble 1986.
- [Durieux 83] Durieux J.L., Julian D.: "Programmation par acteurs" Journées Langages Objets 1983, Bigre No 37.
- [Ferber 83] Ferber J.: "MERINF, langage d'acteur pour la représentation et la manipulation des connaissances". Thèse de Docteur-Ingénieur, Université Paris-VI (1983).
- [Goldberg 83] Goldberg A. et Robson D.: "SMALLTALK-80 the language and its implementation" Addison-Wesley (1983).
- [Gosciny] Gosciny et Uderzo: "Asterix" Dargaud.
- [Habib 86] Bouchitte V., Habib M., Hamroun M. et Jegou P.: "Depth-first search and linear extensions" Laboratoire d'informatique de Brest, Février 1986.
- [Hewitt 73] Hewitt C.: "An universal modular actor formalism for artificial intelligence", Actes du congrès IJCAI 1973 pp. 235-245.
- [Hullot 85] Hullot J.-M.: "CEYX - Version 15", Rapports techniques No 44, 45 et 46, INRIA, Février 1985.

- [Jakobson] Jakobson R.: "Essais de linguistique générale", Seuil.
- [Liebermann 85] Liebermann H.: "Object Oriented Programming Languages", Entry for the Encyclopaedia of AI, S. Shapiro, ed., Wiley 1985.
- [Liebermann 86] Liebermann H.: "Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object-Oriented Systems" Actes des journées Afcet-Informatique Langages Orientés Objets, BIGRE+GLOBULE No 48, 1986.
- [Minsky 75] Minsky M.: A framework for representing knowledge, in: P. Winston (Ed), The Psychology of Computer Vision (McGraw-Hill, New-York, 1975) 211-277.
- [Moon 81] Moon D., Weinreb D.: "Flavors: Message Passing in the Lisp Machine" MIT AI lab. A.I. memo No 602 (1980) et Lisp Machine Manual.
- [Reiter 81] Reiter R. et Criscuolo G.: "On interacting defaults". Proc. IJCAI-81, Août 81, pp 270-276.
- [Reiter 83] Reiter R. et Etherington D.-W.: "On inheritance hierarchies with exceptions". Proc. AAAI-83, Août 1983, pp 104-108.
- [Roche 84] Roche Ch.: "EAGUE-LRO Génération de systèmes experts - Applications à des problèmes d'ordonnements". Thèse de 3ème cycle, Université de Savoie, Chambéry (1984).
- [Shakespeare] dont on sait que ce n'est pas lui qui a écrit son oeuvre, mais un homonyme.
- [Touretzky 81] Fahman S.E., Touretzky D.S. et van Roggen W.: "Cancellation in a parallel semantic network". Proc. IJCAI-81, Août 81, pp 257-263.
- [Touretzky 84] Touretzky D.S.: "Implicit ordering of defaults in inheritance systems". Proc. AAAI-84 pp 322-325.
- [Winograd 75] Winograd T.: "Frame representation and the declarative-procedural controversy". In "Representation and understanding: Studies in cognitive science". Academic Press, New-York (1975).
- [Winston 84] Winston P.-H. et Horn B.K.P.: "LISP" Addison-Wesley (1984).
- [Wright 84] Wright J.M., Fox M.S. et Adam D.L.: "SRL2 User Manual". Carnegie-Mellon University, Pittsburgh (1984).

PREMIERE PARTIE: LE LANGAGE.

- 1. YAFOOL: "YET ANOTHER FRAME-BASED OBJECT-ORIENTED LANGUAGE". p. 5
 - 1.1. Quatre paradigmes de programmation. p. 5
 - 1.1.1. Procédures.
 - 1.1.2. Objets.
 - 1.1.3. Données.
 - 1.1.4. Règles.
 - 1.2. YAFOOL = langage orienté objets et données (LOOD). p. 6
 - 1.2.1. Objets.
 - 1.2.2. Frames.
 - 1.2.3. Frames de YAFOOL.
 - 1.2.4. Règles.
 - 1.2.5. Procédures.
- 2. LES CONCEPTS DU LANGAGE. p. 8
 - 2.1. Organisation du langage. p. 8
 - 2.1.1. Noyau "dur".
 - 2.1.2. Noyau "mou".
 - 2.1.3. Interactions noyau dur / noyau mou.
 - 2.2. Utilisation de YAFOOL. p. 8
 - 2.2.1. Niveaux utilisateurs.
 - 2.2.2. Documentation YAFOOL.
 - 2.3. Les concepts de YAFOOL. p. 9
 - 2.3.1. Principe d'implémentation.
 - 2.3.2. Principe de dualisation.
 - 2.3.3. Principaux objets duaux.
 - 2.3.3.1. Attribut.
 - 2.3.3.2. Comportement.
 - 2.3.3.3. Réflexe.
 - 2.3.3.4. Facette.
 - 2.3.4. Héritage.
 - 2.3.5. Autres concepts.
 - 2.3.5.1. Continuations.
 - 2.3.5.2. Retardements.
- 3. L'HERITAGE, LES LIENS ET LA HIERARCHIE. p. 12
 - 3.1. Interprétation. p. 12
 - 3.1.1. Interprétation ensembliste.
 - 3.1.2. YAFOOL: interprétation conceptuelle.
 - 3.1.3. Lien est-un et instanciation.
 - 3.1.4. Héritage et autres liens.
 - 3.1.5. Liens et réseau sémantique.
 - 3.2. Héritage et délégation. p. 14
 - 3.3. La multiplicité de l'héritage. p. 14

- 3.4. Les exceptions à l'héritage: la facette sauf. p. 14
- 3.5. Recherche dans la hiérarchie. p. 15
 - 3.5.1. Algorithme de recherche. p. 15
 - 3.5.1.1. Algorithmes naïfs.
 - 3.5.1.2. L'algorithme de YAFOOL.
 - 3.5.2. Ordres partiels et totaux, extension linéaire.
 - 3.5.3. Développements.
 - 3.5.3.1. La multiplicité n'est pas un ordre.
 - 3.5.3.2. Contre exemple.
 - 3.5.4. Les exceptions.
 - 3.5.5. La facette Herit.
 - 3.5.6. L'environnement d'appel
 - 3.5.7. Hiérarchies définies par d'autres liens.

DEUXIEME PARTIE: LE NOYAU DUR.

- 4. PRINCIPES GENERAUX D'IMPLEMENTATION.
 - 4.1. Liste d'association. p. 23
 - 4.1.1. Valeur objet. p. 23
 - 4.1.2. Remise du slot en tête.
 - 4.2. Marquage. p. 23
 - 4.2.1. Des objets. p. 23
 - 4.2.2. Des slots.
 - 4.3. Multiplicité et atomicité des valeurs. p. 23
 - 4.4. Dualisation des réflexes. p. 24
 - 4.4.1. Répartition des accès entre frames et slots. p. 24
 - 4.4.2. Accès unique au slot.
 - 4.5. Valeur NIL.
 - 4.6. Les primitives. p. 24
 - 4.6.1. Fonctions décrites. p. 24
 - 4.6.2. Convention de nom.
 - 4.6.2.1. Convention de la facette standard.
 - 4.6.2.2. Mode d'accès.
- 5. LES ACCES EN LECTURE.
 - 5.1. Lecture simple. p. 26
 - 5.1.1. Recherche dans la hiérarchie. p. 26
 - 5.1.2. Les accès standards.
 - 5.2. Les accès multiples. p. 26
 - 5.2.1. Fermeture transitive des liens.
 - 5.2.2. Collecte sur toute la hiérarchie.

- 5.3. La facette défaut. p. 27
- 5.4. Activation des réflexes si-besoin. p. 27
 - 5.4.1. Affectation.
 - 5.4.2. Réflexes si-ajout.
 - 5.4.3. Contrôle de boucle.
 - 5.4.4. Vérification a posteriori.
- 5.5. Primitives de vérification. p. 28
- 6. ACCES EN ECRITURE. p. 30
 - 6.1. Création, affectation et ajout. p. 30
 - 6.2. Accès en retrait ou effacement. p. 31
 - 6.3. Déclenchement des réflexes. p. 32
 - 6.3.1. Evaluation des réflexes.
 - 6.3.2. VALUE+: la valeur dans les réflexes d'écriture.
 - 6.3.3. Ordre d'évaluation.
 - 6.4. Cas particuliers des liens. p. 32
 - 6.4.1. L'affectation des liens.
 - 6.4.2. Cas particulier du lien est-un.
 - 6.5. Réflexes si-possible. p. 33
 - 6.6. Ecriture et héritage. p. 33
- 7. LES COMPORTEMENTS. p. 34
 - 7.1. Généralités. p. 34
 - 7.1.1. Nom générique et valeurs fonctionnelles.
 - 7.1.2. Arguments.
 - 7.1.3. Environnement d'appel.
 - 7.2. Activation des méthodes. p. 34
 - 7.2.1. Absence de méthode.
 - 7.2.2. Comportement de masque.
 - 7.2.3. Simulation par un SELECTQ.
 - 7.3. Activation des applicateurs. p. 35
 - 7.3.1. Activation des réflexes.
 - 7.3.1.1. Héritage à 2 dimensions.
 - 7.3.1.2. Arguments.
 - 7.3.1.3. Rôle de VALUE+
 - 7.3.2. Exemples.
 - 7.3.3. Simulation par un SELECTQ.
 - 7.3.4. Asymétrie des applicateurs.
 - 7.4. Héritage biaisé. p. 36
 - 7.4.1. Super-comportements.
 - 7.4.2. Environnement.
 - 7.4.3. Réflexes

8. CONTROLE D'ENVIRONNEMENT ET COMMUNICATION.

8.1. Liaison dynamique d'objets. p. 40

8.2. Liaison dynamique de valeurs. p. 40

8.3. Continuations et retardements. p. 40

 8.3.1. Le principe "officiel" de la continuation. p. 41

 8.3.2. Continuation locale.

 8.3.2.1. Continuation simple.

 8.3.2.2. Pseudo-continuation.

 8.3.3. Retardements.

 8.3.4. Mécanisme général des continuations.

 8.3.4.1. Structure d'un "tag-continuation".

 8.3.4.2. Rôle de VALUE+.

 8.3.4.3. Ordre des évaluations.

 8.3.4.4. Liaisons des)-variables.

 8.3.5. Continuations en cascade.

 8.3.5.1. Cascade de retardements.

 8.3.5.2. Cascade avec continuation locale.

 8.3.5.3. Cascades avec liaisons de)-variables.

 8.3.6. Les continuations dialogantes.

 8.3.6.1. Continuations emboîtées.

 8.3.6.2. Continuation dialogante.

 8.3.6.3. Exemples.

 8.3.7. Continuations globales.

9. AUTRES PRIMITIVES.

..... p. 50

TROISIEME PARTIE: LE NOYAU MOU.

10. LE NOYAU MOU.

10.1. Principes. p. 53

10.2. Résultats. p. 53

11. LE DUAL.

11.1. Généralités. p. 54

 11.1.1. Obligation de consistance. p. 54

 11.1.2. Facette-standard.

11.2. Attributs.

 11.2.1. Attribut1 / attribut2. p. 54

 11.2.2. Relation.

 11.2.2.1. Relation1 / relation2.

 11.2.2.2. Exemples.

11.3. Comportement.

 11.3.1. Méthodes. p. 55

 11.3.2. Applicateurs.

11.4. Réflexes. p. 55

11.5. Facettes.	p. 56
11.5.1. Facettes et slots standards.	
11.6. Slot-autoload.	p. 56
12. UNIVERS.	p. 57
12.1. Les modèles.	p. 57
12.2. L'objet univers.	p. 57
12.3. La création d'univers.	p. 57
12.3.1. Passe 1: initialisation des objets.	
12.3.2. Passe 2: init-slot et init-facet.	
12.3.3. Passe 3: caract.	
12.3.4. Passe 4: creer-slot et creer-facet.	
12.3.5. Les méthodes de création d'univers.	
12.3.5.1. Init-slot.	
12.3.5.2. Init-facet.	
12.3.5.3. Caract.	
12.3.5.4. Creer-slot.	
12.3.5.5. Creer-facet.	
12.4. Les CLES comme des MOTS-CLES.	p. 61
12.5. Insertion dans un univers.	p. 61
12.6. La hiérarchie des univers.	p. 62
12.6.1. Univers-contenus et univers-contenant.	
13. MACROS ET APPLICATEURS D'ACCES.	p. 63
13.1. Généralités.	p. 63
13.2. Macros.	p. 63
13.3. Obligation de consistance.	p. 63
13.4. Principes d'implémentation.	p. 63
13.4.1. Slots, macros et applicateurs.	
13.4.2. Macro-caractère ":".	
13.4.3. Syntaxe générale.	
13.4.3.1. Ordre d'évaluation.	
13.4.4. Expansion avec évaluation.	
13.4.4.1. Valeur déplacée et valeur retournée.	
13.4.4.2. Problèmes de compilation.	
13.4.5. Elision de FRAME+.	
13.5. Les réflexes des :applicateurs.	p. 65
13.6. Les différents :applicateurs.	p. 65
13.6.1. La macro générale.	
13.6.2. Les macros des :applicateurs.	
13.6.3. Le :applicateur ::, lecture et déclenchement.	
13.6.3.1. Lecture des attributs.	
13.6.3.2. Déclenchement des comportements.	

- 13.6.4. Le :applicateur := , affectation.
- 13.6.5. Le :applicateur :- , retrait et effacement.
- 13.6.6. Le :applicateur + , ajout.
- 13.6.7. Le :applicateur ? , vérification.
- 13.6.8. Le :applicateur < , fermeture transitive des liens.
- 13.6.9. Les :applicateur :> et :>> , super-comportement.
- 13.7. Pseudo :applicateurs :/ , :% et :?? .
- 13.8. Accès à des facettes: le slot standard. p. 67
- 14. L'UNIVERS NOYAU. p. 67
- 14.1. Objet-ideal. p. 68
- 14.2. Le dual. p. 68
 - 14.2.1. Les réflexes. p. 69
 - 14.2.2. Les attributs.
 - 14.2.2.1. Multiplicité des attributs.
 - 14.2.2.2. Relation.
 - 14.2.2.3. Type-de-recherche.
 - 14.2.3. Comportements.
 - 14.2.4. Facettes.
- 14.3. Univers. p. 73
- QUATRIEME PARTIE: LES EXTENSIONS.
- 15. LES EXTENSIONS.
 - 15.1. Extension de l'héritage. p. 77
 - 15.1.1. La facette LIKE. p. 77
 - 15.1.2. Les facettes INIT et INIT-EVAL.
 - 15.1.3. Les facettes HERIT-FROM et HERIT-BY.
 - 15.1.4. La facette VALUE-OF.
 - 15.2. Commentaires. p. 79
 - 15.3. Constantes. p. 79
 - 15.4. Création et destruction d'instances. p. 80
 - 15.4.1. Méthode création.
 - 15.4.2. Méthode efface.
 - 15.5. Les autoloads. p. 81
 - 15.5.1. 2 catégories d'autoloads.
 - 15.5.1.1. Slot autoload.
 - 15.5.1.2. Objet autoload.
 - 15.5.2. Fichiers d'autoload.
 - 15.5.3. Remise en autoload.
 - 15.6. Booléen. p. 82
 - 15.7. Domain. p. 83

15.8. Valeur et range.	p. 83
15.9. Trace de slots.	p. 84
15.9.1. Trace des attributs.	
15.9.2. Trace des facettes.	
15.9.3. Trace des comportements.	
15.10. Edition.	p. 85
16. UNIVERS TEMPORELS.	p. 86
16.1. Objets temporels et atemporels.	p. 86
16.2. Univers temporel.	p. 88
16.2.1. Liens et instances temporels.	
16.2.2. Liens temporels.	
16.2.2.1. Liens (a)temporels et est-un.	
16.2.3. Modes d'accès particuliers en lecture.	
16.2.4. Instances temporelles.	
16.3. Booléens temporels.	p. 90
16.3.1. Booléen-t.	
16.3.2. Accès particulier.	
16.3.3. Liste-bool.	
16.4. Applications aux applicateurs.	p. 91
16.4.1. applicateurs sur objets temporels.	
16.4.1.1. Lecture.	
16.4.1.2. Ecriture.	
16.4.2. applicateur d'écriture booléen	
16.4.3. Programmation par métonymie.	
16.5. Comportements temporels.	p. 92
16.5.1. Création.	
16.5.1.1. Création d'instances temporelles.	
16.5.1.2. Création d'instance atemporelles.	
16.5.1.3. Effacement d'instance temporelles.	
16.5.2. Retour en arrière.	
16.5.3. Nettoyage d'historique.	
16.6. Constantes temporelles.	p. 93
16.7. Qt et durée.	p. 94
16.8. Fonctions temporelles.	p. 94
16.8.1. Fonctions de manipulation d'instances temporelles.	
16.8.2. Recherche des instances (a)temporelles d'un objet.	
16.8.3. Fonctions d'historique.	
17. L'EXTENSION VIDEO.	p. 95
17.1. Les écrans.	p. 95
17.1.1. Les coordonnées.	
17.1.2. Ecriture.	

- 17.1.3. Prologue et épilogue.
 - 17.1.3.1. Prologue.
 - 17.1.3.2. Epilogue.
- 17.1.4. Ecran par défaut.
- 17.2. Les champs.
 - 17.2.1. Syntaxe de #:VIDEO:CHAMP. p. 97
 - 17.2.2. Coordonnées par défaut.
 - 17.2.3. Attributs complémentaires.
- 17.3. Les points.
 - 17.3.1. Les repères. p. 98
 - 17.3.1.1. Les axes.
 - 17.3.1.2. Repère courant.
 - 17.3.2. Trace des déplacements.
 - 17.3.3. Mémorisation des points affichés.
 - 17.3.3.1. Changement de repère.
- 17.4. Activation et désactivation.
- 17.5. Figure. p. 102
- 18. EXEMPLE: OBJETS COMPOSITES. p. 102
 - 18.1. Objets composites. p. 103
 - 18.2. Exemple. p. 103
 - 18.3. Méthodes d'univers. p. 103
 - 18.4. Utilitaires. p. 104
 - 18.5. Classes, instances, parties et héritage. p. 105
 - 18.6. Les fichiers de l'autoload. p. 105
 - p. 106
- CINQUIÈME PARTIE: MANUEL D'UTILISATION.
- 19. PROGRAMMER EN YAFDOL.
 - 19.1. Un univers d'objets. p. 109
 - 19.2. Des lieux et des personnages. p. 109
 - 19.2.1. Instances et accès. p. 109
 - 19.2.2. Comportement des personnages.
 - 19.2.3. Classes de personnages et instances de lieux.
 - 19.3. Les gaulois.
 - 19.3.1. Les comportements des gaulois. p. 113
 - 19.3.2. Les druides.
 - 19.4. Les irréductibles.
 - 19.4.1. Le chef (courageux et ombrageux) et le druide. p. 113
 - 19.4.2. Des individualités assez fortes.

19.5. Les romains et gallo-romains.	p. 120
19.6. Les lieux.	p. 122
19.7. De l'art de se faire des politesses.	p. 123
19.8. Les comparses.	p. 124
19.9. Exemple de session.	p. 124

ANNEXES

20. TOUTES LES FONCTIONS.	p. 143
20.1. Syntaxe.	p. 143
20.1.1. Facette standard.	
20.1.2. Plan.	
20.2. Primitives des frames.	p. 144
20.3. Structures de contrôle.	p. 150
20.3.1. Le package #:CONTINU	
20.4. Primitives d'univers.	p. 153
20.5. Fonctions diverses.	p. 154
20.5.1. Utilitaires de chargement et compilation.	
20.5.2. Fonctions de mapping.	
20.5.2.1. Syntaxe de ces macros.	
20.6. Fonctions temporelles.	p. 161
20.6.1. Fonctions temporelles générales.	
20.6.2. Fonctions d'édition de l'historique.	
20.6.3. Fonction de recherche dans l'historique.	
20.7. Fonctions de video.	p. 164
20.8. Boite à outil.	p. 165
20.8.1. Comportements.	
20.8.2. Réflexes de relation.	
20.9. Le package #:YAFDOL	p. 166
21. UTILISATION ET CHARGEMENT.	p. 168
21.1. Caractéristiques de la réalisation.	p. 168
21.2. Installation (UNIX).	p. 168
21.3. Chargement du système.	p. 168

22. PERSPECTIVES.

22.1. YAFOOL et la programmation logique. p. 169

22.2. Nouvelle version des applicateurs. p. 169

22.3. Réflexes a priori: si-possible. p. 169

22.4. Environnement de programmation. p. 169

 22.4.1. Aide à la mise au point. p. 169

 22.4.2. Multi-fenêtrage et graphique.

 22.4.3. Editeur d'objets.

 22.4.4. Editeur de règles.

 22.4.5. Gestion de bases d'objets et de règles.

22.5. Le_Lisp, version 15.2.

22.6. Le_Lisp et Common Lisp. p. 170

23. BIBLIOGRAPHIE. p. 170

..... p. 171

Imprimé en France

par
l'Institut National de Recherche en Informatique et en Automatique