4.4 **Méthodes**

On distingue dans l'interprète trois types de méthodes:

- les méthodes SMALLTALK,
  dont l'énoncé sera donné en SMALLTALK, ce seront donc les méthodes utilisateur, analogues aux EXPR en LISP. Elles seront définies à l'aide du sélecteur *under|stand|*, et le lecteur produira un appel à la fonction *react* (par exemple la méthode *fac* vue précédemment).

- les *méthodes primitives*,
  analogues aux SUBR en LISP, leur corps sera cette fois spécifié dans le langage d'implémentation, (actuellement Le_Lisp, et plus tard META). Elles sont définies selon le même protocole que les précédentes, le lecteur reconnaissant les structures parenthésées comme des données.

- les méthodes de *sélecteur primitif*,
  la recherche de la méthode associée dans la hiérarchie est alors lancée au niveau du lecteur et non de l'évaluateur comme pour les précédentes, et devient de ce fait définitive. Ces méthodes, en court-circuitant une recherche dynamique dans la hiérarchie, permettent une plus grande efficacité dans leur traitement, leur corps étant expansé en un appel *direct* d'une fonction dans le langage cible. Elles sont définies (ainsi que l'arité du sélecteur) à l'aide du sélecteur *pri|mary:stand|*. Un exemple en est le sélecteur <- qui sera expansé en un appel de la fonction *setq*.

Toutes ces méthodes sont totalement accessibles à l'utilisateur par les messages *methodfor|*, *forget|*, *primaryforget|*, ... qui peut ainsi totalement reconfigurer cette répartition, et rétablir une recherche dynamique où il le désire.

Dans certains cas, on voudrait une expansion plus sélective, par exemple pour l'opérateur générique +, la sémantique associée étant différente suivant le type du receveur. Il suffit alors de définir modulairement les différentes méthodes pour chaque type:

```
number pri| + mary: 2 stand| (mcons '+ self l-args);
  under| (+ n) stand| (+ self n).
string pri| + mary: 2 stand| (mcons 'catenate self l-args);
  under| (+ s) stand| (catenate self s)
list pri| + mary: 2 stand| (mcons 'append self l-args);
  under| (+ l) stand| (append self l) !
```

A l'apparition d'un message de sélecteur +, le lecteur trouvant plusieurs méthodes associées à ce sélecteur primitif, va tenter d'en sélectionner une en tentant de déterminer le type du receveur et sinon de l'argument dans l'ensemble des types associés aux classes considérées i.e. *number*, *string* et *list*.
Dans l'affirmative le lecteur produira la forme primitive associée, et dans la négative un

appel à *react*, la recherche de la méthode étant alors dynamique lors de l'évaluation:

```
[a + (2 3)]  ☛ (append a '(2 3))
[a + b]      ☛ (react a '+ b)
```

Cet exemple montre la souplesse et la modularité de ce modèle.

4.5 **Auto-génération**

L'interprète propose une *méta-description* de son univers, les classes résidentes étant créées par instanciation au même titre que celles définies par l'utilisateur.
Plus encore qu'un souci esthétique et pédagogique, cela est motivé par la volonté de rendre accessible la construction du langage à l'utilisateur permettant une transparence et une possibilité d'intervention à tous les niveaux de l'interprète. L'interprète s'auto-génère donc à partir d'un boot-strap minime, construction provisoire du squelette des classes *metaclass* et *object* et des méthodes *new* et *under*| *stand*|. Cela permet l'auto-génération et description cette fois extensive de la classe **metaclass**, premier procréateur de l'univers, suivie de la création de **class**:

```
metaclass new name| metaclass;
  under| new stand| (create-class self);
  under| (name| name) stand| (progn (put-class-name self name) (set name self));
  under| (under| selector stand| body) stand|
    (if (atom selector) (create-method self selector () () body)
      (let ( (keyword) (var) )
        (while selector
          (newl keyword (nextl selector))
          (newl var     (nextl selector)))
        (create-method self (apply 'concat (nreverse keyword)) (nreverse var) () body)));
  under| (pri| selector mary: arite stand| call) stand|
    (create-primary-method self selector arite call);
                                    ...
  under| name stand| (class-name self);
  under| (subclassof: superclass) stand| (put-superclass self superclass);
  under| subclassof stand| (superclass self);
  under| (fields| fields) stand| (put-var-fields self fields);
  under| fields stand| (var-fields self);
  under| selectors stand| (selectors self);
  under| (methodfor| selector) stand| (pprint (get-method selector self)) !
```

```
metaclass new name| class;
  subclassof: metaclass;
  under| new stand| (create-object self);
  under| (newval: values) stand| (create-object-val self (var-fields self) values) !
```

puis la création des classes **variable** et **object**, les classes **number**, **string** et **list** n'étant pas présentées ici pour des raisons de place

```
class new name| variable;
  under| undefined-variable-error stand| (smalltalk-error self "undefined variable");
  pri| <- mary: 2 stand| (mcons 'setq self l-args)      , sélecteurs primitifs
  pri| value mary: 1 stand| (list 'cval self);
                                                ...
  pri| input mary: 1 stand| (list 'input (and self (kwote (concat self ".stk")))) !
```

Le code LISP des méthodes primitives et des sélecteurs primitifs apparaît en italique.

```
class new name| object;
  under| init stand| none;                                  , pas d'initialisation par défaut
  under| (? field) stand| (cval field);                      , méthodes primitives
  under| (? field <- value) stand| (set field value);
  under| class stand| self-class;
  under| (values| values) stand|
    (put-val-fields self (var-fields self-class) values);
                                                ...
  pri| end mary: 0 stand| (exit end-smalltalk);              , sélecteurs primitifs
  pri| then|else| mary: 3 stand| (mcons 'if self l-args);
  under| pretty stand|                                       , méthode SMALLTALK
    ["I am a " prin. self is print.
     self-class class fields print. self-class values print.
     self-class fields print. self values print. "done!"] !
```

5. EXEMPLE

Présentons maintenant l'exemple introduit par Alan Kay[Kay76a] et maintenant devenu très classique[9] : la classe **box**, analogue de la factorielle (*foo*) en LISP, ou de la drosophile et sa mutation *bar* en génétique.

```
[metaclass new subclassof: class;          , crée une métaclasse anonyme
  fields| (c);                             ,  un champ: le caractère utilisé
  return] new name| box;                   , retour en valeur, et instanciation
   values: ("@");                          ,  ce sera la brique des boxes
   fields| (o wid ht);                     ,  spécification du type box
   under| init stand| [o <- wid <- ht <- 4];
   under| (t: x op: y) stand| [x spaces. [[c + " "] dupl y] print];
   under| (in: x si: y de: z) stand|
    [z - 2 repeat| [x spaces. c prin. [y * 2 - 3] spaces. c print]];
   under| draw stand| [self t: o op: wid; in: o si: wid de: ht; t: o op: wid. "nice?"];
   under| (move: x) stand| [o <- o + x. self draw];
   un| turn der| (a) stand| [a <- wid. wid <- ht. ht <- a. self draw];
   under| (gr: x ow: y) stand| [wid <- wid + x. ht <- ht + y. self draw];
   under| (zoom: coeff) stand|
    [coeff >= 0 then| [wid <- wid * coeff. ht <- ht * coeff]
            else| [coeff <- coeff -. wid <- wid / coeff. ht <- ht / coeff].
     self draw] !
```

On remarque que les variables de classes sont ici incarnées par les champs de la classe *box*, ces champs étant spécifiés par la métaclasse (que l'on décide de laisser anonyme) génératrice de *box*.

9. que l'on retrouvera par exemple dans[Cointe81a, Borne82a] et[Ferber83a].

```
? (smalltalk)                          ; en route
Meta-Smalltalk se meta-construit sous vos yeux!
j'ai charge metaclass.stk
j'ai charge class.stk
j'ai charge variable.stk
j'ai charge object.stk
j'ai charge rock.stk
Bienvenue dans le monde merveilleux de Meta-Smalltalk
pour sortir  # end!        pour charger un fichier:  # <fichier> load!
comme ex, chargez fac, compteur, point et box
#
# box load !
> box.stk
# isa <- box new !
> *a_box*
# box selectors !
 zoom: gr:ow: turn move: draw in:si:de: t:op: init
> done!
# box is !                             , box: instance non terminale anonyme,
> *anonymous*
# box class !
> *a_metaclass*                        , sa métaclasse n'étant pas nommée
# isa draw !
   @ @ @ @
   @       @
   @       @
   @ @ @ @
> nice?
# isa pretty !
I am a box
(c)                                    , champs (variable puis valeur) de la classe box
(@)
(o wid ht)                             , champs de l'instance isa
(4 4 4)
> done!
# box pretty !
I am a *anonymous*
()
()
(c)
(@)
> done!
```

```
# pie <- box new; zoom: 2 !
   @ @ @ @ @ @ @ @
   @               @
   @               @
   @               @
   @               @
   @               @
   @               @
   @ @ @ @ @ @ @ @
> nice?
# pie ? wid <- 3; draw !                    , accès aux champs de pie
   @ @ @
   @   @
   @   @
   @   @
   @   @
   @   @
   @   @
   @ @ @
> nice?
# box ? c <- "%". isa draw. pie turn !      , accès au champ de box
   % % % %
   %       %
   %       %
   % % % %
   % % % % % % % %
   %               %
   % % % % % % % %
> nice?
# pie ? c !                                 , pie y a aussi accès
> %
# pie ? c <- "$"; draw. isa draw !
   $ $ $ $ $ $ $ $
   $               $
   $ $ $ $ $ $ $ $
   $ $ $ $
   $     $
   $     $
   $ $ $ $
> nice?
```

Les instances ont aussi accès aux champs de leur classe, communs à toutes ses instances. Cela permet de définir des variables de classe, sans introduire une nouvelle structure.

```
# box class new name| boite;                    , un nouvel ensemble de boîtes
#  subclassof: box;
#  values: ("o") !                              , distinct du précédent
> (o)
# bepaul <- boite new; draw. isa draw !
   o o o o
   o     o
   o     o
   o o o o
   $ $ $ $
   $     $
   $     $
   $ $ $ $
> nice?
# end !
= Que Meta-Smalltalk soit avec vous
? (end)
Que Le_Lisp soit avec vous.
$
```

On crée un nouveau type de boîtes, dont toutes les instances auront le même caractère (actuellement le point) pour dessiner leurs cadres.

Pour une meilleure compréhension, voici comment les divers objets qui viennent d'être créés prennent place dans l'arbre d'instanciation:

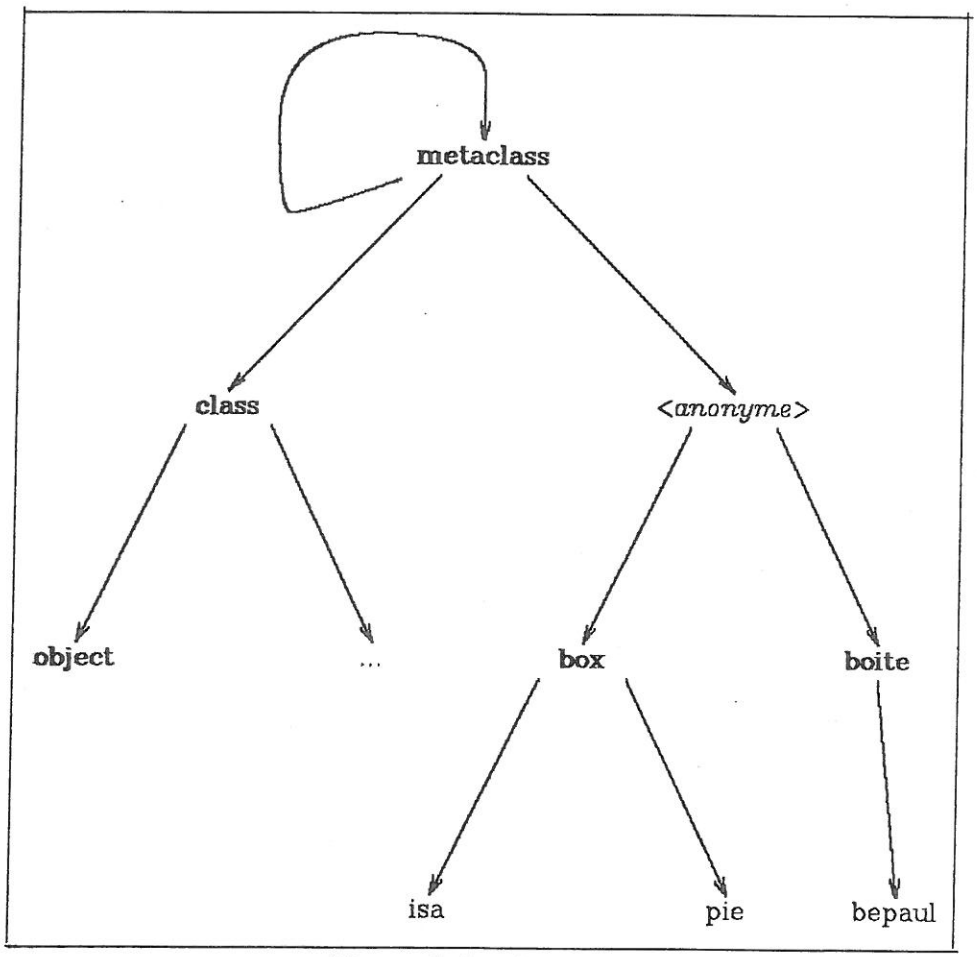

Figure 4. Les instances

Et maintenant l'arbre hiérarchique:

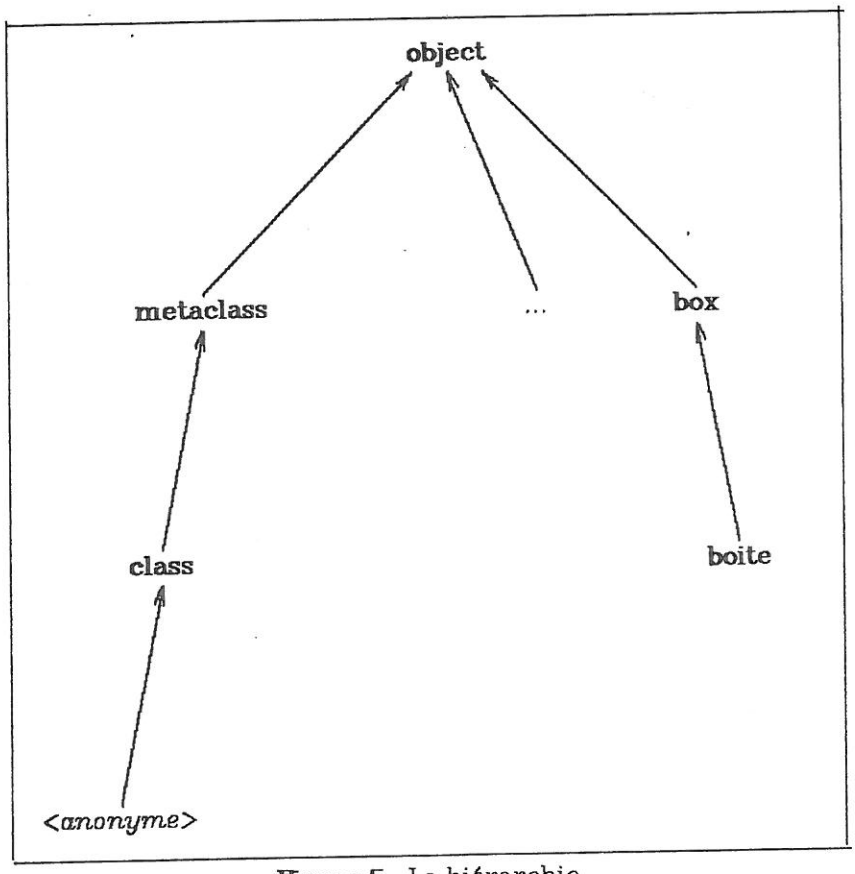

Figure 5. La hiérarchie

6. CONCLUSION

Nous n'avons décrit ici que certains des aspects du langage, la place nous manque en effet pour décrire de manière plus approfondie l'implémentation (sinon cet exposé déjà long le serait bien plus encore!). De plus cet exposé apparaît n'être qu'un "état des travaux", l'interprète présenté n'étant qu'un premier prototype du langage en cours de développement, la phase ultérieure étant l'écriture en META (et en français!) de ce prototype.

7. REFERENCES

[Barthes53a] R. Barthes, "Le degré zéro de l'écriture," dans *Bibliothèque Méditations*, ed. Editions Gonthier, (1953).

[Bobrow83a] D. Bobrow and M. Stefik, *The LOOPS Manual*. December 83.

[Borne82a] I. Borne, *Manuel Smalltalk-76*, Université PARIS-6 (Juillet 1982). Mémoire de DEA

[Borne84a] I. Borne, *Micro-Smalltalk pas-à-pas*, draft Juin 84.

[Borning79a] A. Borning, "THINGLAB -- A Constraint Oriented Simulation Laboratory," SSL-79-13, Palo Alto Research Center (July 1979). Revised Thesis

[Borning82a] A. Borning and D. H. Ingalls, "Multiple Inheritance in SMALLTALK-80," pp. 234-237 dans *Proceedings of the AAAI-82*, Pittsburgh (August 82).

[Briot83a] J.P. Briot, "L'instanciation dans les langages objets," *Bigre*, (37) pp. 173-209 (Décembre 83).

[Cannon7?a] H.I Cannon, *Flavors A non-hierarchical approach to object-oriented programming*, Draft 197?.

[Chailloux78a] J. Chailloux, "VLISP 10.3, Manuel de Référence," RT-16-78, Université Paris 8 - Vincennes (Avril 1978).

[Chailloux84a] J. Chailloux, *Le_Lisp Version-14*, INRIA (Février 1984).

[Chailloux83a] J. Chailloux, *Le_Lisp 80 (version 12)*, INRIA rapport technique No: 27 (Juillet 83).

[Cointe81a] P. Cointe, "Fermetures dans les λ—Interprètes : Application aux Langages LISP, PLASMA et SMALLTALK (thèse de 3ième cycle)," LITP 82-11, Université Paris VI, Paris (Décembre 1981).

[Cointe82a] P. Cointe, "Une réalisation de SMALLTALK en VLISP," *TSI* **Volume 1**(4) pp. 325-340 (Juillet-Aout 1982).

[Cointe83a] P. Cointe, "A VLISP Implementation of SMALLTALK-76," pp. 89-102 dans *Integrated Interactive Computing Systems*, ed. P. Degano & E. Sandewall, North-Holland, Amsterdam, New York, Oxford (1983).

[Cointe84b] P. Cointe, "Une extension de Vlisp par les Objets," *Science of Computer Programming*, (161)North Holland, (1984).

[Cointe84c] P. Cointe and X. Rodet, *Formes: an Object & Time Oriented System for Music Composition and Synthesis*, Conference Record of the 1984 ACM symposium on LISP and Functional Programming, Austin (Texas) (5-8 August 84).

[Cointe84d] P. Cointe, *Implémentation et interprétation des langages objets, application aux langages Formes, ObjVlisp et Smalltalk (thèse d'Etat).*, Université Paris VI (Décembre 84). draft

[Cointe84a] P. Cointe, "Etude de la programmation orientée objet, Application à Objvlisp et Smalltalk.," Département Informatique, Université Paris 8 - Vincennes, St-Denis (Janvier 84). Support de cours, UV Langages orientés Objets (M9385)

[Ferber83a] J. Ferber, *MERING: un langage d'acteur pour la représentation et la manipulation des connaissances (thèse de docteur ingénieur)*, Université Paris-6 Décembre 83.

[Goldberg83a] A. Goldberg and D. Robson, *SMALLTALK-80 The Language and its Implementation*, Addison-Wesley Publishing Compagny (1983).

[Greussay82a] P. Greussay, "Le Système VLISP-UNIX," Département Informatique, Université Paris 8 - Vincennes (Février 1982). Draft

[Kay76a] A. Kay and A. Goldberg, "SMALLTALK-72 Instruction Manual," SSL 76-6, Xerox Palo Alto Research Center, Palo Alto, CA. (March 1976).

[Krief84a] P. Krief, *Une compilation de LOGO en LISP*, Université PARIS-6 (Juillet 1984). Mémoire de DEA

[Lieberman81a] H. Lieberman, "A Preview of Act1," AI Memo No. 625, MIT (June 1981).

[Moon80a] D. A. Moon and D. Weinreb, "Flavors: Message Passing In The Lisp Machine," AI Memo No 602, MIT (November 1980).

[Reynolds72a] J. C. Reynolds, *Definitional Interpreters for higher-order programming languages*, Proc. 25th ACM National Conference , New York (1972).

[Theriault81a] D. G. Theriault, "A primer for the Act 1 language," Working paper 221, MIT (June 1981).

[Theriault83a] D. G. Theriault, "Issues in the Design and Implementation of Act2," Technical Report 728, MIT (June 1983).